Fraunhofer IAIS

Roberta® Lernen mit Robotern

Stausimulation

Tutorial

Diese Stau-Simulation kann mit dem Open Roberta Lab und LEGO MINDSTORMS EV3-Robotern durchgeführt werden.

Das Ziel ist es, die Roboter so zu programmieren, dass sie sich wie Fahrer im Straßenverkehr (allerdings in einem zirkulären Verlauf) verhalten, durch Beschleunigen und Bremsen soll der Verkehr zwischen Stau und fließendem Verlauf variieren und ein bestimmtes Muster zeigen - ganz ohne Einfluss von Außen!

Die folgende Schritt-für-Schritt Beschreibung des Programms kann als Anleitung verwendet werden, um diese Simulation mit EV3-Robotern und dem OpenRobertaLab selber durchzuführen!

1 Aufgabenstellung 2
2 Roboter und Strecke vorbereiten 3
3 Grober Aufbau des Programms 5

1 Aufgabenstellung

In der Verkehrsforschung ist Entstehung und Auflösung von Staus ein sehr relevantes Thema. Unabhängig von externen Faktoren, sind auch welche bekannt, die durch das Fahrverhalten der Verkehrsteilnehmer selbst entstehen.
Die Idee dieser Simulation ist eine Darstellung solcher Situationen mit Robotern, die dieses Verhalten aufweisen.

Eine grundsätzliche Ursache ist das Sättigungsproblem. Auf jede Strecke passt nur eine bestimmte Anzahl von Verkehrsteilnehmern, wenn diese mit einer Idealgeschwindigkeit konstant fahren. Sobald jemand schneller oder langsamer fährt, sinkt diese Kapazität, was Stau hervorrufen kann. Vor Allem konstantes Fahren ist unrealistisch. Viele Autofahrer fahren zu dicht auf, was sie dann dazu zwingt, schnell abzubremsen. Die aufschließenden Fahrer müssen auch bremsen, bis der Vorfahrende wieder beschleunigen kann. Das löst sogar stockenden Verkehr in Abschnitten aus, die viel weiter hinten liegen. Selbst wenn der ursprüngliche Auslöser nicht mehr vorhanden ist, wirken sich die Folgen immernoch auf den Verkehr aus. Der Stau wird zusätzlich durch unregelmäßiges Anfahren gefördert.

Mehr Details zu Stauentstehung, sowie weitere Ergebnisse der Forschung an diesem Thema können zum Beispiel hier nachgelesen werden:
Die drei Haupttheoreme der Stauforschung

Bei der Umsetzung mit Robotern soll beobachtet werden, wie lange es dauert, bis sich ein Stau löst und ein normaler Verkehr wieder einpendelt. Damit nicht ständig Stau herrscht, muss die Kapazität der Bahn bei Maximalgeschwindigkeit ermittelt werden.

Für die Programmierung der Roboter haben wir entschieden, folgende Randbedingungen zu beachten:

- **Beschleunigung**: Aus dem Stillstand wird die Geschwindigkeit langsam erhöht , bis die Maximalgeschwindigkeit erreicht ist. Diese wird dann bei freier Bahn beibehalten.
- **Bremsen**: Wenn der Abstand zum Vordermann zu klein ist, bleibt der Roboter komplett stehen (bis der Abstand wieder groß genug ist).
- **Trödeln** (als "Stau-Faktor"): Es besteht die Wahrscheinlichkeit, dass Fahrer nicht direkt oder regelmäßig beschleunigen. Die Roboter sollen nicht ständig trödeln, sondern unregelmäßig - allerdings mit einer bestimmten Wahrscheinlichkeit (in unserem Fall 15%)

Zur Umsetzung dieser Bedingungen, müssen die Roboter grundsätzlich erstmal in der Lage sein, einem Straßenverlauf zu folgen und den Abstand zum Vordermann messen zu können.
Deswegen werden Lichtsensor und Ultraschallsensor verwendet.

Es ist manchmal praktisch, wenn man während der Ausführung auf dem Roboter beobachten kann, welcher Schritt im Programm gerade durchlaufen wird. Dazu kann man das Programm stellenweise "markieren" - dazu wird die Statusleuchte verwendet.

Ziel dieses Experiments ist eine Verkehrssituation mit Robotern auf einer zirkulären (endlosen) Strecke zu simulieren. Insbesondere kann man beobachten, durch welches Fahrverhalten ein Stau entsteht, der keinen von Außen beeinflussten Ursprung hat.

