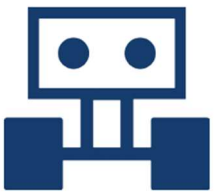
ICDL Robotik Basic

Handbuch Version 1.4



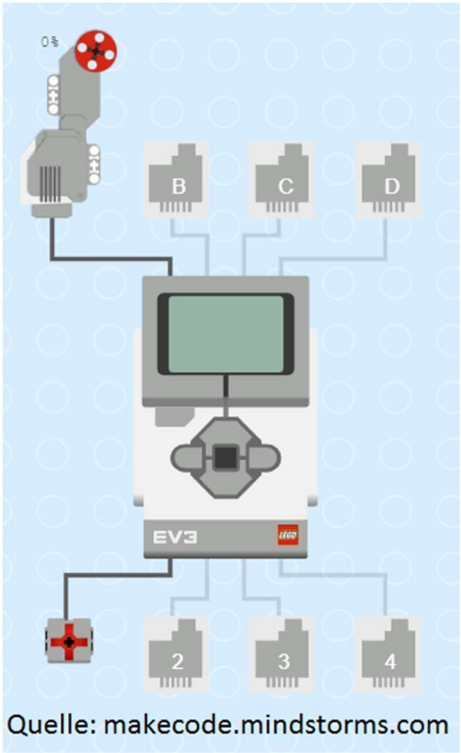
Inhalt

In diesem Dokument wird der Punkt (“.”) statt dem Komma (“,”) als Dezimaltrennzeichen verwendet.

Definition

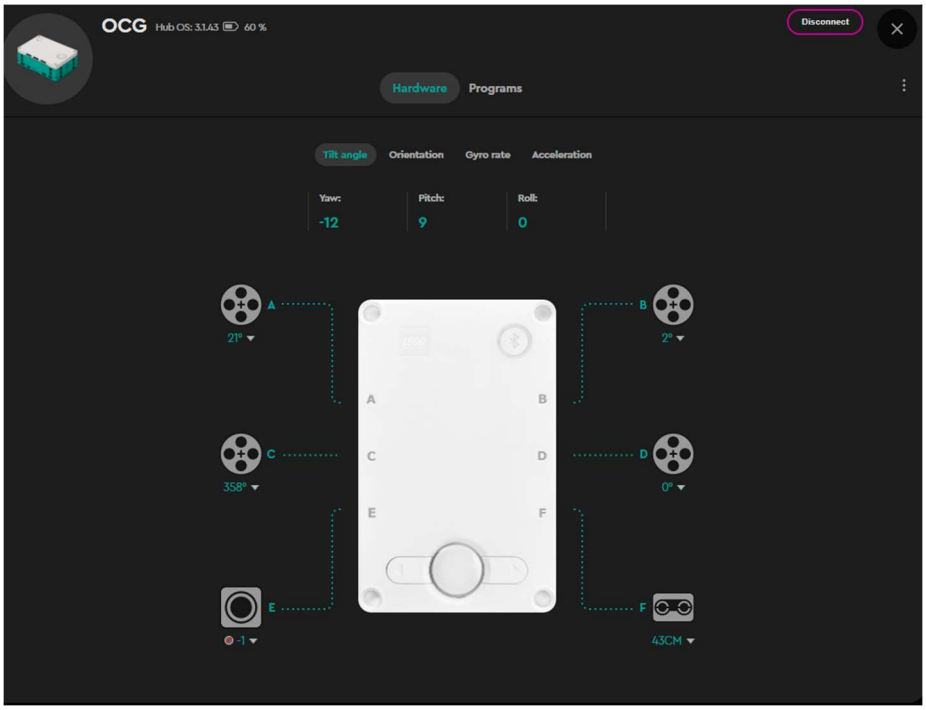
Ein Roboter ist eine technische Apparatur, die üblicherweise dazu dient, dem Menschen häufig wiederkehrende mechanische Arbeit abzunehmen. Roboter können sowohl ortsfeste als auch mobile Maschinen sein. Oft werden Roboter von Computerprogrammen gesteuert. Ein Computerprogramm ist eine den Regeln einer bestimmten Programmiersprache genügende Folge von Anweisungen.

Wichtiger Bestandteil eines modernen Roboters ist die zentrale Recheneinheit (= ein Computer) - man könnte auch sagen "das Gehirn" des Roboters. Diese zentrale Recheneinheit ist über Schnittstellen mit den Sensoren und Aktoren verbunden. Sie kann Daten sammeln und speichern, Berechnungen durchführen, Signale empfangen und Signale senden. Auch Schnittstellen nach außen sind üblicherweise vorhanden, mit denen der Roboter gesteuert werden kann oder über welche der Roboter mit anderen Systemen kommunizieren kann. Die Schnittstellen können kabelgebunden (zum Beispiel über USB) oder drahtlos (zum Beispiel über Bluetooth) verbunden werden. Das folgende Bild zeigt die zentrale Recheneinheit (Bildmitte), die Verbindungen zu den Aktoren (oben) und zu den Sensoren (unten). Links oben ist ein Motor als Aktor und links unten ein Tastsensor als Sensor eingezeichnet (entnommen aus MakeCode für Mindstorms).



Quelle: makecode.mindstorms.com

Das nachfolgende Bild zeigt die Recheneinheit und die Verbindungen zu Aktoren und Sensoren des Lego Mindstorms (Robot Inventor bzw. Spike Prime).



Quelle: Screenshot aus Lego Mindstorms IDE 10.3.1

Eine wichtige Aufgabe der Recheneinheit ist das Ausführen von Programmen, die das Verhalten des Roboters steuern.

Als Recheneinheit kommen in modernen Systemen häufig Mikrocontroller (wie zum Beispiel beim Arduino) oder Single-Board-Computer (SBC) (wie zum Beispiel Raspberry Pi) zum Einsatz. Mikrocontroller haben weniger Ressourcen und kein eigenes Betriebssystem, sie führen ein Programm aus, das man auf den Mikrocontroller überträgt. Single-Board-Computer führen aber ein Betriebssystem aus, was mehr Möglichkeiten bietet, aber auch mehr Ressourcen verbraucht. Roboter enthalten immer Aktoren (manchmal auch Aktuatoren genannt) zum Agieren innerhalb ihrer Umgebung. Aktoren sind Baueinheiten die ein Signal vom Steuerungscomputer in mechanische Bewegungen oder in eine Veränderung physikalischer Größen wie Druck (zum Beispiel durch eine Pumpe) oder Temperatur (zum Beispiel durch einen Heizstab) umsetzen.

Meist enthalten Roboter Sensoren zur Erfassung von Informationen über die Umwelt, aber auch über sich selbst, wie zum Beispiel der eigenen Achsenpositionen. Sensoren können Veränderungen physikalischer Größen wie Druck oder Temperatur wahrnehmen. In vielen Fällen sind externe Sensoren zur Bewältigung von Arbeiten notwendig. Zum Beispiel, wenn nicht vorhersehbare Umstände erkannt werden müssen und der Roboter darauf reagieren soll. Gewisse Aufgaben kann der Roboter aber auch ohne Sensoren erledigen.

Meist sind alle Bauteile auf einem mechanischen Gestell montiert, inklusive eventueller Getriebe.

Einige der Bestandteile des Roboters benötigen eine Energieversorgung (Batterien, Akkus, Solarzellen, …), üblicherweise sind das die Recheneinheit, die Aktoren und die Sensoren. Manche Sensoren können aber auch ohne Energiezufuhr messen (passive Sensorik).

Weil die moderne Robotik zu einem großen Teil auf elektronischen Bauteilen beruht, wird die Energie meist in Form von Strom bereitgestellt. Als Stromquellen werden häufig Batterien oder Akkus eingesetzt, manchmal auch zusätzlich Solarzellen.

Ein Roboter muss nicht unbedingt vollständig autonom handeln können. Darum unterscheidet man autonome und ferngesteuerte Roboter.

Einen Roboter, dessen Konstruktion der menschlichen Gestalt nachempfunden ist, nennt man humanoiden Roboter oder auch Androiden.

(Definition abgewandelt von https://de.wikipedia.org/wiki/Roboter)

Roboterdefinitionen sind nicht immer eindeutig. Manche Systeme können eindeutig als Roboter identifiziert werden, bei anderen Systemen ist dies aber schwieriger und es kommt dann auf die jeweilige Roboterdefinition an, die man verwendet.

Verwendung von Robotern

- **Erkundungsroboter**
  wie zum Beispiel die Mars-Rover
- **Industrieroboter**
  zur Handhabung, Montage oder Bearbeitung von Werkstücken
- **Medizinroboter**
  werden unter anderem bei medizinischen Eingriffen eingesetzt
- **Serviceroboter**
  erbringen einfache Dienstleistungen zum Beispiel in der Gastronomie
- **Personal Robot**
  Serviceroboter für den privaten Gebrauch, zuhause oder unterwegs.
- **Bildungs- oder Lernroboter (Educational Robotics)**
  werden als günstige Standardplattformen auch in Forschung, Lehre und Schule eingesetzt
- **Transportroboter**
  befördern Gegenstände und Lasten aller Art

Autonomes Fahren

Auch autonomes Fahren ist ein Teilgebiet der Robotik. Ziel ist es, Fahrzeuge mit ausreichenden Sensoren, Künstlicher Intelligenz und benötigten Aktoren auszustatten, sodass ein autonomer Betrieb eventuell auch ganz ohne Person am Steuer durchgeführt werden kann. Auf ausgewählten Strecken, zum Beispiel als Flughafenshuttle, klappt das schon recht gut, aber ein allgemeiner Betrieb auf beliebigen Strecken, der keinen menschlichen Eingriff erfordert, funktioniert noch nicht. Eine große Herausforderung stellen momentan noch stark wechselnde Bedingungen und dynamische Situationen dar, zum Beispiel Stadtverkehr mit Personen, die zu Fuß oder mit dem Fahrrad unterwegs sind, anderen Autos, Ampeln, … . Ein großes Problem sind vor allem auch unterschiedliche Sichtbedingungen durch unterschiedliche Wetterbedingungen und unterschiedliche Tageszeiten.

Bestandteile eines Roboters

Wesentliche Teile und Komponenten des Roboters umfassen:

- Zentrale Recheneinheit

- Sensoren
- Aktoren
    - Motoren
    - Antrieb: Räder, Ketten, Beine, Rotoren, …
    - Ausgabe: Bildschirm, Lautsprecher, …
    - Sonstige, wie zum Beispiel Heizstab, ...
- Kabel
- Gestell (Chassis), Außenhaut
- Energieversorgung: Akku, Batterie, Solarzelle, …
- Zubehör: Ersatzteile, Werkzeug für Zusammenbau, Wartung, Reparatur, ...

Sensoren

Sensoren können Änderungen in der Umgebung wahrnehmen, wie: Lichtstärke, Entfernung, Winkel, Temperatur, ...

Kenntnis der physikalischen Funktionsweise des jeweiligen Sensors lässt meist Rückschlüsse auf die Beschränkungen des Sensors zu.

- Entfernungssensor
    - Ultraschall:
      Der Sensor sendet ein Klickgeräusch und empfängt das Echo dieses Signals. Aus der Laufzeit errechnet er den Abstand.
      Steht der Roboter schräg zu einem Hindernis, ist das Hindernis zu weit entfernt oder sehr schallabsorbierend, dann scheitert die Messung.
    - Infrarot:
      Aus dem Winkel eines reflektierten Infrarotlichtstrahls wird die Entfernung berechnet.
    - Laser:
      Die Messung erfolgt über einen Laserstrahl. Die Messung kann scheitern, wenn der Laserstrahl zum Beispiel auf eine ungeeignete (zu stark absorbierende oder streuende) Fläche fällt.
- Helligkeitssensor
    - Messung des reflektierten Lichts am Punkt auf den der Sensor gerichtet ist.
      In diesem Modus erfordert der Helligkeitssensor einen Maximalabstand vom gemessenen Objekt und eventuell einen Minimalabstand.
    - Messung der allgemeinen Umgebungshelligkeit (Ambient Light).
- Farbsensor

  Messung der Farbe des Punkts auf den der Sensor gerichtet ist.

  Auch hier ist unter Umständen wieder der Maximal- und Minimalabstand einzuhalten.

- Schallsensor

  Der Sensor misst die Schallintensität in der unmittelbaren Umgebung des Roboters.

- Temperatursensor
- Gyrosensor

Der Sensor misst die Drehbewegung und Orientierungsänderungen des Roboters. Es kann somit gemessen werden, wie weit sich der Roboter gedreht hat. Der Gyrosensor kann aber auch verwendet werden, um in eine bestimmte Himmelsrichtung zu fahren.

- GPS-Sensor (Global Positioning System)

  Liefert aktuelle Positions- und Höhenkoordinaten des Roboters. GPS-Sensoren funktionieren in geschlossenen Räumen in der Regel nicht.

- Kamera

  Die Qualität der Kamera aber auch die verwendeten Bilderkennungsalgorithmen bestimmen, welche Aufgaben die Kamera übernehmen kann: Objekterkennung, Gesichtserkennung, Personenerkennung, Navigation, ...

- 3D-Sensor

  Autonome Fahrzeuge setzen oft 3D-Scanner ein. LIDAR (Light Detection and Ranging) ist ein Vertreter aus dieser Kategorie. Ein Laser wird verwendet, um Objekte zu erkennen beziehungsweise zu kategorisieren. Damit ist es möglich die Umgebung des Fahrzeugs in 3D zu erfassen.

- Tastsensor

  Einerseits können Tastsensoren eingesetzt werden, um zu erkennen, wann der Roboter gegen einen festen Gegenstand stößt. Tastsensoren können auch als Eingabegeräte verwendet werden, zum Beispiel zum Starten, Pausieren oder Beenden eines Programms.

Aktoren

Aktoren wandeln ihre (in vielen Fällen elektrische) Antriebsenergie in mechanische Energie oder andere Energieformen um und bewirken eine Änderung in ihrer Umgebung, ihres eigenen Zustandes, ihrer Position, Lage und Verdrehung. Aktoren sind bei mobilen Robotern für die Fortbewegung verantwortlich.

Motoren

In der Robotik ist es sehr wichtig, dass man Motoren einen definierten Winkel drehen lassen kann. Zum Beispiel eine ganze Drehung (360°) oder eine Viertel-Drehung (90°). Das erreicht man, indem man entweder Motoren mit einem eingebauten Drehgeber (Rotary Encoder) oder Schrittmotoren verwendet.

Schrittmotoren werden mit Impulsen angesteuert und drehen bei jedem empfangenen Impuls um einen genau definierten kleinen Winkel weiter.

Motoren können auch gleichzeitig über Sensoren verfügen, die den aktuellen Verdrehungswinkel des Motors auslesen können.

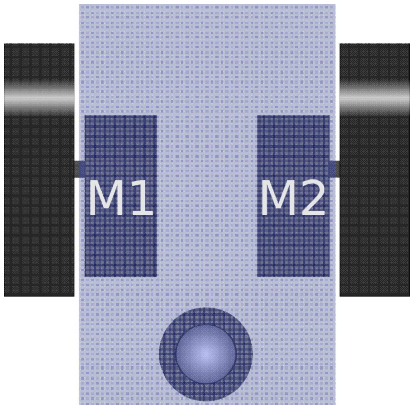
Weitere Aktoren:

- Antriebsräder, Antriebsketten

- Greifer
- Lampen, LEDs
- Bildschirm (Bildschirme sind für Fehlersuche (Programm-Debugging) sehr nützlich)
- Lautsprecher: akustische Signale, Töne, Sprachausgabe

Differentialantrieb

In der Robotik findet man häufig autonome Fahrzeuge folgender Bauart, siehe folgende Abbildung.

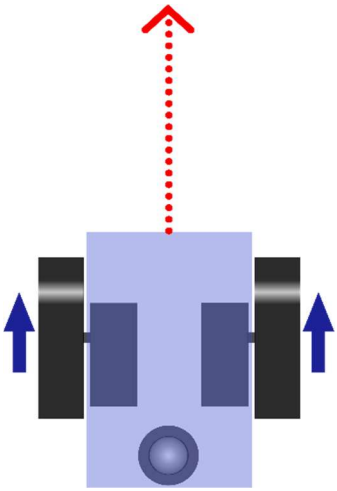


Im obigen Bild, mit dem Blick von oben auf das Fahrzeug, sind zwei Antriebsräder und unten eine passive, nicht angetriebene Kugelrolle mit Halterung zu sehen. Die Motoren M1 und M2 treiben die Räder an. Das Fahrgestell ist blau durchscheinend dargestellt.