Das ganze Programm kann mit dem Open Roberta Lab geschrieben werden.

Die Lösungshinweise dienen als Schritt-für-Schritt Anleitung und Vorlage für das Programm.

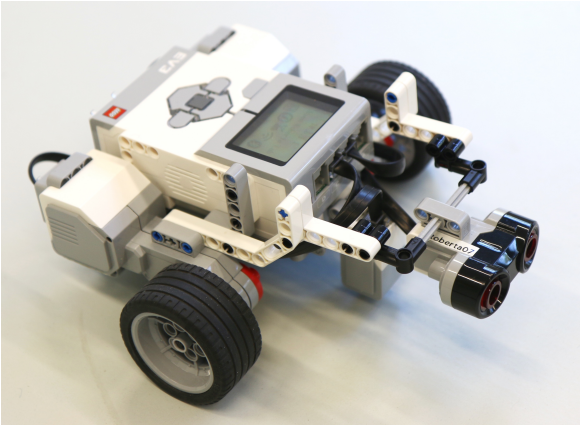
2 Roboter und Strecke vorbereiten

Grundaufbau
Für das Experiment werden LEGO MINDSTORMS EV3-Roboter (mit differential drive) verwendet, die nach einem Standard-Modell aufgebaut sind.

In den folgenden Schritten wird u.A. beschrieben, wie die Sensoren angebracht werden müssen.

Für den Grundaufbau ist es ausreichend, einen fahrenden Roboter mit zwei Motoren und einem Kugelrad zu bauen. Nach unserem Modell kann man sich nach diesen Werten richten:
- Raddurchmesser: 5.6 cm
- Spurbreite: 16 cm

Es ist auch möglich, den Roboter anders aufzubauen, allerdings muss man sich dabei im Klaren sein, dass Raddurchmesser und Spurbreite dann nachgemessen und in der Roboterkonfiguration richtig eingetragen werden müssen.
Außerdem werden veränderte Werte das Fahrverhalten der Roboter beeinflussen. Das heißt, dass man im Programm möglicherweise Konstanten (z.B. die Variable kp) anpassen muss.



Sensoren anbringen
Für diese Simulation wird der Farbsensor benötigt, um einer Linie auf dem Boden zu folgen.

Der Sensor wird mittig, vorne am Roboter angebracht. Er darf nicht zu weit nach vorne hervorstehen (also nah am EV3-Brick angebracht werden) und ist mit ca. 2cm Abstand vertikal auf den Boden gerichtet.

Der Ultraschallsensor misst in der Simulation den Abstand zum Vordermann.

Er wird also gerade nach vorne gerichtet und ebenfalls mittig vor den EV3-Brick gebaut. Dabei muss man darauf achten, dass er nicht zu hoch sitzt, damit andere Roboter vor ihm im "Blickfeld" liegen. Der Ultraschallsensor liegt also etwas niedriger, als die obere Kante des EV3-Bricks.

Eine möglichst stabile Befestigung ist für beide Sensoren empfohlen, da so ungenaue Werte, die durch Wackeln entstehen können, vermieden werden.

Es ist wichtig zu kontrollieren, ob die Roboterkonfiguration im Open Roberta Lab zum tatsächlichen Aufbau des Roboters passt. ***In diesem Fall sollte die Einstellung beider Motoren für Regulierung auf nein umgestellt werden!***

Fahrbahn markieren
Die Roboter werden auf einer schwarzen Markierung im Kreis fahren. Damit man einen Stau simulieren kann, ist es sinnvoll, mindestens 3 Roboter mit dem Programm zusammen auf der Kreisbahn fahren zu lassen.

Wichtig bei der Markierung der Strecke ist, möglichst keine scharfen Kurven einzubauen, um zu vermeiden, dass die Roboter von der Bahn abkommen. Außerdem muss ein gewisser Kontrast zwischen Farbe der Markierung und Boden bestehen. Ideal ist eine schwarze Linie auf gleichmäßig weißem Untergrund.

Gut zum Markieren ist ein schwarzes Klebeband mit einer Breite von ca. 5cm.

Als Daumenregel passen drei Roboter auf eine kreisförmige oder ovale Strecke mit einer Länge von ca. 2,35 m (und das kann man dan hochrechnen auf die Zahl der Roboter, die man einsetzen möchte. In unserem Experiment sind acht Roboter gefahren.)

3 Grober Aufbau des Programms

Unser Hauptprogramm hat drei unter-Programme, die in einer endlos -Schleife wiederholt abgerufen werden.

- Erst muss der Roboter einer Linie folgen können (P-Regler).
- Dann muss die Geschwindigkeit reguliert werden - dafür brauchen wir eine Funktion zur Beschleunigung.
- Zum Schluss wollen wir noch den Trödelfaktor einbauen.

Zunächst brauchen wir also eine Schleife (Wiederhole unendlich oft).

Damit der Roboter dem "Straßenverlauf" folgen kann, also einer schwarzen Markierung auf weißem Untergrund, muss der Lichtsensor abgefragt werden und passende Befehle müssen dann an die Motoren weitergegeben werden.

Der Roboter soll ohne gezackte oder wellige Bewegungen der Linie folgen, also wird ein P-Regler verwendet (genauere Beschreibung im nächsten Lösungshinweis), der eine proportionale Funktion verwendet.

Ganz zum Schluss haben wir noch mit der Statusleuchte Markierungen eingefügt, damit man während der Programmausführung sehen kann, an welcher stelle des Programms der Roboter sich gerade befindet.

In den folgenden Lösungshinweisen wird beschrieben, wie man graduell das Programm aufbaut, bis es die Aufgabe korrekt ausführt.

Mit einem passenden Modell kann das Programm also so erarbeitet oder direkt übernommen werden - Sie können das fertige Programm hier runterladen.
(Die Datei Stausimulation.zip enthält das NEPO-Programm Stausimulation.xml, welches im Open-Roberta-Lab hochgeladen werden kann.)

(Diese Datei kann im Open Roberta Lab hochgeladen und von dort an die Roboter geschickt werden. //Achtung: Die Roboterkonfiguration sollte im Open Roberta Lab kontrolliert und angepasst werden - insbesondere wird der Roboter nur richtig laufen, wenn die Motoren nicht reguliert sind! Auch die Variablen kp und offset sollten angepasst werden. Wie das geht, wird in diesem Tutorial beschrieben.)

P-Regler

(Eine detaillierte Erklärung eines P-, bzw. eines PID-Reglers kann man hier finden!)

Ein PID-Regler ist nichts anderes, als ein P-Regler, der ein bisschen mehr kann. Für diese Simulation ist dieser aber weder notwendig, noch sinnvoll.)

Der P-Regler hat drei Aufgaben:

- *Er liest kontinuierlich Werte vom Lichtsensor ab*
- *berechnet daraus einen Fehler*
- *korrigiert damit die Fahrtrichtung des Roboters, indem er die Motoren reguliert.*

Eigentlich folgt der Roboter nicht der Linie, sondern ihrem Rand!
Wir brauchen also den Licht-Wert (die Helligkeit) des Randes der schwarzen Linie, damit der Roboter weiß, nach welchem optimalen Wert er sich richten soll. Diesen Wert nennen wir *offset*.

Erstelle folgende Variablen:

offset:

Aus Perspektive des Roboters ist ein Rand nur eine Fläche, die in weiße und schwarze Pixel unterteilt ist. Daraus berechnet der Farbsensor einen Mittelwert, der für unser Auge ziemlich grau aussieht!

In einer perfekten (unrealistischen) Situation läge *offset* bei 50 (weiß ist 100, schwarz 0).
In der Realität sollte der Wert für weiß und schwarz erst gemessen werden. Dann ist

offset = Schwarz + (Weiß-Schwarz)/2.

error:
Die *offset* Variable brauchen wir, um einen *error* zu berechnen, und zwar die Differenz zwischen *offset* und dem tatsächlichen Sensor-Wert (*lightvalue*) im Moment der Messung.
(Wir machen *offset* zu einer globalen Variable, also im Hauptprogramm. Sie wird allerdings nur im Unterprogramm vom P-Regler verwendet.)

lightvalue:
Der *lightvalue* ist eine eigene Variable, die direkt den Wert vom Lichtsensor abfragt.

turn und kp:
Damit der Regler jetzt proportional zu der Abweichung des Lichtwerts die Geschwindigkeit der Motoren anpasst, wird eine weitere Variable *turn* berechnet:

*Turn = Kp * error*

Diese Funktion stellt die proportionale Beziehung zwischen Lichtwert und Motoren mit einer linearen Funktion (y=mx+b) dar.
Dabei ist *Kp* die Konstante (m), welche die Steigung angibt, *turn* ist der y-Wert, der bestimmt, wie viel die Motoren korrigiert werden müssen und *error* der x-Wert, der für die Abweichung vom Rand der Linie steht.
(b ist der y-Achsenabschnitt, in unserem Fall aber 0, also kann er weggelassen werden.)