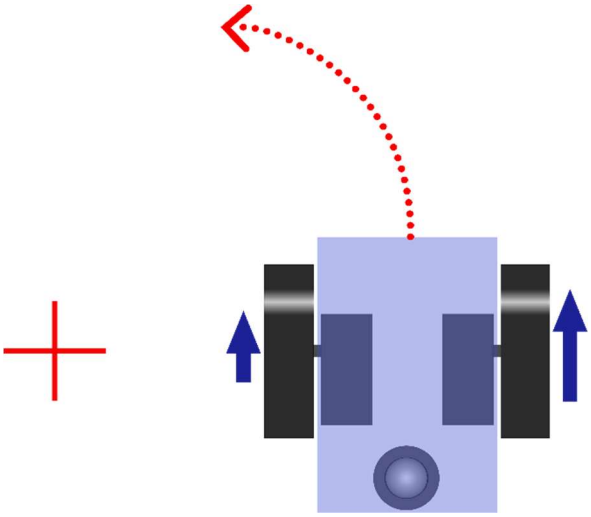
Statt der Kugelrolle könnte hier auch eine Lenkrolle (auch Castor-Rolle oder Castor-Rad genannt) zum Einsatz kommen.

Bei dieser Bauart gibt es keine Lenkung so wie bei einem traditionellen Auto oder einem Fahrrad, denn die Räder können nicht verdreht werden. Stattdessen werden beide Antriebsräder von je einem eigenen Motor einzeln angesteuert. Das Prinzip ist ganz einfach:

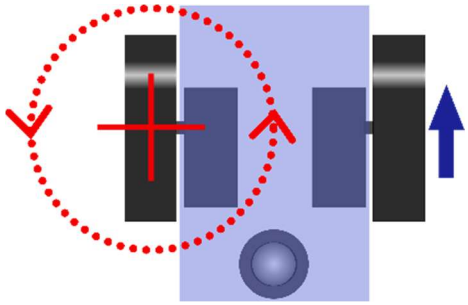
1) Werden beide Motoren mit gleicher Umdrehungszahl angesteuert (wie durch die beiden blauen Pfeile angedeutet) dann fährt das Fahrzeug geradeaus.



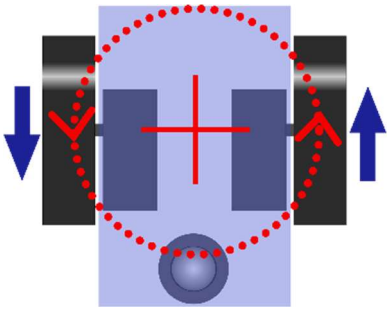
2) Werden beide Räder mit verschiedener Umdrehungszahl angesteuert, dann fährt das Fahrzeug einen Bogen.



3) Wird nur ein Rad angesteuert und das andere bleibt still, dann dreht sich das Fahrzeug. Drehpunkt ist dabei das stillstehende Rad.



4) Drehen sich beide Räder mit gleicher Leistung, aber gegenläufig, dann dreht sich das Fahrzeug am Stand.



Zusammenbau

Manche Roboter können fertig zusammengebaut erworben werden, manche gibt es als Bausatz. Im Bereich der Educational Robotics sind Baukästen sehr beliebt, aus denen verschiedene Modelle gebaut werden können. Die Verbindungskabel sind dann meist als Steckverbindungen ausgeführt, die in Ports (Steckplätze) gesteckt werden müssen. Die Ports werden dann entweder mit Nummern oder Buchstaben beschriftet. Im Programm werden die Sensoren oder Aktoren dann mithilfe dieser Nummern oder Buchstaben angesprochen. Zum Beispiel schaltet ein Befehl den Motor auf Port A an oder liest die Entfernung vom Abstandssensor auf Port 1.

⚠Achtung

Das ist eine häufige Fehlerquelle, denn ist der Port im Programm falsch eingetragen, funktioniert das Programm nicht und meist gibt es auch keine Fehlermeldung, die auf den Fehler aufmerksam macht. Ein Aktor, der auf einem falschen Port angesprochen wird, reagiert nicht. Ein Sensor, der auf einem falschen Port ausgelesen wird, liefert keinen korrekten Wert.

Gegenstrategien

1) Kommen bei einem Workshop mehrere Baukästen zum Einsatz, ist aus diesem Grund darauf zu achten, dass alle Verbindungen gleich gesteckt sind, also dass immer die gleichen Portnummern mit den gleichen Sensoren und Aktoren verbunden sind. Sonst kann es passieren, dass ein Programm auf einem Gerät funktioniert und auf einem anderen nicht.
2) Aus dem Grund ist es sehr praktisch, wenn der Roboter über einen Ausgabebildschirm verfügt, denn dann können Sensorwerte dort ausgegeben werden und es fällt sofort auf, wenn Werte fehlerhaft sind. Bei einem Simulator kann es statt einem Ausgabeschirm auch ein Textfenster (Konsole) sein.

Auch bei den gängigen Simulatoren werden die Portnummern so verwendet wie bei den Baukästen, aber möglicherweise werden die Ports automatisch richtig vom Simulator befüllt oder es gibt eine Einstellung „**`auto`**“ die automatisch richtig funktioniert. Das ist möglich, da dem Simulator intern die Verbindungsstruktur bekannt ist.

Ausnahme ist, wenn zwei oder mehr Sensoren oder Aktoren vom gleichen Typ verwendet werden. Dann muss der Port manuell angegeben werden. Wenn zum Beispiel mehrere Abstandssensoren verwendet werden, dann kann der Simulator nicht automatisch ermitteln welcher der beiden Sensoren gemeint ist.

GearsBot

GearsBot ist ein kostenloser und quelloffener Robotik-Simulator. GearsBot ist webbasiert (läuft im Browser). Selbstinstallation auf einem eigenen Webserver ist möglich.

Wichtige Links

Simulator: https://gears.aposteriori.com.sg/

GitHub: https://github.com/QuirkyCort/gears/

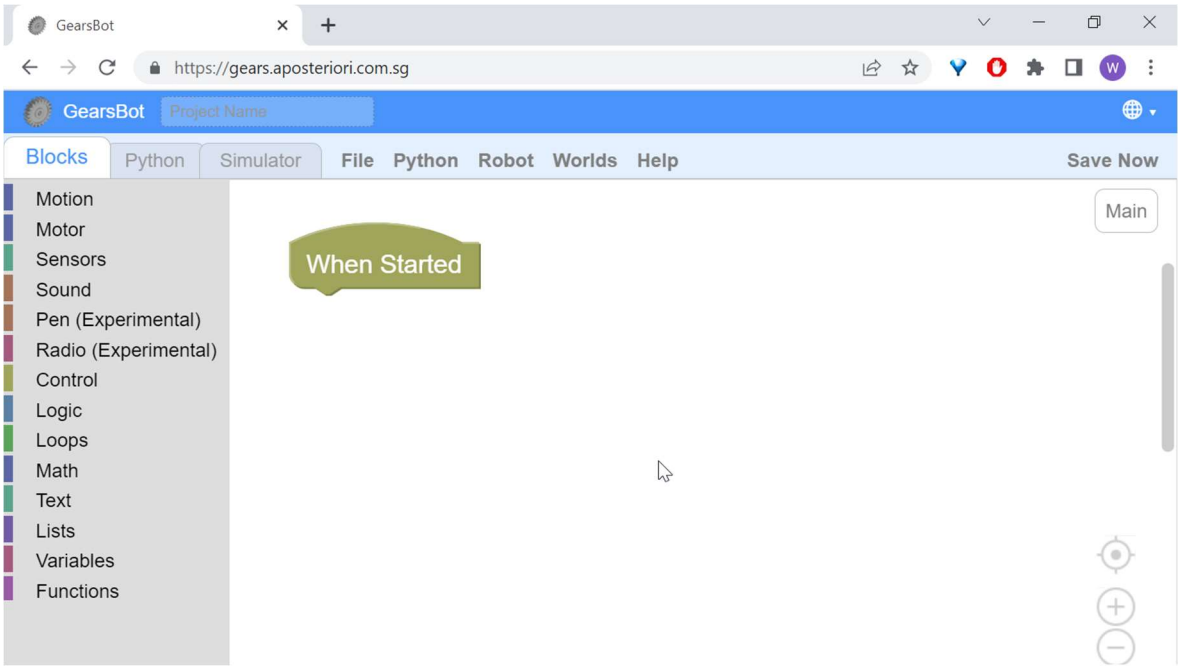
Wiki: https://github.com/QuirkyCort/gears/wiki

YouTube

A Posteriori-Kanal (von der entwickelnden Organisation):
https://www.youtube.com/channel/UCPMqSjdcAAkUa-qNf6m79Kg/videos

Yoni Garbourg (einer der Entwickler):
https://www.youtube.com/channel/UCbkSK_vRxYSoAxIUPB4tlaw/videos

Oberfläche



Ganz rechts beim Weltkugelsymbol im blauen Balken kann man die Sprache umschalten. Hier sollte man eher bei Englisch bleiben, als Vorbereitung auf das Programmieren mit textbasierten Sprachen, wo so eine Umschaltung nicht existiert. Von links können Blöcke aus farblich sortierten und nach Themen zusammengefassten Kategorien nach rechts gezogen werden. Alle Blöcke, die an den „**`When Started`**“-Block angefügt werden, sind aktiv. Blöcke, die nicht am Startblock hängen, werden nicht ausgeführt.

Blockbasierte Programmiersprachen

In blockbasierten (visuelle) Programmiersprachen erstellt man Programme, indem man Blöcke aus dem Blockbereich (linke Spalte in der obigen Abbildung) in den Editorbereich (rechter Bereich in der obigen Abbildung) zieht und aneinanderfügt. Das Prinzip ist, dass Bausteine nur dort eingefügt werden können, wo sie auch hingehören, zum Beispiel Zahlen nur in Zahlenfelder, Bedingungen nur in Bedingungsfelder und so weiter. Dadurch ist es fast unmöglich den Regeln der Programmiersprache nicht entsprechende (syntaktisch falsche) Programme zu bauen. Das ist ein großer Vorteil im Vergleich zu textbasierten Sprachen. Blockbasierte Programmiersprachen sind deshalb der ideale Einstieg in die

Programmierung. Die gesammelte Erfahrung kann später in der textuellen Programmierung angewendet werden.

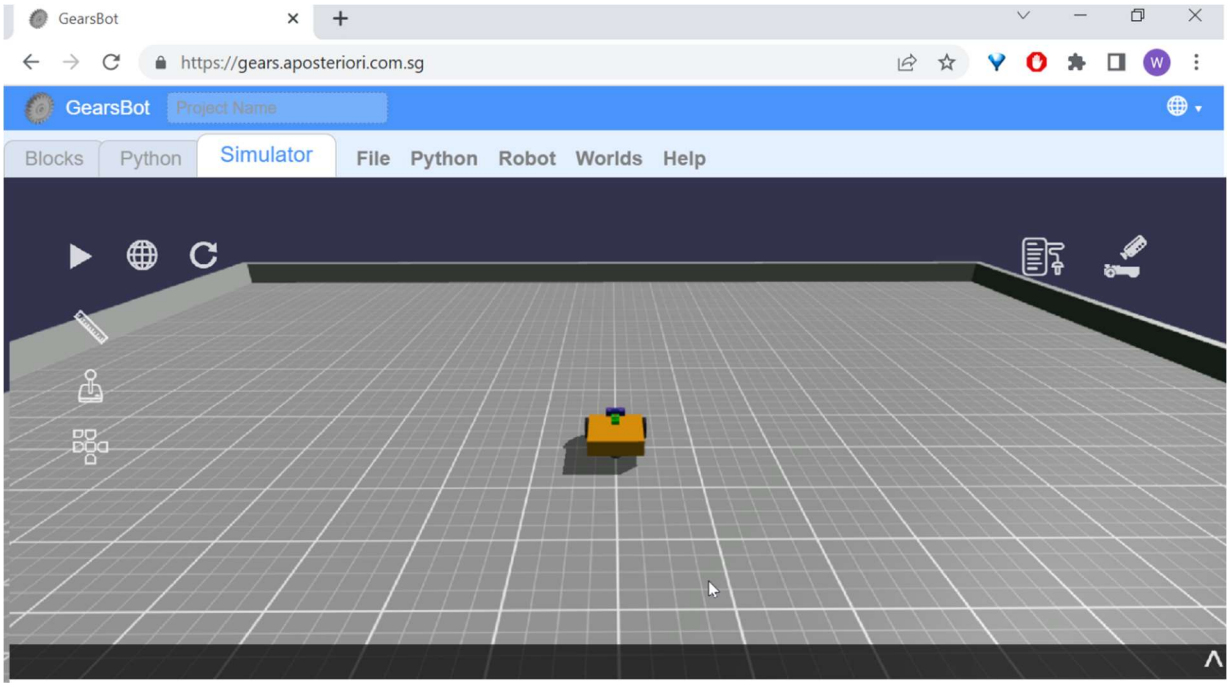
Ereignisbasierte Programmierung

Oft wird bei der Steuerung von Robotern ereignisbasierte Programmierung eingesetzt. Entsprechende Blöcke beginnen mit “When ...” oder “On ...”. Bei GearsBot trifft das zum gegenwärtigen Zeitpunkt nur auf den Startblock zu. Der Programmfluss springt zu diesen Blöcken, wenn das dazugehörige Ereignis auftritt.

Simulator

Vorbemerkung

Autonome Fahrzeuge sind nur ein sehr kleines Teilgebiet der Robotik, aber hier lassen sich anschauliche und interessante, aber auch anspruchsvolle Aufgaben erstellen. Die bei Fahrzeugen erforderliche Navigation sei hier als Beispiel genannt.

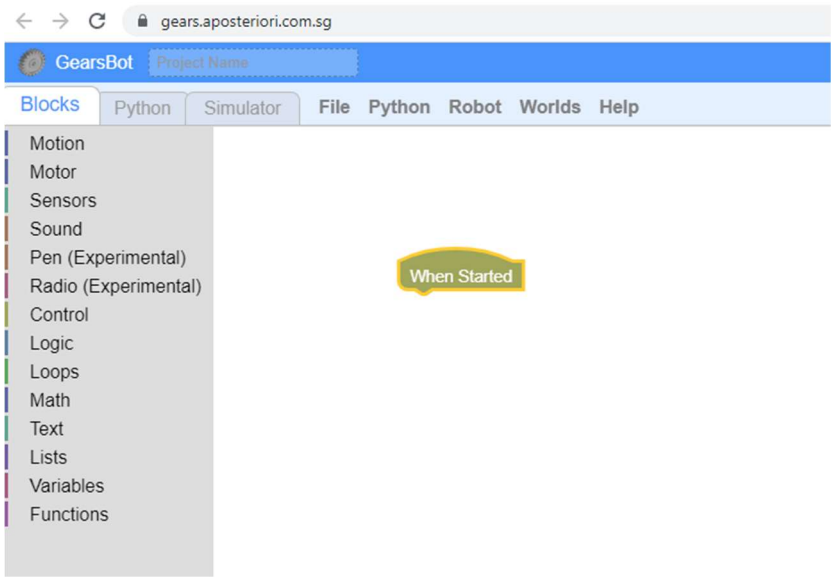


Die Standardwelt ist ein quadratisches Raster, das von einer Mauer umgeben ist. Mit der Play-Taste (▶) kann das vorher erstellte Skript gestartet werden. Mit der Reset-Taste (↻) wird das Programm gestoppt und das Fahrzeug auf den Ausgangspunkt zurückgesetzt. Ganz unten befindet sich eine erweiterbare Ausgabezeile, wo Textausgaben des Skripts angezeigt werden. Sowohl die virtuelle Welt als auch der Roboter sind in eigenen Menüpunkten konfigurierbar. Im Menüpunkt „File“ kann das aktuelle Skript auf dem lokalen Computer gespeichert oder ein vorher gespeichertes Skript geladen

werden. Vor dem Speichern empfiehlt es sich im Feld links oben einen Projektnamen zu vergeben. Andernfalls kann man die Datei aber auch nach dem Speichern umbenennen.

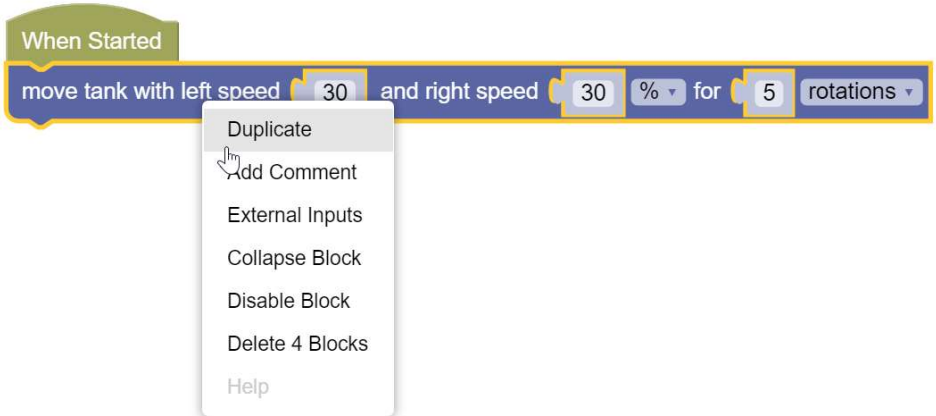
Das erste Programm

Ziehen wir zuerst einen **`move-tank`**-Block (Kategorie Motion im linken Bereich) in den Blockeditor.

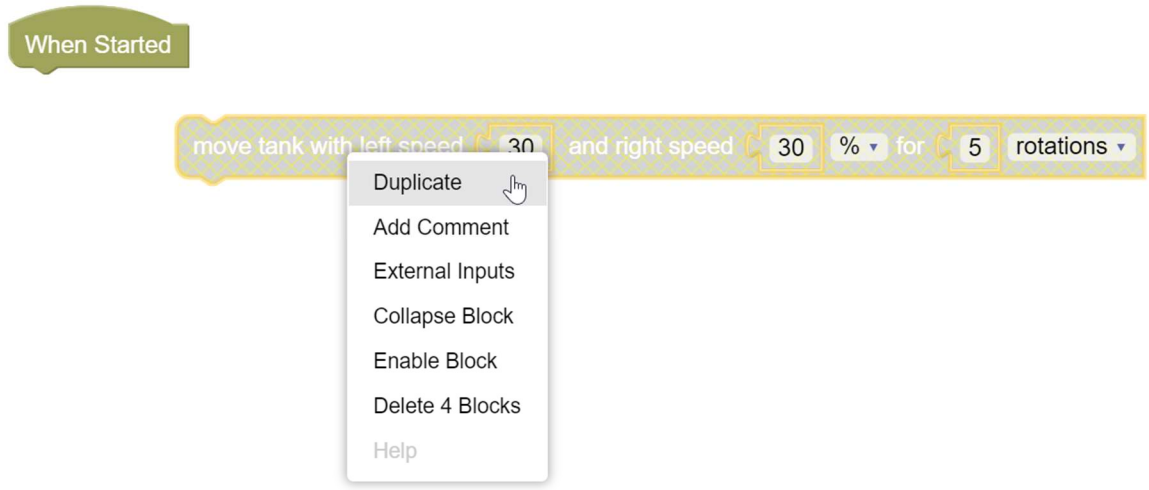


Solange der Block nicht am Startblock angeheftet ist, wird er durscheinend dargestellt. Damit wird angedeutet, dass der Block nicht aktiv ist und beim Start nicht ausgeführt wird.

Heften wir den Block an den Startblock, dann wird er aktiv und farbig dargestellt. Befüllen wir ihn dann wie unten zu sehen mit 30/30/5. Mit der Tabulatortaste (↹) kann man zum jeweils nächsten Feld springen. Klicken wir den Block dann mit der **rechten** Maustaste an. Dann öffnet sich ein Kontextmenü und wir wählen „**`Duplicate`**“ (Duplizieren).

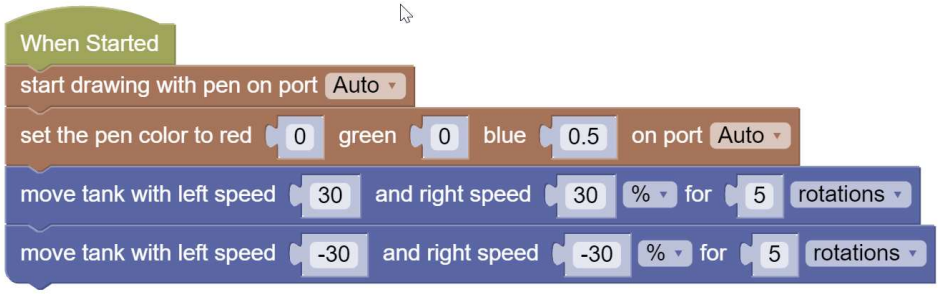


Das Duplizieren funktioniert auch vor dem Anheften:



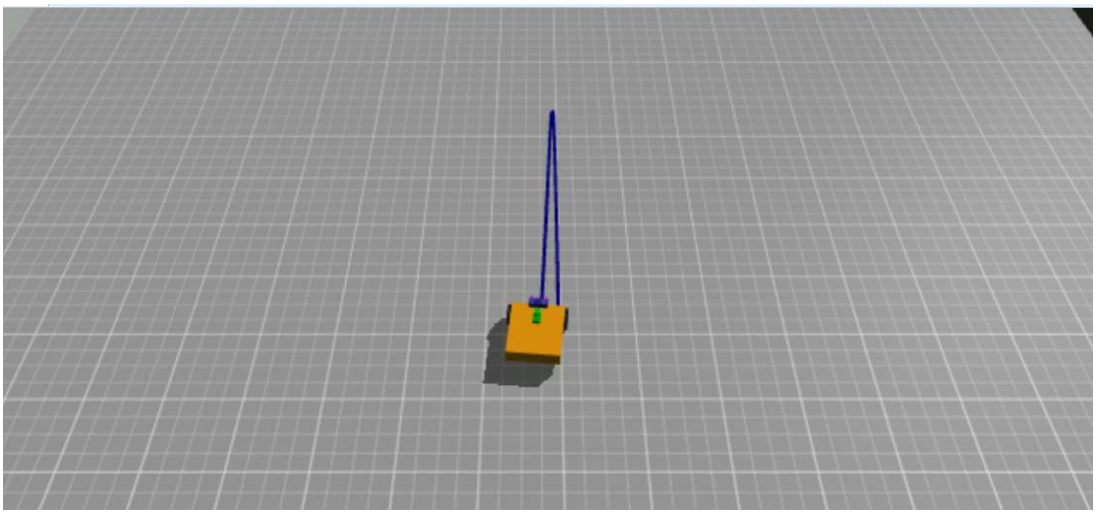
Den duplizierten Block heften wir an und befüllen ihn wie unten. Alternativ können wir auch einfach einen zweiten Block hereinziehen, aber duplizieren ist vor allem praktisch, wenn eine Reihe von Blöcken ähnlich konfiguriert werden soll. Dann nimmt man diese Einstellungen vor dem Duplizieren vor.

Zuletzt fügen wir noch aus der Kategorie Pen (Stift) zwei Blöcke wie unten gezeigt hinzu.



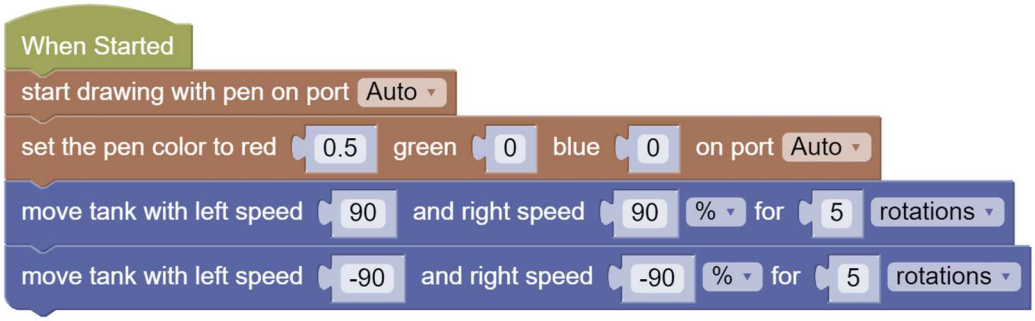
Denn wechseln wir zum Simulator und starten das Programm.

Ergebnis

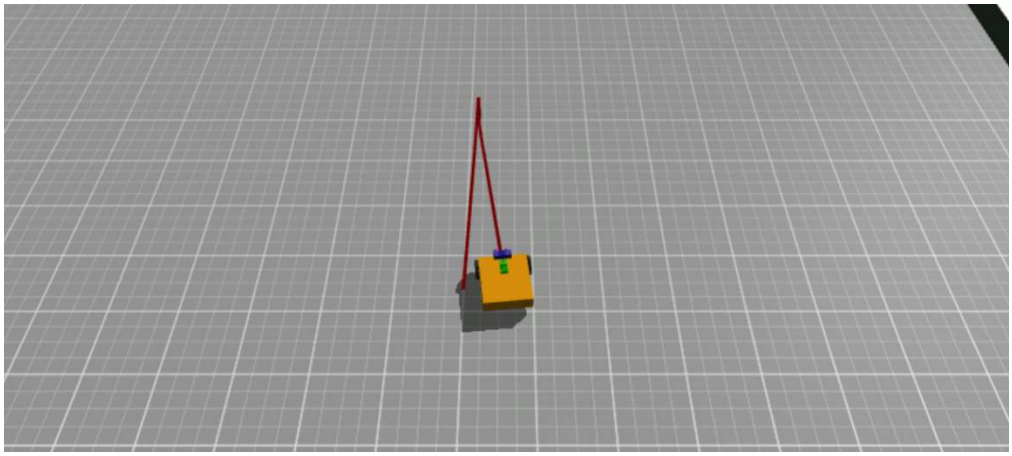


Der Roboter fährt 5 Radumdrehungen nach vor und dann ohne Unterbrechung wieder 5 Radumdrehungen zurück. Er landet dann ungefähr wieder an derselben Stelle.

Wir wandeln das Programm jetzt etwas ab. Die 30 ersetzen wir immer mit 90 und beim Stift wandern die 0.5 (in diesem Dokument wird der Punkt (“.”) statt dem Komma (“,”) als Dezimaltrennzeichen verwendet) von Blau nach Rot.



Dann starten wir wieder den Simulator.



Jetzt ist die Abweichung größer als bei niedrigerer Geschwindigkeit. Aber das Ergebnis ist nicht immer gleich, wenn wir den Versuch mehrmals wiederholen.

Bereits jetzt können wir einige interessante Beobachtungen machen:

- Im Gegensatz zum klassischen Programmieren führt mehrmaliger Programmaufruf desselben Programms auch vom gleichen Ausgangspunkt nicht immer zum gleichen Ergebnis. GearsBot simuliert die physikalischen Kräfte, die auf das Fahrzeug wirken, inklusive der in der Realität immer auftretenden kleinen Schwankungen, zum Beispiel verursacht durch verschiedene Bodenbeschaffenheiten.
- Schaltet der Motor aus, dann wird das Fahrzeug nicht abrupt zum Stehen kommen, sondern die Massenträgheit des Fahrzeugs wird dazu führen, dass das Fahrzeug sich noch etwas weiterbewegt. Das gilt auch bei Drehungen.
- Alle Bewegungen werden durch Reibung gebremst. Auch die Reibung ist Schwankungen durch Unterschiede in den Materialien oder kleinen Verunreinigungen unterworfen.

Diese Faktoren verstärken sich bei höherer Geschwindigkeit des Fahrzeugs. Man kann sich auch vorstellen, dass diese kleinen Abweichungen sich bei längeren Programmen zu großen Abweichungen summieren können.

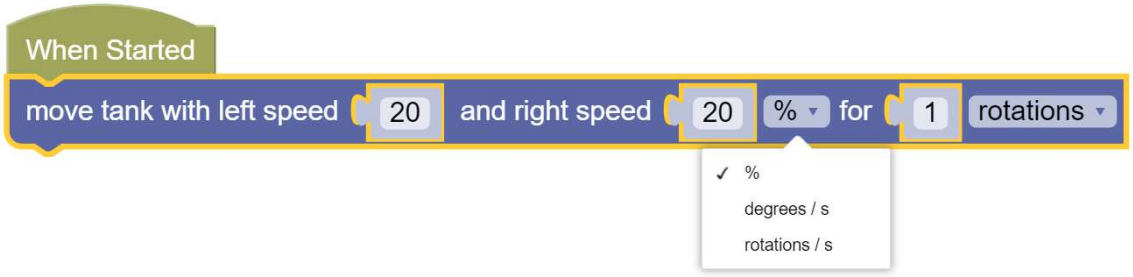
Ein wichtiger Bestandteil der Roboterprogrammierung ist es, diese Ungenauigkeiten und Abweichungen möglichst klein zu halten oder diese entsprechend zu korrigieren.

Langsamere Geschwindigkeiten haben den Nachteil, dass in gleichen Zeitabständen weniger Weg zurückgelegt wird. Dasselbe gilt auch für Winkelgeschwindigkeiten bei einer Drehung. Ist es wichtig, dass Aufgaben schnell erledigt werden, muss ein guter Kompromiss gefunden werden. Später werden wir sehen, wie mithilfe von Korrekturverfahren die Geschwindigkeit wieder erhöht werden kann, ohne ungewünschte Abweichungen zu vergrößern.

Alle Blöcke, die wir bis jetzt verwendet haben, repräsentieren Anweisungen (=Befehle). Einfache Programme bestehen nur aus Anweisungsfolgen (Listen von Anweisungen), die streng chronologisch, der Reihenfolge nach abgearbeitet werden. Nach Ausführung der letzten Anweisung wird das Programm beendet. Programme, die nur aus Anweisungsfolgen bestehen, können schon interessante Aufgaben bewältigen, aber sie können nicht auf äußere Einflüsse oder innere Zustände reagieren.

Weiteres zum move-tank-Block

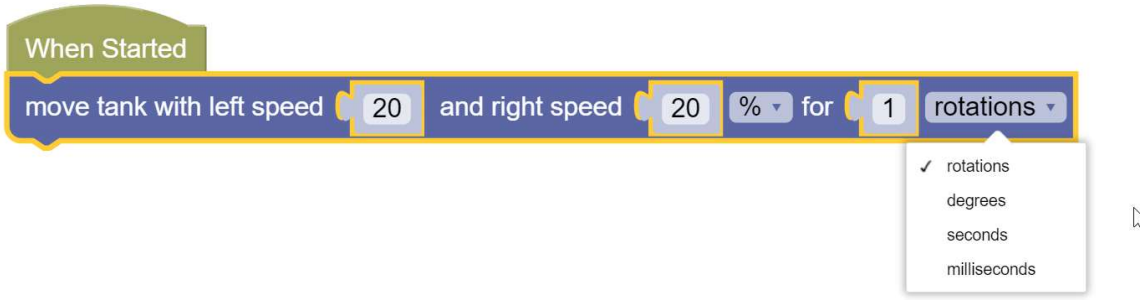
Den **`move-steering`**-Block wollen wir zunächst beiseitelassen. Den werden wir später genauer behandeln.



Die Motoren können mit %-Leistung, Grad pro Sekunde, Umdrehungen pro Sekunde angesteuert werden. Wir werden mit % auskommen aber je nach Situation können die anderen Optionen fallweise geeigneter sein. Eine höhere Leistung erhöht die Motordrehzahl, dadurch erhöht sich die Geschwindigkeit.

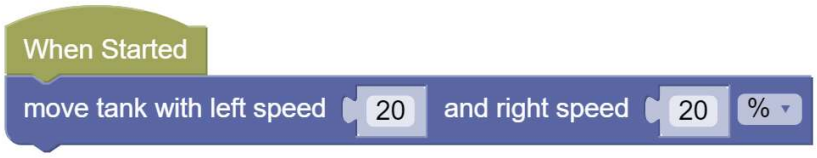
⚠Achtung

Ob ein Motor unter Last (zum Beispiel, wenn der Roboter eine Rampe hinauffährt) langsamer dreht, ist von der Bauart des Motors abhängig. Bei manchen Motoren sinkt die Drehzahl unter Last, bei manchen bleibt sie konstant. In der Robotik kommen eher letztere zum Einsatz. Steigt die Last wegen Bergauffahrt oder durch höhere Geschwindigkeit, dann steigt jedenfalls der Energieverbrauch.



Hier gilt wie oben: Wir werden „**`rotations`**“ verwenden aber in speziellen Fällen könnten die anderen Optionen praktischer sein.

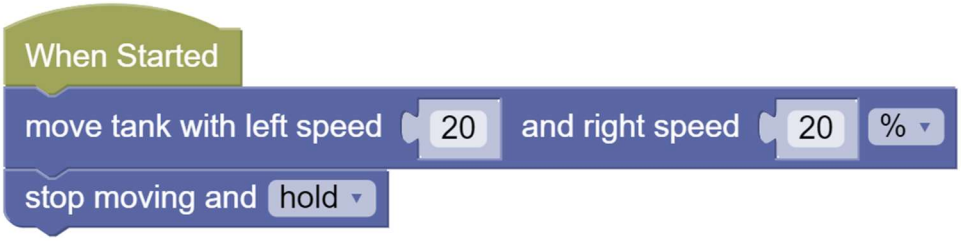
Aber warum gibt es den Baustein einmal in der Variante ohne rotations-Option?