In unserer Anwendung muss *turn* steigen, da mit zunehmender Abweichung, die Fahrtrichtung zunehmend korrigiert werden muss!
Das heißt, *Kp* muss eine positive Zahl sein. Je größer *Kp* ist, desto stärker korrigiert der Roboter seine Bahn. Der perfekte *Kp* Wert muss durch Ausprobieren gefunden werden - er kommt auch auf Geschwindigkeit und Radabstand an!
(Auch *kp* ist bei uns eine globale Variable im Hauptprogramm.)

tp:
Im letzten Schritt, wird mit *turn* die Geschwindigkeit für die Motoren berechnet.
Wenn beide Motoren auf die selbe Geschwindigkeit eingestellt sind, fährt der Roboter geradeaus. Man sollte sich also erstmal die Geschwindigkeit überlegen, mit der grundsätzlich gefahren werden soll. Wir nennen sie *Tp* (target power, also Ziel-Geschwindigkeit).

Der Roboter soll ja nur gerade fahren, wenn kein error besteht, also keine Richtungsveränderung notwendig ist.
Gibt es aber einen error-Wert, muss jeweils ein Motor einen höheren Wert bekommen, als *Tp*, und der andere einen niedrigeren. Das führt dazu, dass der Roboter eine Kurve fährt!

Ein "Motor Port" sollte also an Tempo *Tp* + *turn* gestellt werden, der andere an Tempo *Tp* - *turn*.

Hier ein kleines Beispiel, um einen Durchlauf vom P-Regler darzustellen:

- Wir nehmen an, der Lichtsensor gibt bei weiß einen Wert von 65 und bei schwarz 5.
  offset = 5 + (65-5)/2 = 35

- Unsere gewünschte Geschwindigkeit ist 50 und wir haben herausgefunden, dass für uns eine proportionalitäts-Konstante von 0.8 gut funktioniert.
  tp = 50
  kp = 0.8

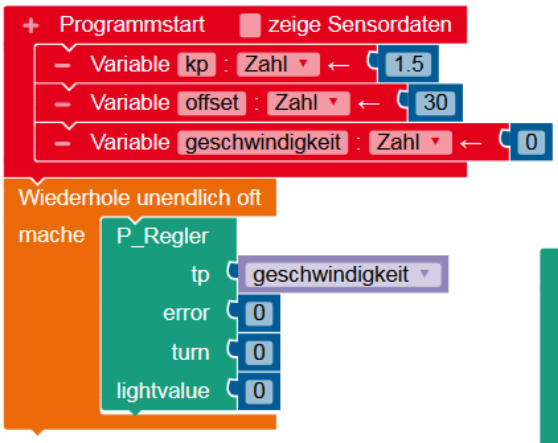
- Zunächst wird der aktuelle Wert des Lichtsensors abgefragt (nehmen wir an, der Roboter fährt etwas neben dem Linienrand und misst 50)
  lightvalue = 50

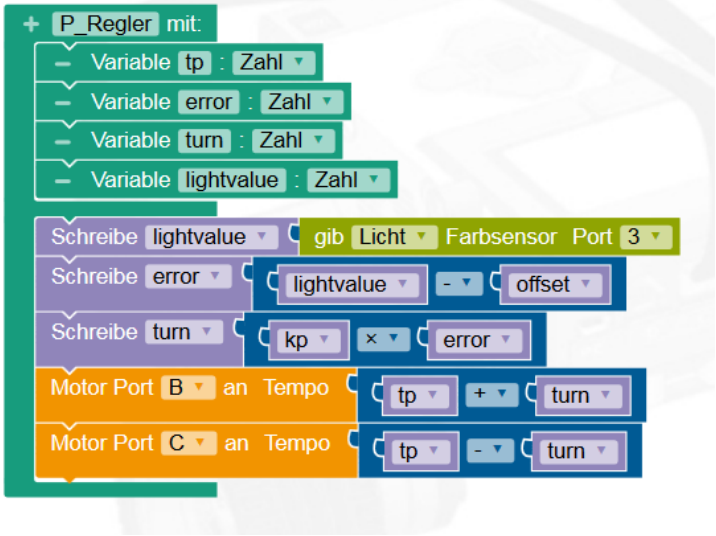
- Dann wird der error berechnet:
  error = lightvalue - offset = 50-35=15

- Dann wird turn berechnet:
  *turn = kp * error = 0.8*15=12*

- Und das geht dann in die Geschwindigkeitsberechnung für Motoren B und C ein:
  Motor Port B an Tempo = *tp + turn = 50+12 = 62*
  Motor Port C an Tempo = *tp - turn = 50-12 = 38*
  (Zur Kontrolle: 62 + 38 = 100 und 100/2 = 50)

Das Programm sollte nun ungefähr so aussehen:
(Die Variable *geschwindigkeit* wird im nächsten Schritt erst eingefügt. Die Variable *tp* kann zum Ausprobieren des P-Reglers beispielsweise auf 20 gestellt werden.