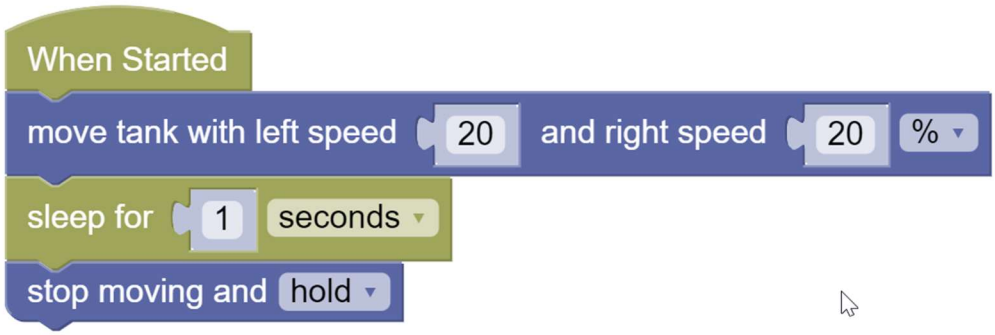
Wie lange dreht dann der Motor?

Die Antwort lautet: bis der nächste Motor-Block aufgerufen wird oder bis der **`stop-moving`**-Block (Kategorie Motion) aufgerufen wird. Speziell in GearsBot laufen die Motoren sogar nach Beendigung des Programms weiter, aber andere Simulatoren oder Bausätze können anders reagieren.

Folgendes Programm wird nicht wie erwartet funktionieren. Der **`move-tank`**-Block wird zwar aufgerufen, aber das Fahrzeug bewegt sich nicht, weil unmittelbar danach der **`stop-moving`**-Block kommt. Der **`move-tank`**-Block hat keine Zeit zu wirken.



Folgendes Programm bringt das Fahrzeug zum Fahren. Durch den **`sleep`**-Block (Kategorie Controls) hat der **`move-tank`**-Block 1 Sekunde Zeit zu wirken.

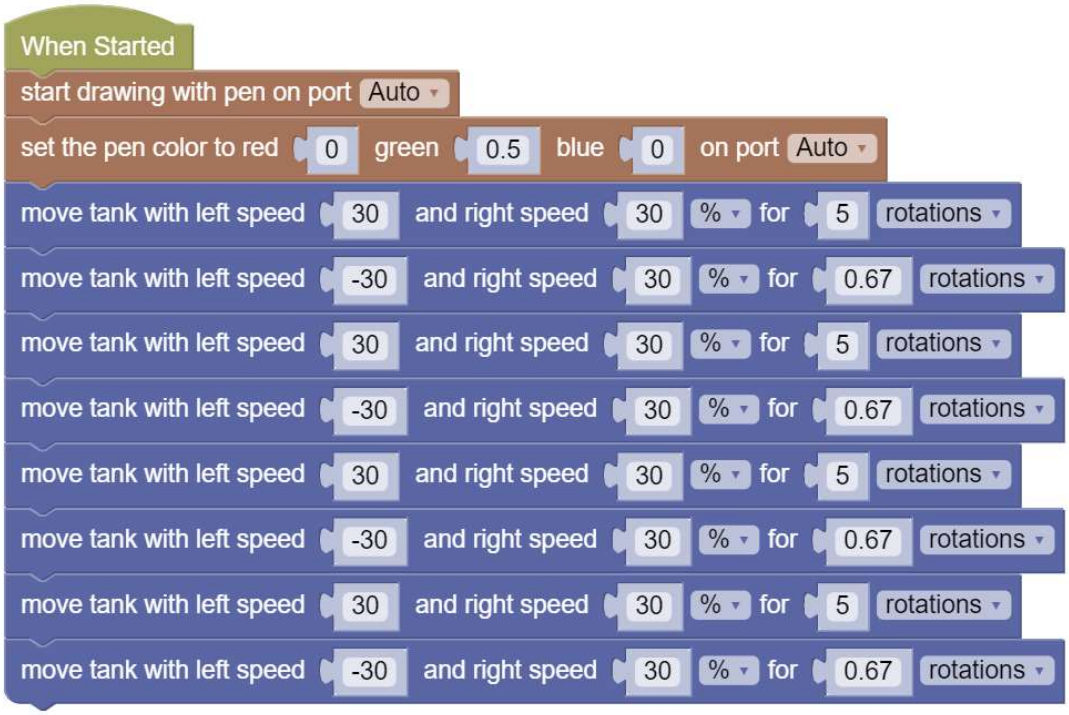


⚠Achtung

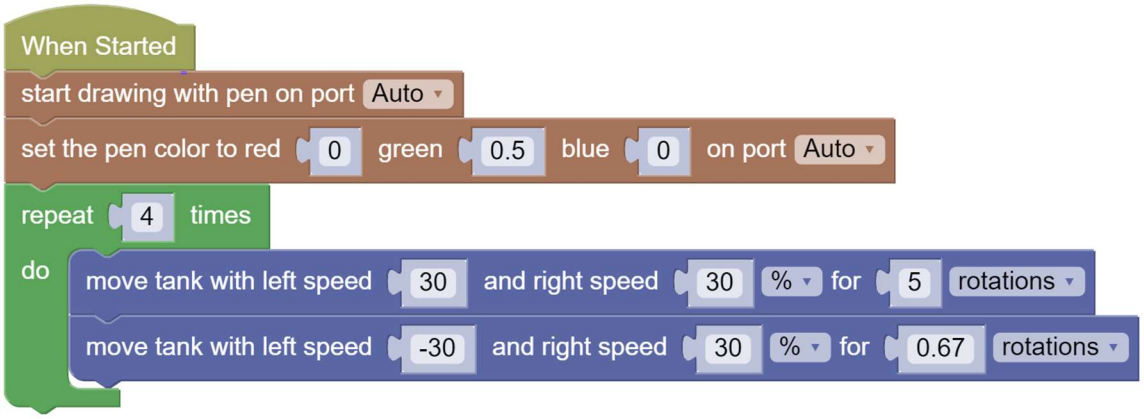
Das ist nicht ganz intuitiv. Üblicherweise, wenn wir Anweisungen haben und diese, wie oben erwähnt, streng chronologisch abgearbeitet werden, dann sind wir es gewohnt, dass die nächste Anweisung erst ausgeführt wird, wenn die Anweisung davor beendet ist. Streng genommen ist das auch hier der Fall, mit Ausnahme der Wirkung auf den Motor. Der dreht nämlich weiter, parallel zur Abarbeitung der folgenden Anweisungen. Später werden wir sehen, dass manche Programme nur dann verstanden werden können, wenn man diesen subtilen Unterschied bedenkt.

Ein Quadrat abfahren

Betrachten wir nun folgendes Programm. Einer Geradeausfahrt für 5 Umdrehungen folgt eine Drehung am Stand für 0.67 Umdrehungen. Das Ganze wird vier Mal wiederholt.



Dieses Programm kann man aber mit einer Schleife abkürzen.

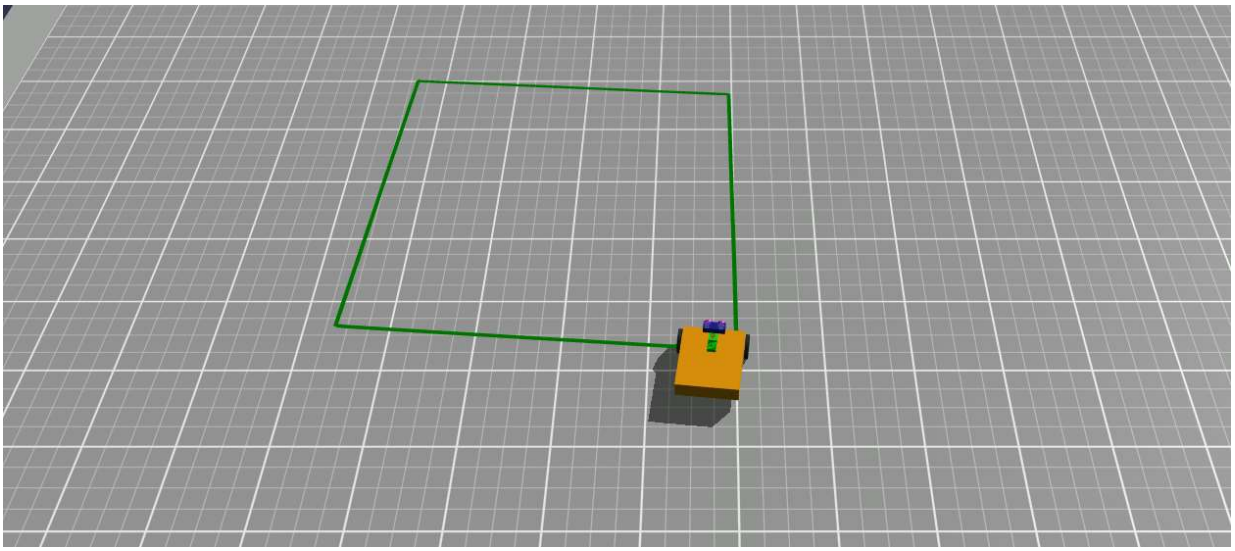


Der `repeat`-Block aus der `Loops`-Kategorie (Schleifen) repräsentiert eine `repeat`-Schleife. Damit wird ermöglicht, eine Anweisung oder einen ganzen Block von Anweisungen mehrfach auszuführen. Der `repeat`-Block ist vor allem dann nützlich, wenn die Anzahl der gewünschten Wiederholungen vorab bekannt ist. Der `repeat`-Block hat viele Ähnlichkeiten mit dem auch in GearsBot verfügbaren

`for`-Block. In vielen anderen Sprachen werden `repeat`-Block und `for`-Block wegen ihrer Ähnlichkeit als `for`-Schleife (`for` loop) zusammengefasst.

Später werden wir auch andere Schleifen kennenlernen

Ergebnis



Nach der Geradeausfahrt dreht das Fahrzeug am Stand (gleiche Leistung der Motoren aber unterschiedliche Vorzeichen). Man beachte, dass die Drehung am Stand um 0.67 Umdrehungen etwa einen Drehungswinkel des Fahrzeugs um 90° ergibt. Dadurch landet das Fahrzeug nach vier solchen Fahrten und Drehungen ungefähr wieder am Ausgangspunkt. Rufen wir uns in Erinnerung, dass es sich bei den Umdrehungen (in dieser Version des **`move-tank`**-Blocks) um Radumdrehungen handelt. Die 0.67 haben wir durch Ausprobieren gefunden, wir können den Wert aber auch theoretisch ermitteln. In GearsBot können wir im Menüpunkt

Robot > Select Robot

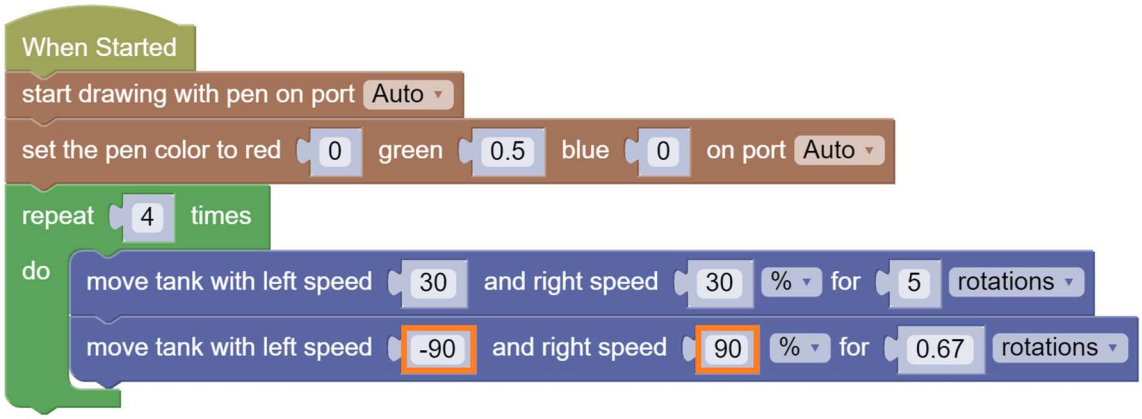
den Radumfang und Radabstand ablesen.

Raddurchmesser 5.6	Umfang 5.6 * π
Radabstand 15.2	Umfang der Kreisbahn, welche die Räder bei Drehung am Stand ausführen 15.2 * π

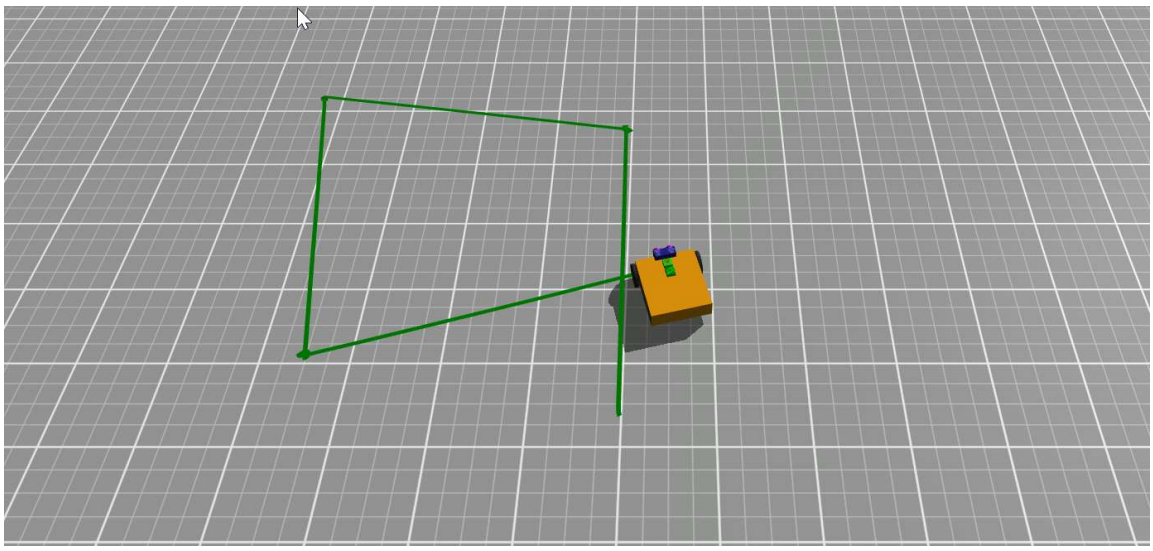
Wie viele Radumfänge passen in einen Viertelkreis der Drehungskreisbahn?

5.6 * π * x = 15.2 * π / 4
x = 15.2 / (5.6 * 4)
x = 0.678…

Der berechnete Wert entspricht fast dem praktisch ermittelten Wert. Das ist einerseits ein gutes Zeichen, andererseits sollten wir den theoretischen Wert in diesem Fall nicht überbewerten, denn folgende kleine Änderung im Programm zeigt, dass der genaue Wert auch von anderen Faktoren abhängt.



Ergebnis



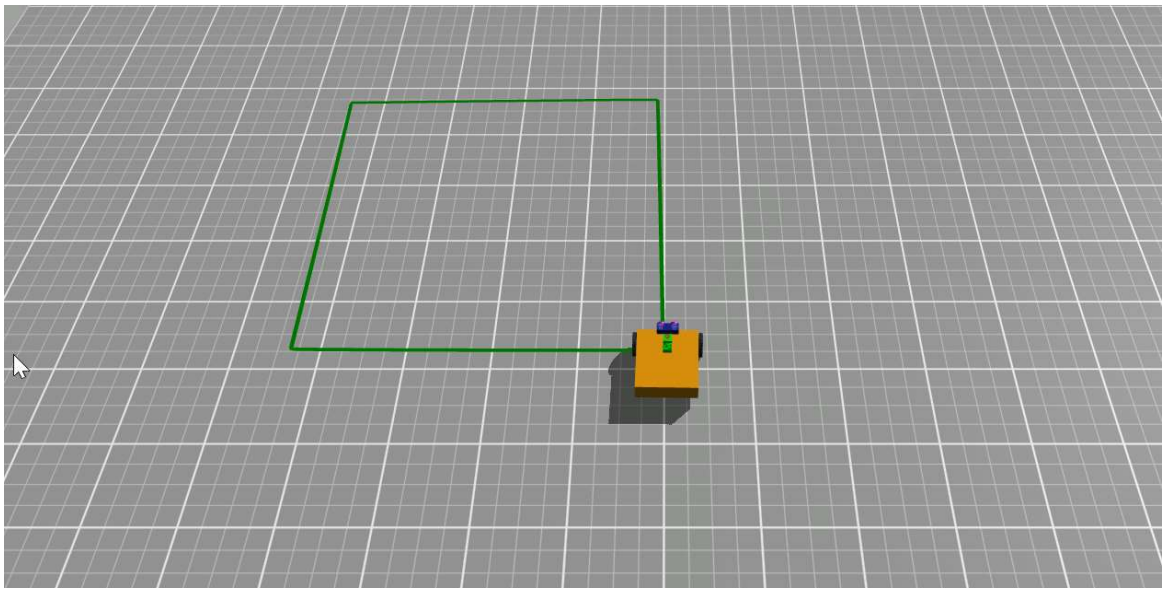
Das erzielte Ergebnis hängt also zum Beispiel auch von der Drehgeschwindigkeit ab. Die Ergebnisse können stark variieren. Einerseits kann es sein, dass das Fahrzeug weiter als 4*90 also insgesamt 360° dreht, wie in der Abbildung oben. Es kann aber bei anderen Versuchen auch vorkommen, dass das Ergebnis deutlich unter 360° liegt. Warum ist das so? Betrachten wir kurz den ganzen Vorgang aus physikalischer Sicht.