Geschwindigkeitsanpassung

Aber zurück zum Hauptprogramm: Jetzt, wo der Roboter der Linie folgen kann, müssen wir ihn dazu bringen, seine Geschwindigkeit anzupassen!

Geschwindigkeit:
Wir brauchen zunächst eine globale Variable *geschwindigkeit*, die einen Anfangswert von 0 hat.
(Im endeffekt wollen wir diese Variable im P-Regler als Zielgeschwindigkeit *Tp* verwenden.)

In unsere Endlos-Schleife fügen wir vor dem P-Regler nun eine Funktion ein, die jede paar Sekunden die *geschwindigkeit* manipuliert.

Timing:
Damit wir das Timing beeinflussen können, fügen wir erst eine Bedingung ein:
Wenn der Timer 500ms (dieser Wert kann angepasst werden) erreicht oder überschreitet, soll der Abstand zum Vordermann gemessen werden und *geschwindigkeit* wird angepasst.

Dann muss der Timer zurückgesetzt werden, damit er erneut anfängt zu zählen, und bei Wiederholung die Bedingung wieder erfüllt werden kann.

Ultraschallsensor abfragen:
Wenn die Timer-Bedingung erfüllt ist, soll also der Ultraschallsensor abgefragt werden. Dann muss, je nach Sensorwert, eine Entscheidung getroffen werden (wenn (Bedingung x erfüllt), mache (Aktion y), sonst (Aktion z)).

Unsere Bedingung, wenn wir zurück an die Richtlinien denken, kann als Frage formuliert werden:
Ist der Abstand (direkte Abfrage vom Ultraschallsensor) größer oder gleich 20cm (unser beliebig gewählte Mindestabstand zum Vordermann)?

Wenn ja, dann soll der Roboter anhalten, also *geschwindigkeit* auf 0 setzen. Wenn nein, sollte er beschleunigen, also *geschwindigkeit* auf einen Wert setzen, der von einem *Unterprogramm "Beschleunigung"* berenchnet wird.

Beschleunigung: Diese Funktion hat die Besonderheit, dass ein konkreter Zahlen-Wert zurückgegeben wird, und zwar *beschleunigung*!
(Zur Erinnerung: Die *geschwindigkeit* wird in der Hauptfunktion in manchen Fällen auf *beschleunigung* gesetzt.)

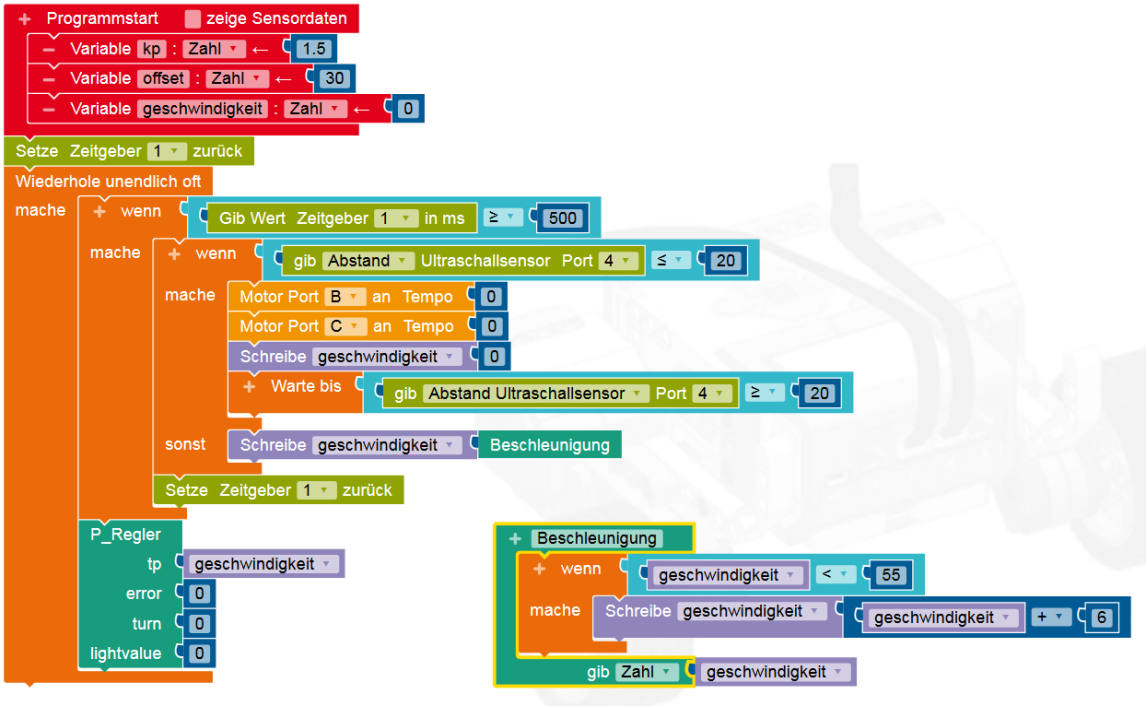
Dafür brauchen wir eine Bedingung (wenn x, (dann) mache y).
Unsere Bedingung ist, dass die vorherige Geschwindigkeit (also den Wert der Variable *geschwindigkeit*) unter der Maximalgeschwindigkeit (beliebiger Wert, in unserem Fall 55) liegt.
Nur dann soll nämlich beschleunigt werden. Ansonsten wird die Variable nicht verändert und der Roboter behält seine Geschwindigkeit bei.

Fährt er allerdings bei "freier Bahn" noch nicht schnell genug, wird die Geschwindigkeit erhöht. Wir verändern also die Variable folgendermaßen:

geschwindigkeit = geschwindigkeit + 6

(6 ist auch ein beliebiger Wert. Mit einem größeren Wert, beschleunigt der Roboter schneller. Allerdings kann es dann auch sein, dass er fast schon sofort mit Maximalgeschwindigkeit fährt, weil das Programm so schnell durchlaufen wird und 500ms nicht sehr lang sind.)

Es ist ratsam, den Beschleunigungswert mal zu verändern und auszuprobieren. Auch kleine Veränderungen können einen Stau hervorrufen oder vermeiden.



Jetzt kann der Roboter eigentlich schon gut fahren - er folgt der Linie, kontrolliert alle 500ms den Abstand zum Vordermann und beschleunigt bis zur Maximalgeschwindigkeit oder bremst ab.

Trödelfaktor

Nach unseren Richtlinien, verhält sich der Roboter noch zu gut. Genau - der Trödelfaktor fehlt noch!

Zunächst wollen wir die Möglichkeit haben, beim Start des Programms auszuwählen, ob der Roboter mit oder ohne *Trödelfaktor* fahren soll - wir schreiben ein Unterprogramm:

Trödelfaktor:
Zunächst sollen die Möglichkeiten auf dem Bildschirm angezeigt werden:

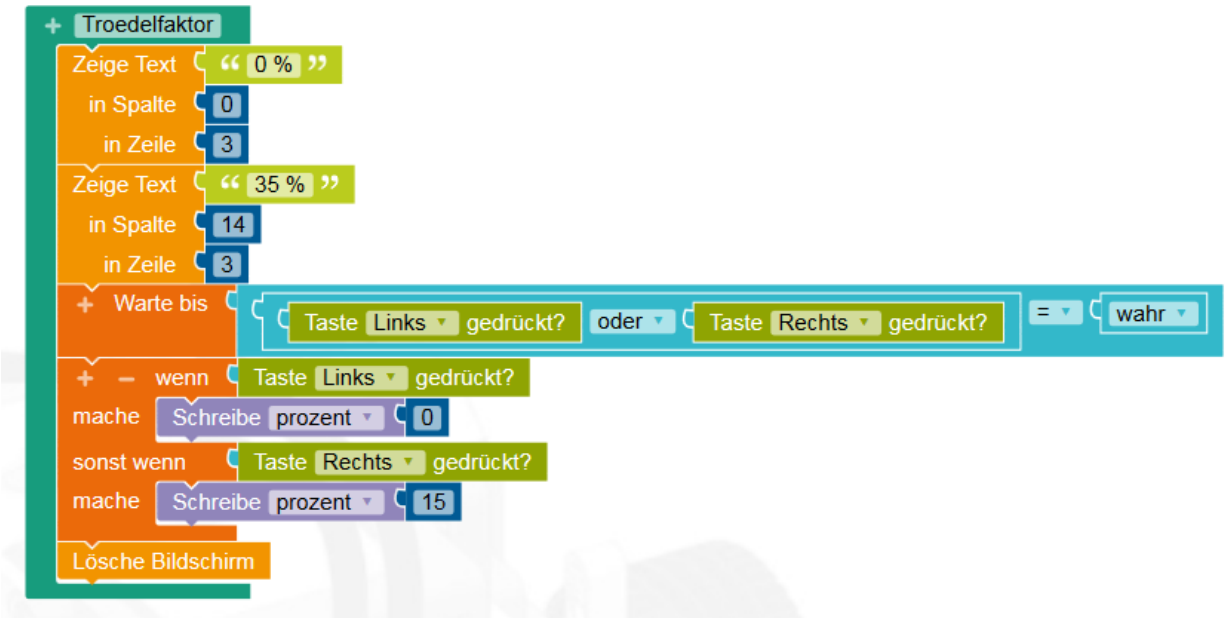
0% Trödelfaktor (also ohne), wenn man auf die rechte Taste (auf dem EV3 Brick) drückt, und 15% Trödelfaktor (mit Trödelfaktor, die 15% werden später erklärt), wenn man auf die linke Taste drückt.