Nach der Geradeausfahrt bekommen die beiden Motoren den Befehl sich gegenläufig zu drehen und starten. Das Fahrzeug verfügt aber über eine gewisse Masse und damit über eine gewisse Trägheit. Das Fahrzeug wird sich der Drehung also zuerst widersetzen. Dieser Effekt kann so stark sein, dass die Räder für einen kurzen Moment durchrutschen, wodurch ein Teil der Drehung verloren geht. Umgekehrt ist es aber am Ende der Drehung. Dann bleiben die Räder abrupt stehen, der Drehimpuls des Fahrzeugs kann aber dazu führen, dass die Räder wiederum leicht durchrutschen und das Fahrzeug dadurch überdreht. Diese beiden Effekte werden sich teilweise gegenseitig aufheben, die verlorene Drehung zu Beginn wird am Ende durch Überdrehung wieder ergänzt. In der Praxis (und die Simulation trägt diesem Umstand Rechnung) werden diese beiden Effekte aber nie genau gleich groß sein, weil es bei praktischen Versuchen durch äußere Faktoren immer zu Schwankungen kommt.

Ein wichtiger Teil der Robotik ist es deshalb, diese Schwankungen gering zu halten oder zu korrigieren.

Schwankungen geringhalten

Eine Möglichkeit Schwankungen gering zu halten ist es, abrupte Bewegungen zu vermeiden. Die Drehgeschwindigkeit könnte generell herabgesetzt werden oder das Fahrzeug könnte zuerst langsam anfahren und dann stufenweise beschleunigen. Auch der Bremsvorgang könnte auf diese Art kontinuierlich erfolgen. Manche Robotiksysteme verfügen über die Möglichkeit stufenlos zu beschleunigen oder abzubremsen. Für das Abbremsen ist diese Funktion in GearsBot mit der Option „**`coast`**“ im **`stop`**-Block umgesetzt.

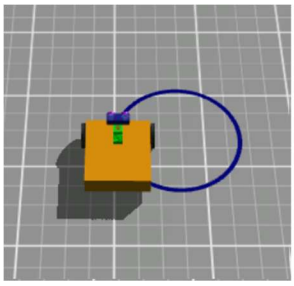
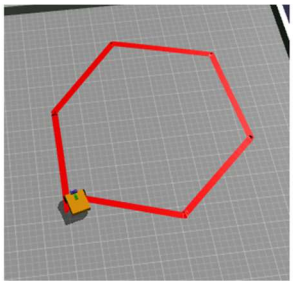
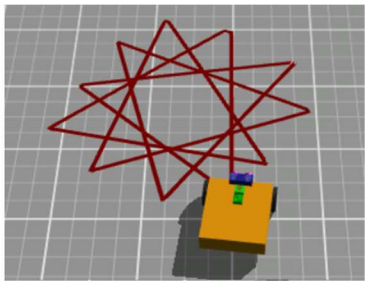


Korrigieren

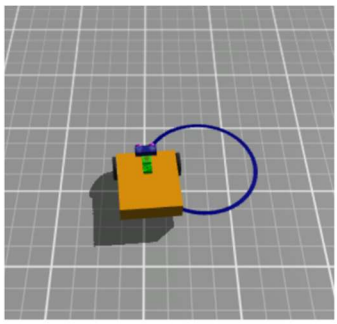
Später werden wir sehen, wie mithilfe von Sensoren Abweichungen korrigiert werden können.

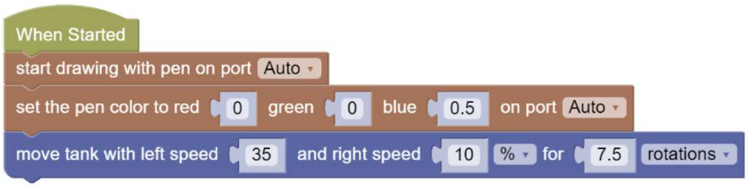
Aufgaben

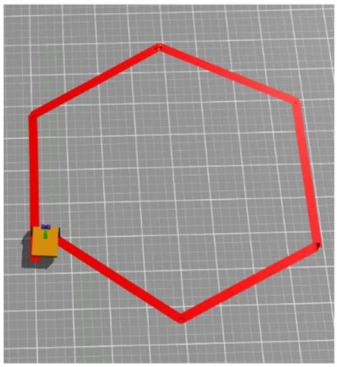
Erzeuge mit geeigneten Programmen folgende Figuren:

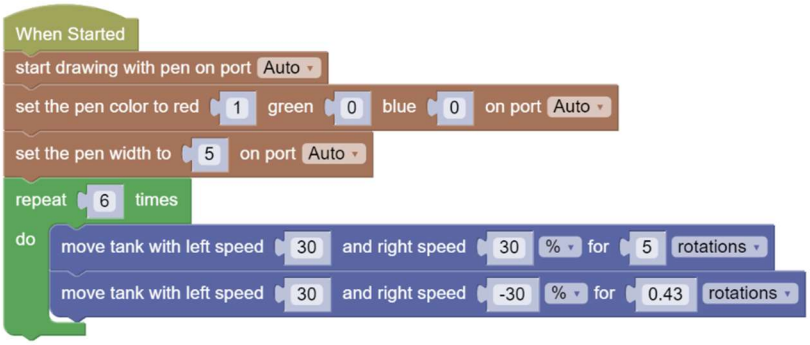




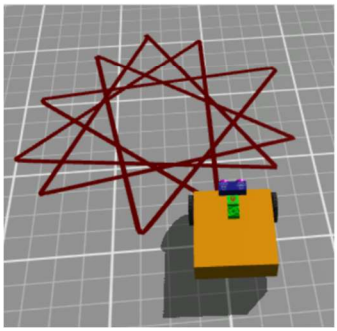
Lösungen

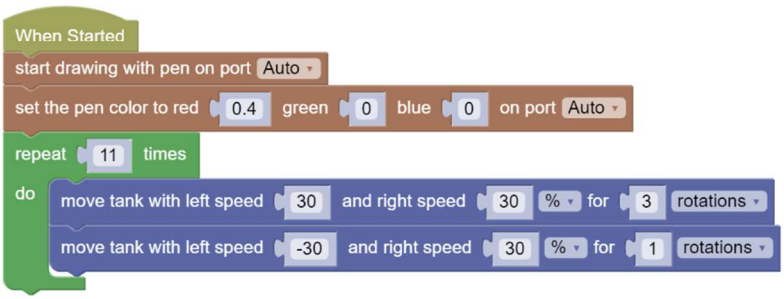



When Started
start drawing with pen on port Auto
set the pen color to red 0 green 0 blue 0.5 on port Auto
move tank with left speed 35 and right speed 10 % for 7.5 rotations



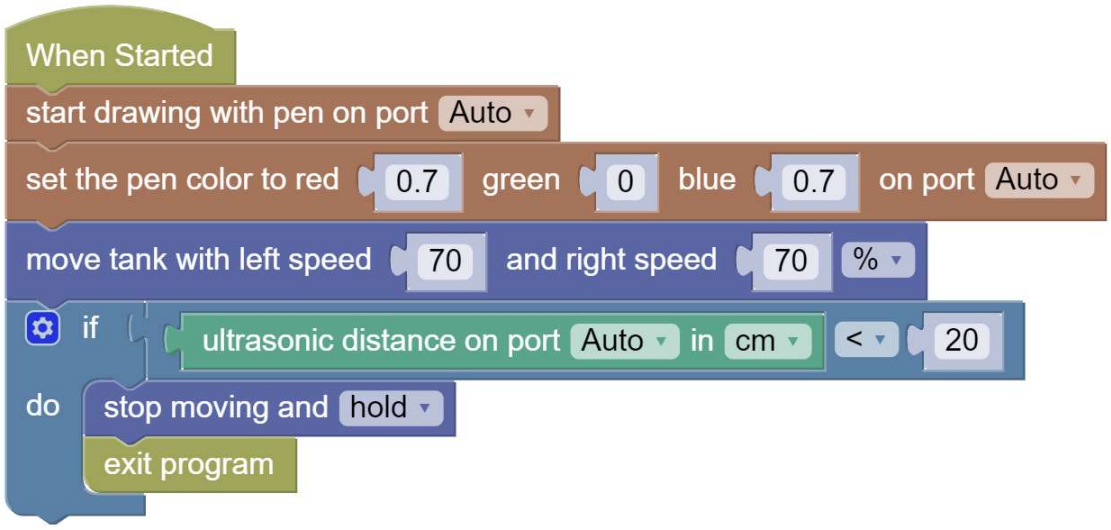

When Started
start drawing with pen on port Auto
set the pen color to red 1 green 0 blue 0 on port Auto
set the pen width to 5 on port Auto
repeat 6 times
do
move tank with left speed 30 and right speed 30 % for 5 rotations
move tank with left speed 30 and right speed -30 % for 0.43 rotations




When Started
start drawing with pen on port Auto
set the pen color to red 0.4 green 0 blue 0 on port Auto
repeat 11 times
do
move tank with left speed 30 and right speed 30 % for 3 rotations
move tank with left speed -30 and right speed 30 % for 1 rotations

Hindernisvermeidung

Folgendes Programm wirkt auf den ersten Blick vernünftig, wird aber nicht funktionieren. Finde den Fehler:



Erklärung

Nach Programmstart fährt das Fahrzeug los. Unmittelbar nach dem Losfahren wird einmalig der Abstand zur Wand überprüft. Da das Fahrzeug sich noch fast beim Ausgangspunkt befindet, wird die Bedingung nicht erfüllt sein (abhängig von der Lage des Ausgangspunkts). Die Anweisungen „**`stop moving`**“ (Kategorie **`Motion`**) und „**`exit program`**“ (Kategorie **`Control`**) werden also nicht ausgeführt. Deshalb fährt das Fahrzeug weiter und stößt gegen die Wand, weil die Überprüfung nicht mehr ausgeführt wird.

Im Programm oben gibt es ein paar neue Bausteine. Der `if`-Block oder auch `if`-Anweisung überprüft, ob eine Bedingung erfüllt ist. Ist das der Fall, dann wird die Anweisung oder der Anweisungsblock innerhalb des `if`-Blocks ausgeführt. Andernfalls wird nach dem `if`-Block fortgesetzt. Bedingung ist in unserem Fall die Frage, ob der Sensorwert des Ultraschallsensors kleiner als 20 ist. Der `if`-Block und der Vergleichsblock für die Bedingung stammen aus der Blockkategorie `Logic`, der Sensorblock aus der Kategorie Sensors. Was „`port Auto`“ bedeutet wurde bereits oben erklärt (wenn es nur einen Sensor dieser Art im Fahrzeug gibt, dann muss die Portnummer nicht angegeben werden, weil der Simulator diese Information nicht braucht. In einem Hardware-Kit müsste man angeben, an welchem Port der Sensor angesteckt ist).

`If`-Anweisungen sind für Programmiersprachen unerlässlich. Sie ermöglichen die Reaktion des Programms auf äußere Einflüsse und innere Zustände. Später werden wir sehen, dass es verschiedene Varianten von `if`-Anweisungen gibt. Mit der `if`-Anweisung haben wir nun den letzten Baustein für eine vollständige Programmiersprache zur Hand. Die wesentlichen Grundbausteine jeder herkömmlichen höheren Programmiersprache sind Anweisungen, Schleifen und Verzweigungen (Entscheidungen).

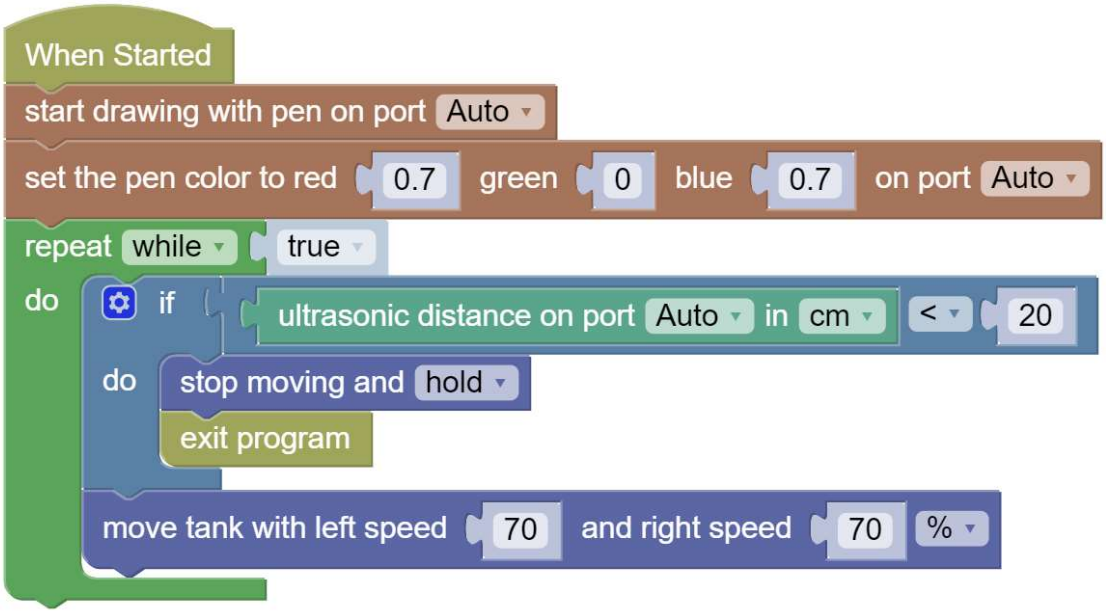
Der `stop`-Block aus dem obigen Programm ist oben auch erstmalig verwendet. Er hält das Fahrzeug an und mit der Option „hold“ bleibt es abrupt stehen. Die Option „break“ verhält sich wie „`hold`“, „`coast`“ lässt das Fahrzeug ausrollen, der `exit`-Block (Kategorie Control) beendet das Programm. Achtung! Eine eventuelle Motoranschaltung bleibt bestehen. Das Fahrzeug bewegt sich weiter. Ist das nicht gewünscht, muss vor dem `exit`-Block unbedingt der `stop`-Block ausgeführt werden.

Sense-Plan-Act (SPA) Zyklus

Das Programm im vorigen Beispiel hat nicht wie gewünscht funktioniert und der Roboter ist gegen das Hindernis gestoßen. Mithilfe des Sense-Plan-Act Zyklus können wir das Programm verbessern:

- **Sense**: Der Roboter bekommt erforderliche Daten über seine Umgebung geliefert
- **Plan**: Der Roboter verarbeitet die Daten und errechnet die beste Vorgangsweise in Bezug auf eine Zielvorgabe
- **Act**: Der Roboter setzt die berechneten Schritte in die Tat um

Diese Schritte soll der Roboter nicht nur einmalig, sondern wiederholt ausführen, bis das Ziel erreicht ist. Damit lassen auch die Unzulänglichkeit des vorherigen Programms beheben:



Erklärung

Im Programm oben wird nicht nur 1x überprüft, ob die Wand näher als 20 cm ist, sondern immer wieder. Solange, bis der Abstand unterschritten ist. Dann werden die Motoren gestoppt und die Schleife wird verlassen bzw. das Programm beendet.

Das ursprüngliche, nicht nach Wunsch funktionierende Programm besteht nur aus einer Anweisungsfolge, die abgearbeitet und dann beendet wird. Erst durch die `while`-Schleife bleibt der Roboter dauerhaft aktiv. Alle Programme, die dauerhaft aktiv bleiben sollen, brauchen diese Konstruktion. Aus Sicht des Programmflusses ist die Schleife gleichsam der Herzschlag des Programms.

Der Sense-Abschnitt ist in unserem Fall das Lesen des Sensorwerts, der Act-Abschnitt ist der **`move-tank`**-Befehl bzw. der **`Stop`**-Befehl. Zwischen diesen beiden Abschnitten befindet sich der Plan-Abschnitt, welcher in diesem Beispiel sehr einfach ausfällt, nämlich mit einem **`if`**-Vergleich auf die Distanz. Normalerweise ist der Plan-Abschnitt umfangreicher.