(Tasten wurden hier beliebig gewählt, dass kann man natürlich auch anders machen!)

Diese Entscheidungsmöglichkeit wird durch Einfügen einer Bedingung ermöglicht:
Wenn (Taste rechts gedrückt, dann) mache (setze *prozent* zu 0), sonst wenn (Taste links gedrückt, dann) mache (setze *prozent* zu 15).

Die Variable prozent sollte im Hauptprogramm erstellt werden, da sie dann im weiteren Programmverlauf die Beschleunigungs-Funktion beeinflussen soll:

Zum Schluss kann der Bildschirm "gelöscht" werden. Die Anzeige ist aber sowieso optional. Man kann sich auch merken, welche Taste zur jeweiligen Einstellung führt und den Bildschirm nicht verwenden.



Den Trödelfaktor im Programm integrieren:
Noch hat die Veränderung der Variable *prozent* (zu Anfang am Besten auf 0 stellen) keine Auswirkung.

Deshalb müssen wir nun noch etwas in dem *Unterprogramm "Beschleunigung"* verändern.

Anstatt bei erfüllter Bedingung (Maximalgeschwindigkeit nicht erreicht), sofort zu beschleunigen, bauen wir noch einen Test ein, sodass in 15% der Fälle die Geschwindigkeit nicht erhöht wird, sondern gleich bleibt. (Wenn der Trödelfaktor "aktiviert" wurde.)

Wir verwenden hier "Schreibe Variable" und "Wenn (Bedingung x, dann) mache (y) sonst (z)".

Und zwar wird getestet, dass *prozent* (entweder 0 oder 15) kleiner ist als eine Zahl zwischen 1 und 100, die jedes mal zufällig generiert wird.
Wenn das wahr ist, wird wie zuvor die Geschwindigkeit erhöht (*geschwindigkeit + 5*), wenn das falsch ist, wird die Geschwindigkeit beibehalten (*geschwindigkeit*).

Wenn der Trödelfaktor fehlt, ist *prozent* 0. Weil 0 immer kleiner ist als eine Zahl zwischen 1 und 100, steigt die Geschwindigkeit konstant.

Mit Trödelfaktor ist *prozent* aber 15, was dazu führt, dass mit einer 15% Wahrscheinlichkeit eine Schleife durchlaufen wird, ohne dass beschleunigt wird.

Der Trödelfaktor von 15% ist keine feste Vorgabe - man kann mit diesem Wert spielen, um das Trödeln mehr oder weniger regelmäßig und auffällig zu machen! Ein Trödelwert über 50% ist allerdings ein sicheres Rezept für Stau...