Der `repeat-while`-Block (in vielen andere Sprachen als `while`-Schleife bezeichnet) ist oben erstmalig vertreten. Das ist eine Schleife, die dann eingesetzt wird, wenn nicht vorab bekannt ist, wie oft sie ausgeführt werden soll, aber eine Schleifenbedingung angegeben werden kann. Die Bedingung wird vor jedem Schleifendurchlauf geprüft und weiter ausgeführt, wenn die Bedingung erfüllt ist. Ist die Bedingung nicht erfüllt, dann bricht die Schleife ab. Achtung! Die Person, die das Programm erstellt, ist selbst dafür zuständig, dafür zu sorgen, dass diese Abbruchbedingung auch jemals erreicht wird oder erreicht werden kann. Ist das nicht der Fall, dann handelt es sich um eine unbeabsichtigte Endlosschleife.

Einen Sonderfall stellen `repeat-while-true` (in anderen Sprachen nur `while-true`) Schleifen dar. Es sind beabsichtige Endlosschleifen, denn der Ausdruck „`true`“ (wahr) evaluiert immer zu wahr, ist also immer erfüllt. Diese Schleifen laufen so lange, bis das Programm aus irgendeinem Grund abgebrochen wird (durch eine Person, Stromausfall, Rechnerabsturz, …).

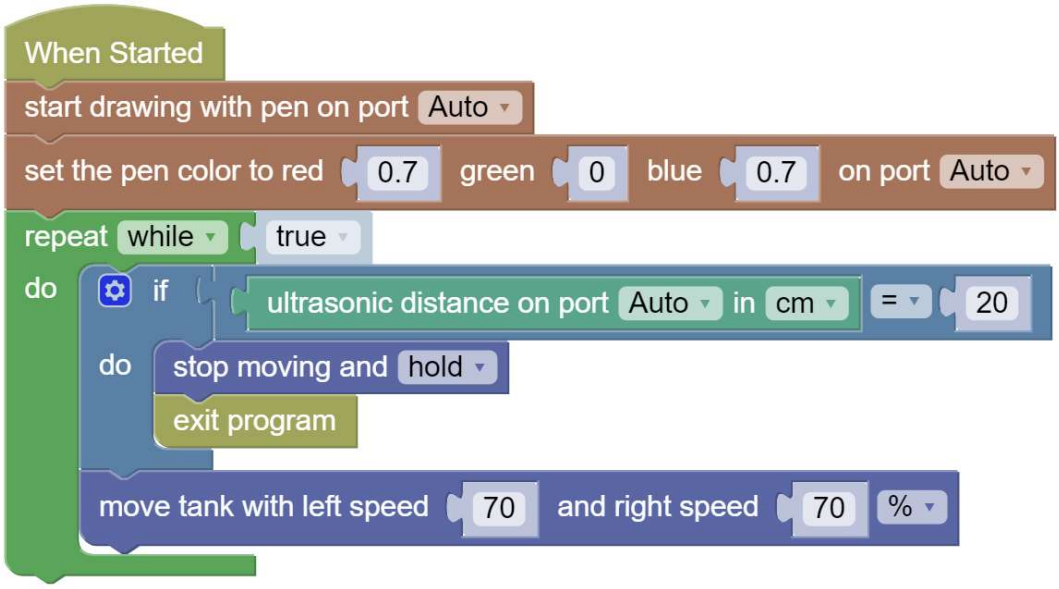
Ausnahme:
Was auf den ersten Blick wie eine Endlosschleife aussieht, muss nicht unbedingt eine sein. Es gibt nämlich den `break-out-of-loop`-Block (Kategorie Loops). Ist dieser innerhalb einer Schleife vorhanden und wird er jemals erreicht, dann kann damit auch eine vermeintliche Endlosschleife abgebrochen werden. Der `exit`-Block im Fall oben verhält sich ähnlich. Da er das Programm beendet, beendet er auch die Schleife. Auch eine `return`-Anweisung (Kategorie Functions) würde so eine Schleife beenden.

Grundlegende Datentypen

`true` und `false` nennt man auch Boolean-Werte. Sie stehen für „wahr“ und „falsch“. Ganzzahlige und Fließkommawerte hatten wir bereits als Einstellungswerte bei den Blöcken und in Ausdrücken (zum Beispiel wie aus Sensorwert `< 20` abgefragt wurde). Man beachte, dass bei Fließkommawerten der „**.**“ (Punkt) und nicht „**,**“ (Komma) eingesetzt wird. Als letzten grundlegenden Datentypen fehlen nur noch Zeichenketten. Diese werden mit ‘ oder “ umschlossen (einfache oder doppelte Hochkomma).

⚠ Achtung

Wer findet den Fehler in folgendem Programm?



Das Programm ist ähnlich der Version von zuvor und wirkt auch von der Bedeutung der einzelnen Befehle her fast identisch, hat aber einen subtilen Unterschied, der dafür sorgt, dass das Programm nicht wie gewünscht funktioniert.

Lösung

Die Abfrage „**= 20**“ wird nicht das gewünschte Ergebnis liefern. Der Rückgabewert des Ultraschallsensors ist eine Fließkommazahl. Fließkommazahlen lassen sich nicht so einfach mit „=“ vergleichen, denn es wäre ein großer Zufall, dass beide Zahlen bis auf die letzte Ziffer übereinstimmen. Beim Vergleich von Fließkommazahlen müsste man so vorgehen, dass man einen Bereich angibt und zwei Zahlen gelten als gleich, wenn dieser Bereich unterschritten wird, also wenn:

|f1-f2|< Epsilon

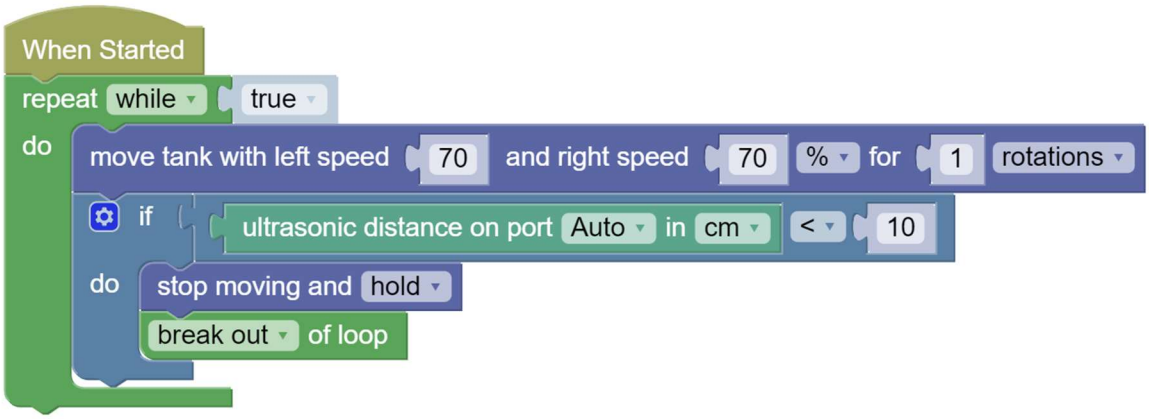
Aber das Ganze können wir uns ersparen, wenn wir uns statt = für < entscheiden, was in dem Zusammenhang fast die gleiche Bedeutung hat.

Oben haben wir argumentiert, dass der Fließkomma-Sensorwert ein Hauptgrund dafür ist, dass das Programm scheitert. Damit wollten wir auch nicht sagen, dass das Programm unproblematisch wäre, wenn der zurückgelieferte Wert ganzzahlig wäre. Der Abstand zum Hindernis wird bei jedem Schleifendurchlauf gemessen. Könnte es sein, dass eine Messung 21 zurückliefert und die nächste 19, sodass die Abbruchbedingung wieder scheitert und das Fahrzeug nicht anhält und gegen die Mauer fährt? Wie oft die Messung aufgerufen wird, die Abtastfrequenz der Messung, hängt von verschiedenen Faktoren ab, wie zum Beispiel der Rechenleistung des Systems und wie viele Befehle sonst noch in der Schleife ausgeführt werden. Auch die Geschwindigkeit des Fahrzeugs ist maßgeblich, ob der Wert von 20 in der Messreihe gar nicht auftaucht. Auch aus diesem Grund ist der **<**-Operator in so einem Fall immer zu bevorzugen.

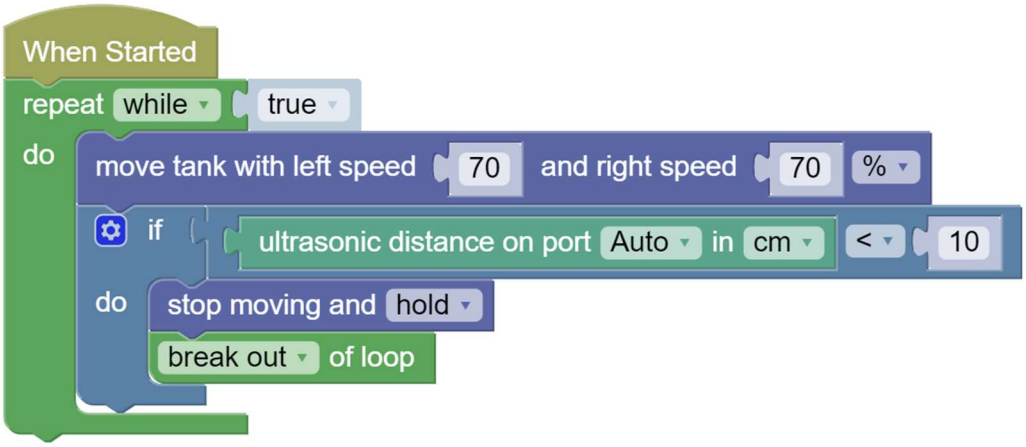
Abtastrate (Abtastfrequenz)

Wie oft wird die Sensor-Abfrage in der obigen **`while-true`**-Schleife pro Zeiteinheit ausgeführt? Das hängt davon ab, wieviel Rechenzeit die anderen Anweisungen in der Schleife verbrauchen oder wie lange sie die Verarbeitung der Schleife blockieren. Bildschirmausgaben brauchen oft ein Vielfaches von einfachen Berechnungsaufgaben, aber ganz drastisch wirken sich Anweisungen wie **`move-tank`**-Aufrufe mit der „**`rotations`**“-Option aus, denn sie haben blockierende Wirkung. Der Programmfluss stoppt so lange, bis die Umdrehungen ausgeführt sind.

Folgendes Programm wird nicht wie geplant funktionieren, das Fahrzeug wird immer zuerst die eine Umdrehung ausführen und dann erst die Sensormessung vornehmen. Dies resultiert in einer ‘ruckelnden’ Bewegung bzw. unter Umständen mit einer Kollision mit der Wand.



Die folgende Variante wird funktionieren. Der blockierende Befehl wurde ersetzt.



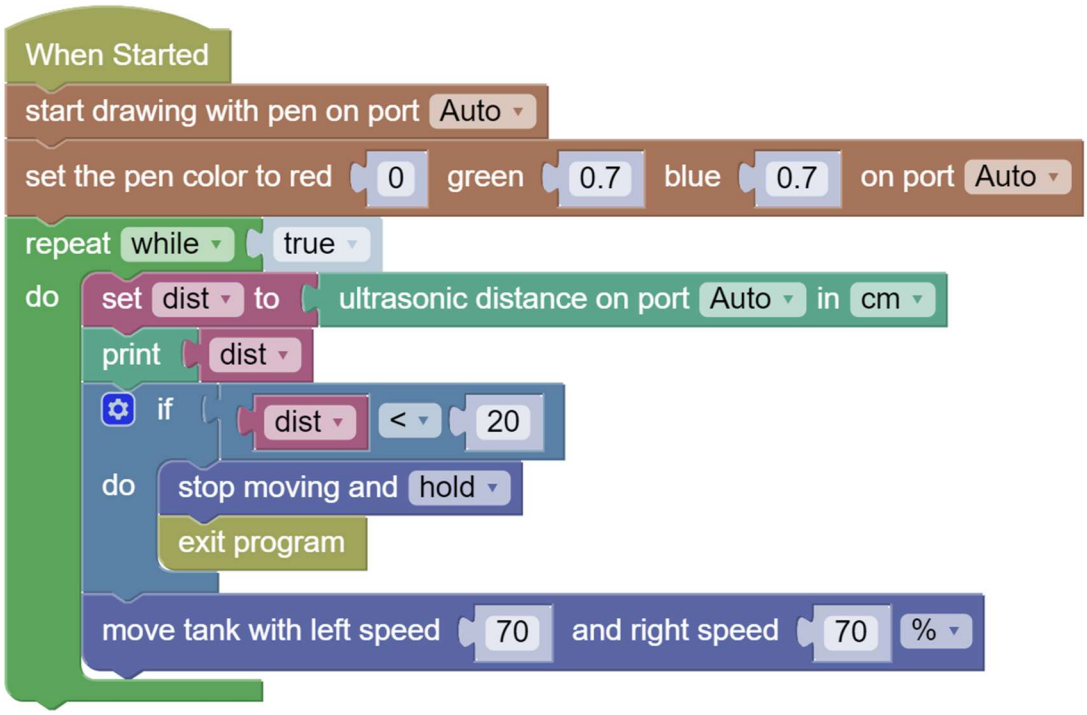
Sensorwerte

Sensorwerte sind eine häufige Fehlerquelle in Programmen. Warum? Programmierfehler haben sehr viel mit Psychologie zu tun. Fehler im Code beruhen oft auf bewussten oder unbewussten, aber falschen Annahmen der programmierenden Person. Sensorwerte sind da keine Ausnahme und mit ihnen kann man das Prinzip gut veranschaulichen:

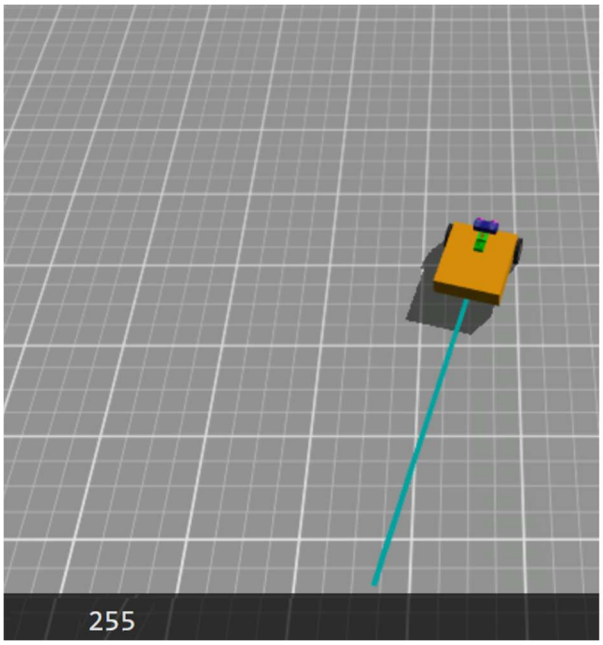
- In welcher Einheit liefert ein Abstandssensor das Ergebnis? m, cm, mm? Fuß, Zoll, …? Muss der Wert erst in eine ander Einheit umgerechnet werden?

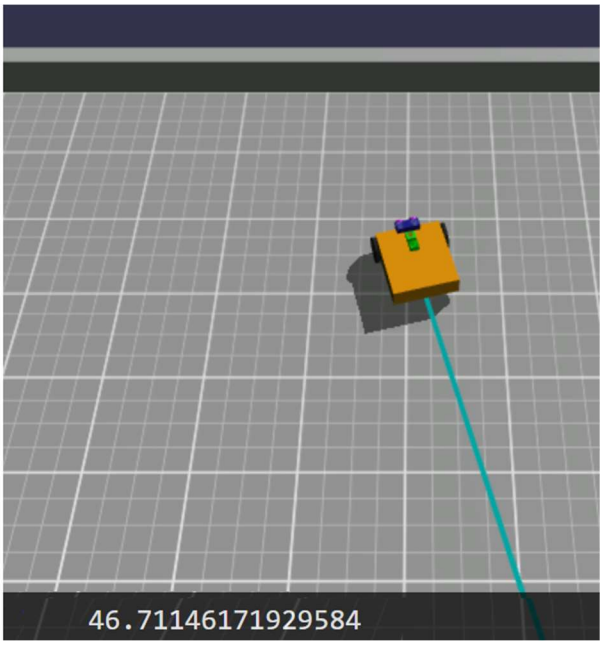
- Ist der Wert ganzzahlig, fließkomma, …?
- Wird überhaupt ein gültiger Wert zurückgeliefert?
- Stimmt die Portnummer, mit welcher der Wert abgefragt wurde?
- Was ist die maximale/minimale Entfernung, die gemessen werden kann?
- Welche Werte liefert der Sensor, wenn diese Werte über- oder unterschritten werden?
- Welche Werte liefert ein defekter Sensor?
- Unter welchen Bedingungen scheitert die Messung?
- Welcher Wert wird dann geliefert? 0, Null, -1, false, …?

Es ist deshalb eine gute Idee, während der Entwicklung eines Roboter-Programms, wenn irgendwelche Zweifel bestehen oder wenn das Programm nicht wie erwartet funktioniert, Sensorwerte auszugeben und so einen guten Überblick über die oben gestellten Fragen zu bekommen.



Ergebnis





In diesem Programm wird die Variable **`dist`** verwendet, um den Sensorwert zu speichern. Die Variable wird in der Kategorie angelegt und die entsprechenden Blöcke sind dann dort verfügbar. Das wäre nicht unbedingt notwendig. Man könnte auch den Sensorwert zweimal holen, einmal für die Ausgabe, einmal für den Vergleich. Die oben gewählte Vorgangsweise hat jedenfalls den Vorteil, dass genau der gleiche Wert angezeigt wird, der dann für den Vergleich herangezogen wird. Das wäre bei zweimaligem Aufruf nicht der Fall gewesen, weil zwischen beiden Aufrufen etwas Zeit verstreicht und sich daher der Abstand zum Hindernis ändert, wenn auch nur in geringem Ausmaß bei geringen Zeitdifferenzen. Die Ausgabe erfolgt in der dunklen Zeile ganz unten. Offensichtlich liefert der Sensor den ganzzahligen Wert 255, wenn kein Objekt im Messbereich ist. Ist ein Objekt im Messbereich, dann liefert der Sensor einen Fließkommawert.

Wird eine Variable in der Kategorie Variables angelegt, dann kann über den dort verfügbaren Variablen-Block auf sie zugegriffen werden. Mit `set` und `change` kann diese Variable initialisiert und geändert werden. In beiden Fällen spricht man auch von Zuweisungen. Zuweisungen gehören zur Gruppe der Anweisungen. Mit `print` (Kategorie Text) werden Ausgaben durchgeführt. Print ist unter anderem zur Fehlersuche sehr hilfreich.

Achtung!
Wie Zuweisungen geschrieben werden, ist in verschiedenen Programmiersprachen nicht ganz einheitlich. Grund dafür ist die Verwechslungsgefahr mit einem Vergleich. In der Mathematik steht "=" für beides, trotz der völlig verschiedenen Bedeutung. Das ist aber deshalb kein Problem, weil aus dem Zusammenhang fast immer klar wird, was gemeint ist. Das gilt nicht für Computerprogramme. Die müssen entweder eindeutig sein oder es muss eine ganz klare Regel geben, mit der die Zweideutigkeit aufgelöst wird. Folgende Tabelle zeigt, wie verschiedene Programmiersprachen damit umgehen. In der ersten Zeile ist die Variante zu sehen, die in GearsBot Verwendung findet, die zweite Zeile zeigt die häufigste Variante bei textorientierten Programmiersprachen, aber auch die Variante von Zeile drei ist üblich. Sie wird gern in Pseudocode verwendet, den wir später kennenlernen.

Programmiersprache	Zuweisung	Vergleich
GearsBot	set	=

Python, C++, Java, ...	=	==
Pseudocode (aber auch zum Beispiel Pascal)	:=	=

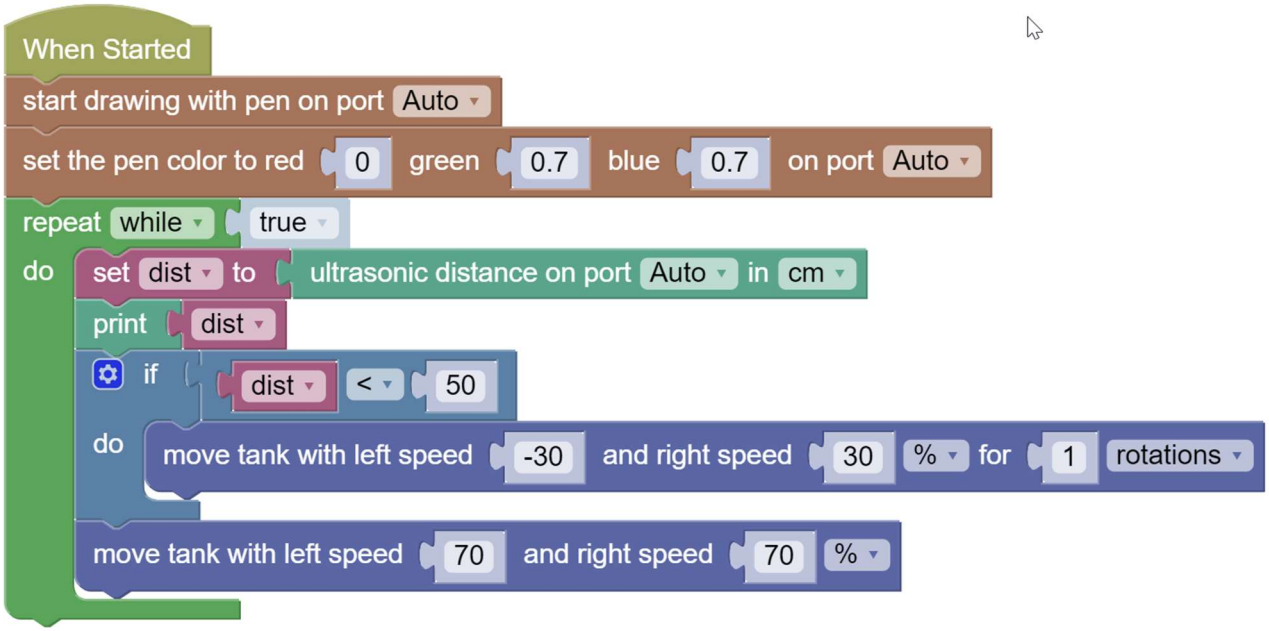
Keine Sorge, falls das auf den ersten Blick verwirrend wirkt. Auch erfahrene Programmierende verwechseln manchmal Zuweisung und Vergleich und schreiben zum Beispiel "=", wenn sie "==" meinen.

Manche Programmiersprachen bieten Variablen an, die nach einer ersten Zuweisung nicht mehr geändert werden können. Dann spricht man von Konstanten.

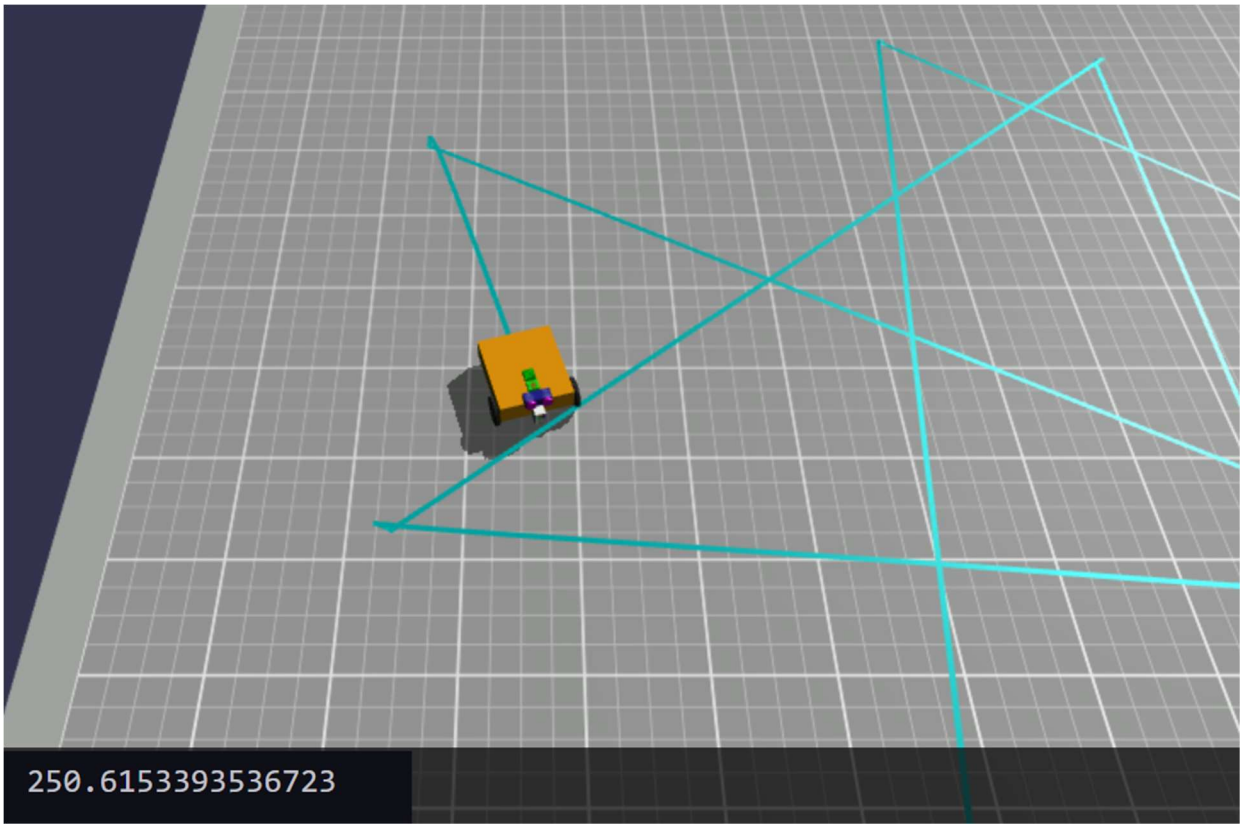
Angenommen eine programmierende Person hätte ein Programm geschrieben, das dafür sorgt, dass sich das Fahrzeug von einem Objekt entfernt und außerhalb einer 0,5 m Zone stehen bleibt. Die Person nimmt an, dass die Sensorwerte in mm geliefert werden. Im Programm steht, die Motoren sollen so lange laufen, solange der Sensorwert < 500 ist. Tatsächlich liefert der Sensor aber den Wert in cm und 500 liegt außerhalb des messbaren Bereichs. Das Programm wäre gescheitert und die programmierende Person hätte nicht gewusst warum. Mithilfe der Ausgabe kann das nicht mehr passieren. Die Einheit des Werts und der Messbereich sind nun offensichtlich.

Jetzt fehlt uns nur noch ein kleiner Schritt zur geplanten Hindernisvermeidung. Statt „**`stop moving`**" und „**`exit program`**" lassen wir das Fahrzeug am Stand vom erkannten Hindernis wegdrehen und fahren dann weiter.

Programm Hindernisvermeidung:



Ergebnis



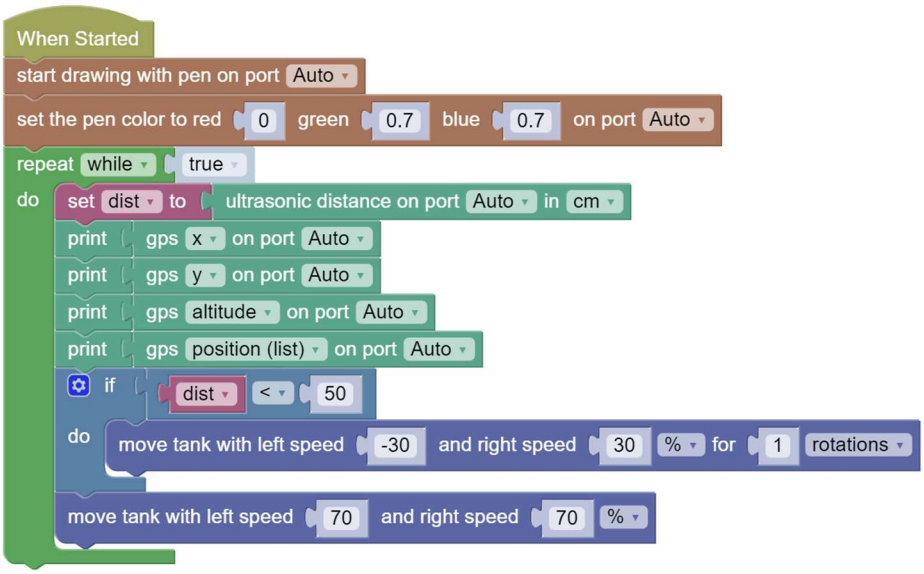
Zur Kontrolle lassen wir die Entfernung weiter anzeigen. Wird der Wert nicht mehr gebraucht, können wir ihn leicht wieder ausblenden, indem wir die print-Ausgabe entfernen.

Aufgaben

Modifiziere das Programm:

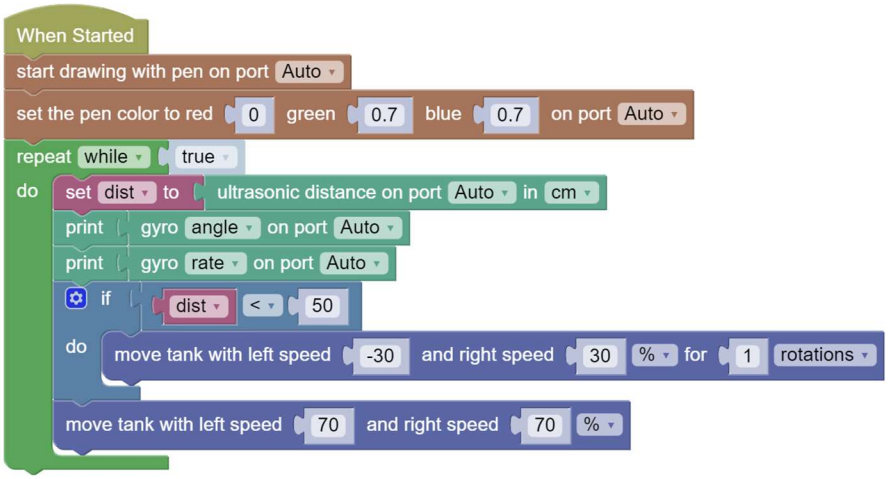
- Statt der Entfernung zum Hindernis soll die GPS-Position in allen möglichen Varianten (*x, y, altitude, list*) angezeigt werden.
- Statt der Entfernung soll der Wert des Gyro-Sensors ausgegeben werden (Option angle oder rate).
- Überlege die Bedeutung der ausgegebenen Werte.

Lösungen




```
-0.1812050342559814
103.6636352539063
6.281200408935547
(-0.1656931340694427, 107.8813781738281, 6.283285617828369)
```

Die x-, y-Werte des GPS-Sensors sind die Koordinaten der Bodenfläche. (0, 0) Ist die Startposition des Fahrzeugs im Mittelpunkt der Fläche. Der altitude-Wert ändert sich kaum, weil es keine großen Erhebungen gibt.



Der angle-Wert zeigt 360 bei einer vollen Drehung nach rechts. Mehrere Umdrehungen werden durch Vielfache davon angezeigt. Drehung nach links wird negativ angezeigt. Der rate-Wert ist bei Geradeausfahrt 0. Der Wert steigt nur während einer Drehung an und zeigt die Drehungsgeschwindigkeit.

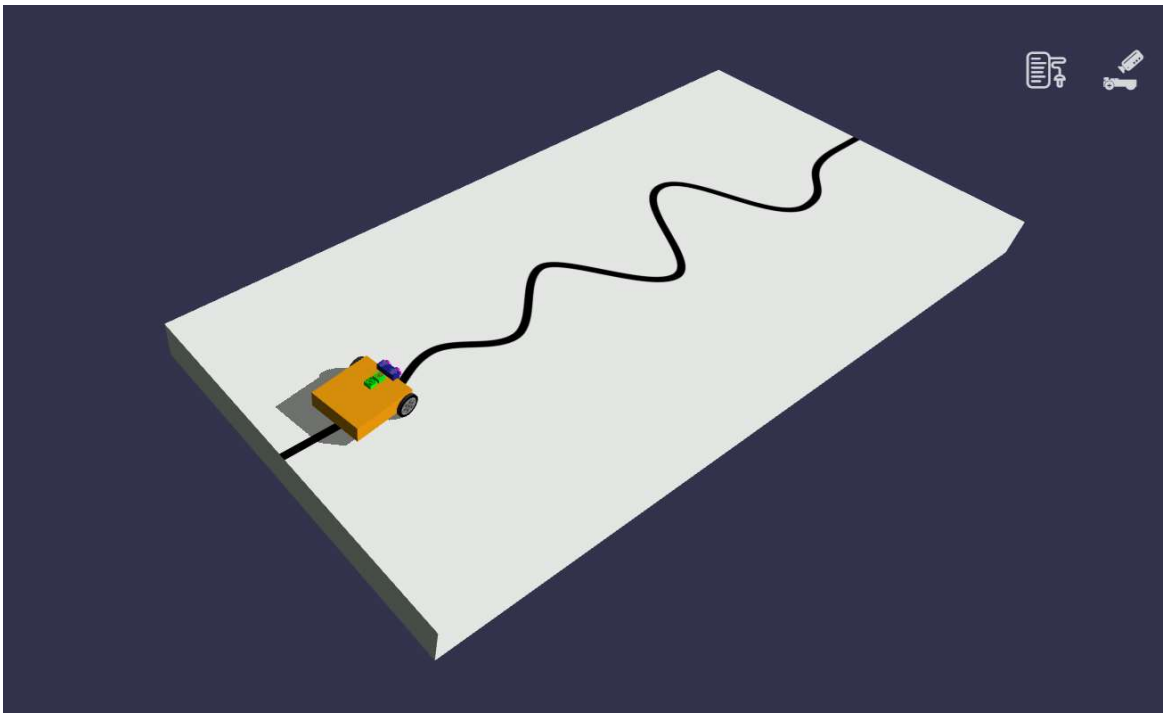
Linienfolger I (einfachste Version)

Der Linienfolger vereint die zuvor vorgestellten Konzepte in einem anschaulichen Anwendungsbeispiel. Es ist ein kurzes Programm mit dennoch interessantem Verhalten.