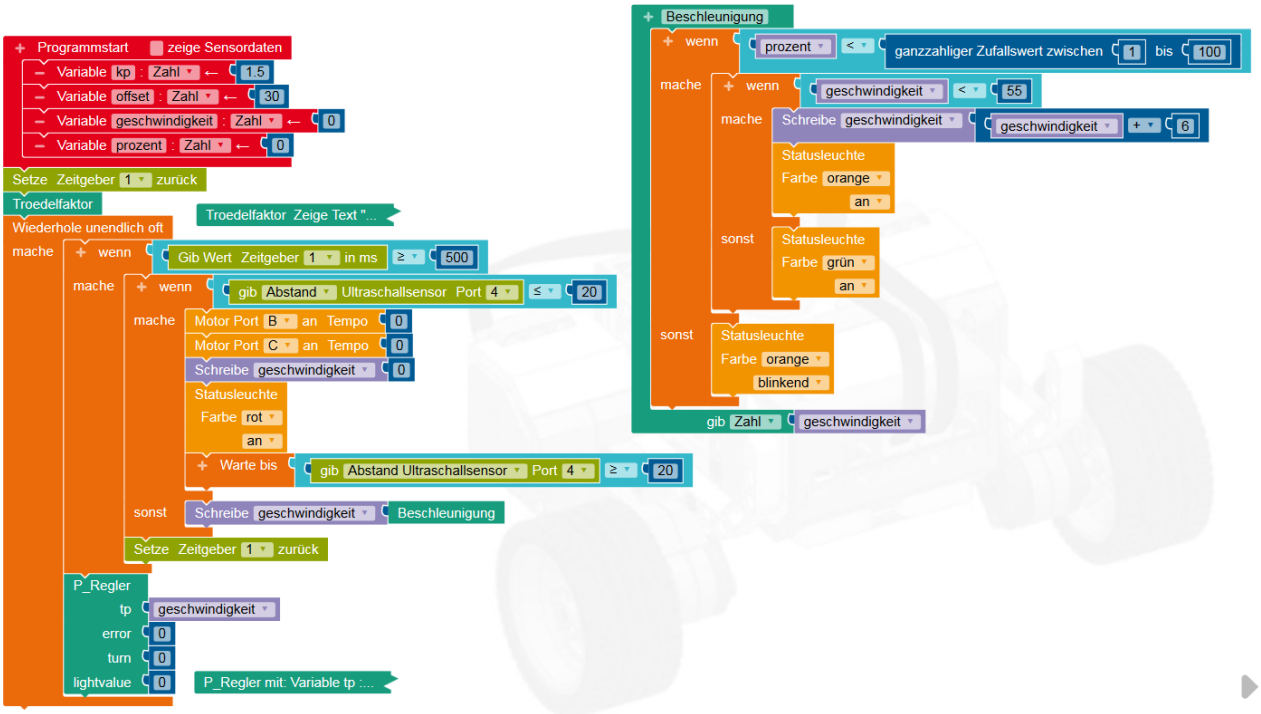
Blinkanzeige

Um ein Programm zu testen und den internen Verlauf bei der Ausführung sichtbar zu machen, ist es immer hilfreich, Elemente wie Leuchten, Töne oder Anzeigen auf dem Bildschirm einzubauen, wenn bestimmte Teile des Programms durchlaufen werden.

Zu diesem Zweck haben wir die Statusleuchte verwendet und an einigen Stellen im Programm eingebaut, sodass sie als Markierung für bestimmte Verhaltensmuster des Roboters dient.

Wir haben zum Beispiel den Roboter so programmiert, dass seine Statusleuchte beim Beschleunigen orange leuchtet und blinkt wenn er gerade trödelt. Außerdem leuchtet er grün, sobald er die Maximalgeschwindigkeit erreicht hat und rot bei Geschwindigkeit 0.

So sehen die Teile des Programmes aus, nachdem die Blinkanzeige eingebaut wurde:
(Bei den Unterprogrammen Trödelfaktor und P-Regler wird kein Statusleuchte-Block eingefügt, sie bleiben unverändert.)



Fraunhofer IAIS

Roberta® Lernen mit Robotern

Weitere Informationen Ansprechpartner

Weitere Informationen

Videos:

Open Roberta Stau-Simulation	https://www.youtube.com/watch?v=E1k7OcB9ouE
Vollbremser auf vollen Straßen - Open Roberta Simulation	https://www.youtube.com/watch?v=cn-BN29nChY

Material zur Stausimulation:

Programm zum Download:	http://roberta-home.de/de/node/825
Tutorial zur Stausimulation:	http://roberta-home.de/de/userarea/downloads - unter Tutorial: Open Roberta Stau-Simulation für EV3
Artikel zur Stauforschung:	http://www.zukunft-mobilitaet.net/3344/strassenverkehr/wie-entstehen-staus-phantomstau/
Tutorial für einen PID Controller:	http://www.inpharmix.com/jps/PID_Controller_For_Lego_Mindstorms_Robots.html

Ansprechpartner

Thorsten Leimbach	Thorsten.Leimbach@iais.fraunhofer.de
Beate Jost	Beate.Jost@iais.fraunhofer.de
Webseite	www.roberta-home.de