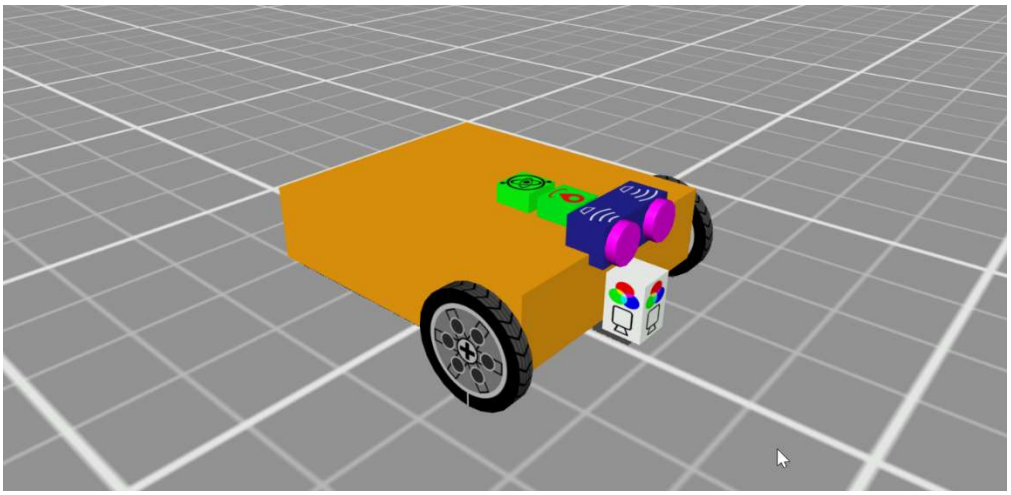
Zunächst laden wir in GearsBot im Menüpunkt

World > Select World

die Welt „*Line Following Challenges*“, wie unten zu sehen.

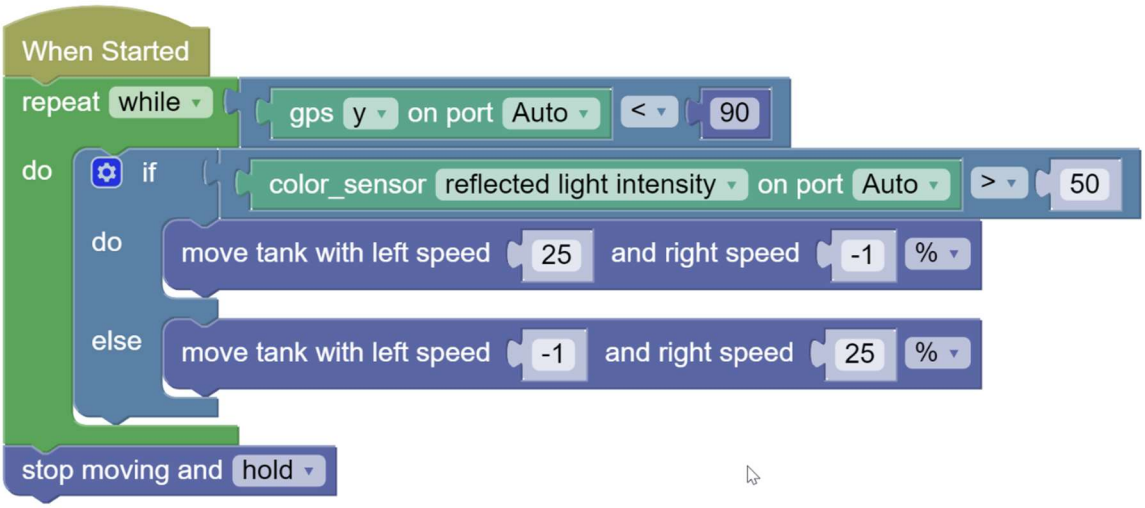


Als Fahrzeug kann das Standardfahrzeug zum Einsatz kommen.



Vorne am Fahrzeug ist der weiße, nach unten gerichtete Helligkeits-/Farbsensor zu erkennen. Nach vorne in blau ist der Ultraschallsensor zu sehen. Dahinter in grün befinden sich der GPS-Sensor und der Gyro-Sensor in dieser Reihenfolge.

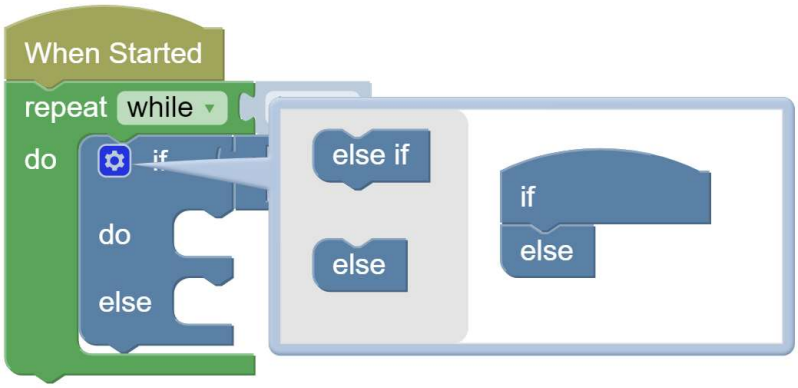
Zum Einsatz kommt dann folgendes Programm:



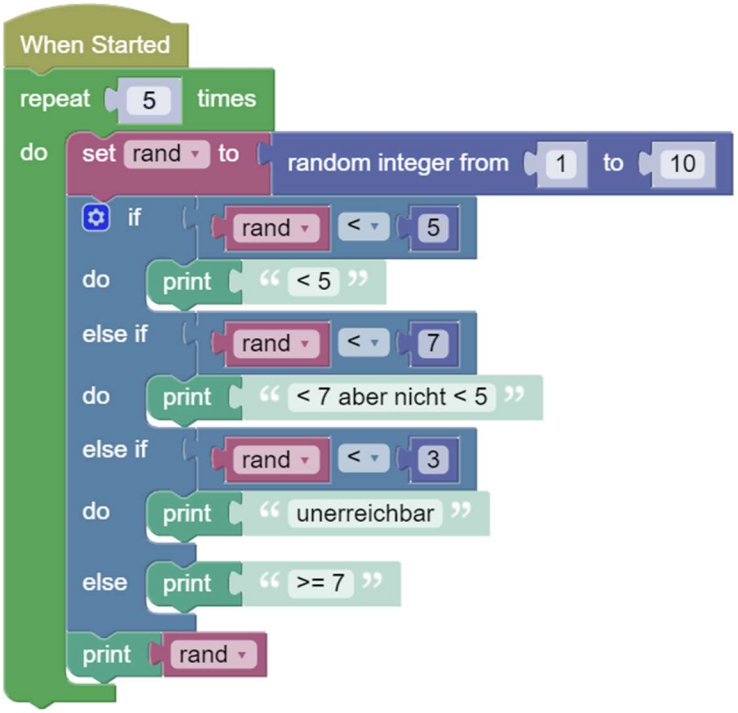
Erklärung

In diesem Fall ist die Abbruchbedingung der `while`-Schleife erfüllt, sobald die GPS-y-Koordinate den Wert 90 erreicht. Damit wollen wir verhindern, dass das Fahrzeug über den Rand der Welt hinausfährt und hinunterfällt.

Die `if`-Anweisung tritt nun in einer erweiterten Variante auf als zuletzt. `If`-Anweisungen können neben dem do-Zweig auch einen `else`-Zweig und einen oder mehrere `else-if`-Zweige haben. Konfiguriert wird das über das kleine Zahnrad. Durch Klicken auf dieses wird ein Dialog eingeblendet und durch Ziehen von Elementen kann man definieren, um welche Variante es sich handelt. `else-if`-Zweige haben jeweils eine eigene Bedingung, die überprüft wird. Es wird immer nur maximal ein Zweig durchgeführt: entweder der `if`, der `else-if` oder der `else`-Zweig, der zum Zug kommt, wenn keine der Bedingungen vorher erfüllt ist. Nochmaliges Klicken auf das Zahnrad schließt den Dialog wieder.



Achtung! Die Reihenfolge der `if`, `else-if` Bedingungen ist von Bedeutung und die programmierende Person ist selbst dafür verantwortlich, dass diese sinnvoll ist. Das folgende Beispiel verdeutlicht das:



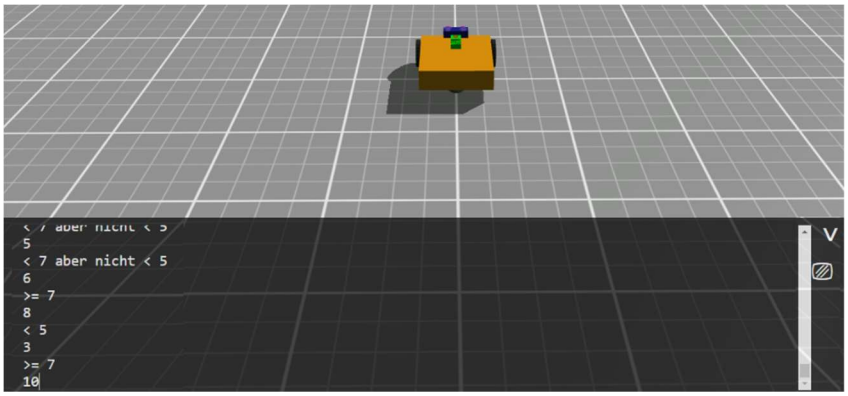
Hier wird der `random`-integer-Block (Kategorie Math) erstmals verwendet. Er liefert zufällige ganze Zahlen im angegebenen Bereich, inklusive der genannten Schranken.

- Im `if`-Zweig wird auf < 5 abgefragt und eine entsprechende Ausgabe gemacht. Wird dieser Zweig ausgeführt, dann wird die `if`-Anweisung danach sofort verlassen.
- Im ersten `else-if`-Zweig wird auf < 7 abgefragt. Vergessen wir dabei nicht, dass < 5 Fälle aber schon vorher abgehandelt wurden. Sie erreichen diesen Zweig nicht. Der Zusatz “nicht < 5” trägt dem Rechnung.
- Der nächste `else-if`-Zweig < 3 ist so nicht sinnvoll. Dieser Zweig kann nie erreicht werden denn Fälle < 3 wurden schon im Zweig < 5 abgehandelt. Diesen Zweig nennt man auch „toten Code“.

Behebung:

`< 3` müsste vor `< 5` abgefragt werden.

Ausgabe



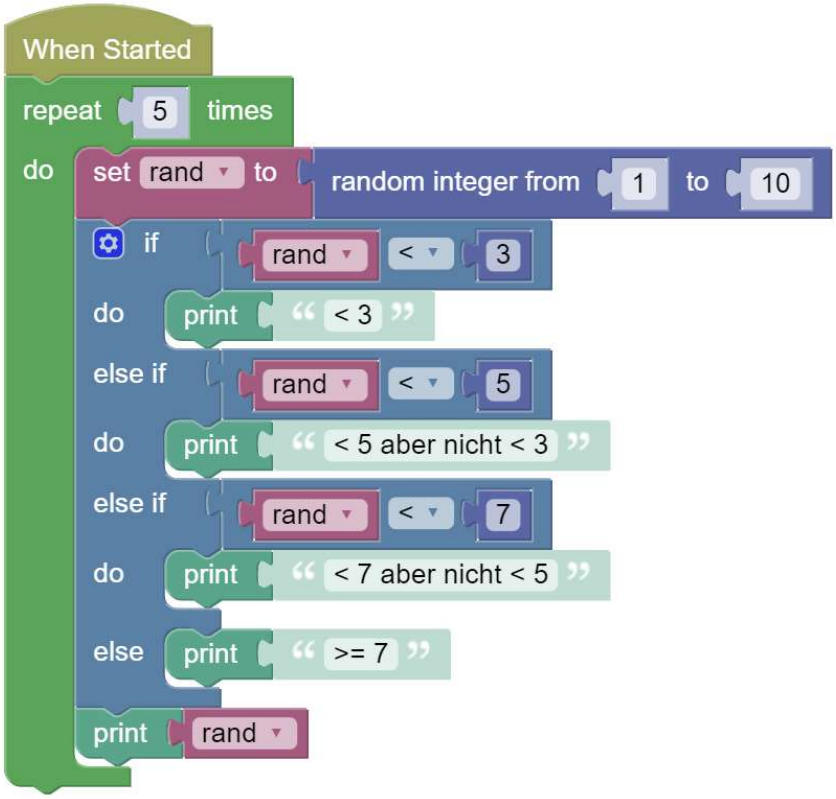
Mit dem Pfeilsymbol kann die Ausgabe für eine mehrzeilige Ansicht aus- und wieder eingeklappt werden. Mit dem Symbol darunter kann die Ausgabe gelöscht werden.

Aufgaben

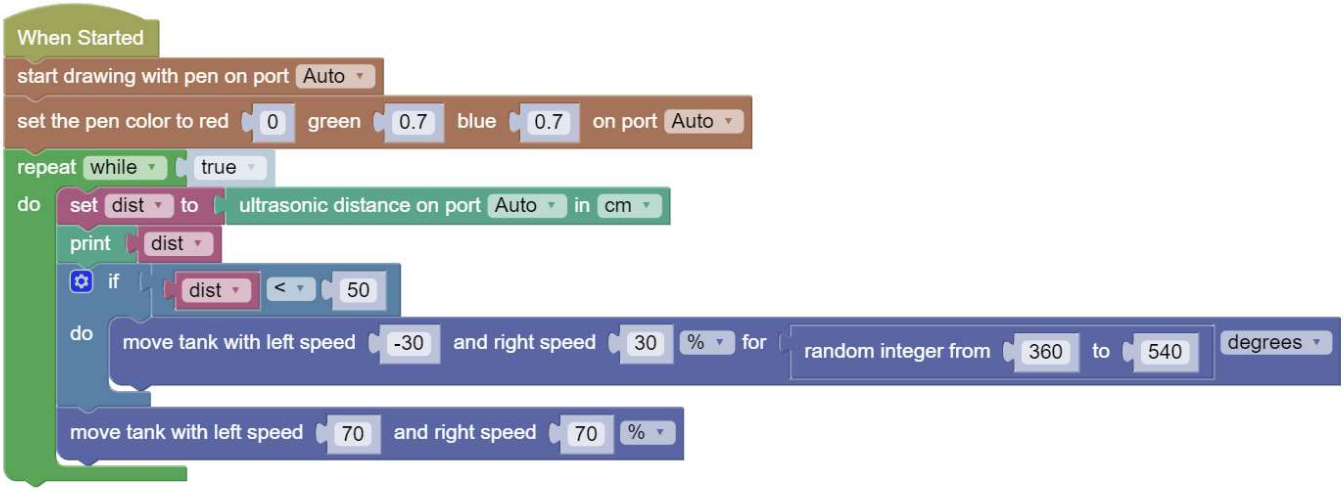
- Korrigiere das Programm, sodass alle Zweige erreichbar sind.
- Adaptiere das Programm Hindernisvermeidung. Beim Wegdrehen vom Hindernis soll nicht mehr um 1 Rotation gedreht werden, sondern zufällig zwischen 360° und 540°. Verwende dafür den **`random-integer`**-Block (Kategorie Math).
- Wie im vorigen Punkt, aber bei jedem Ausweichmanöver, soll auch die Geschwindigkeit zufällig auf 30-70% eingestellt werden, und zwar für beide Räder gleich. Dafür benötigst du eine Variable.
- Wie im vorigen Punkt aber die Geschwindigkeit soll bei beiden Rädern unabhängig und zufällig eingestellt werden. Dafür benötigst du keine Variablen.

Lösungen

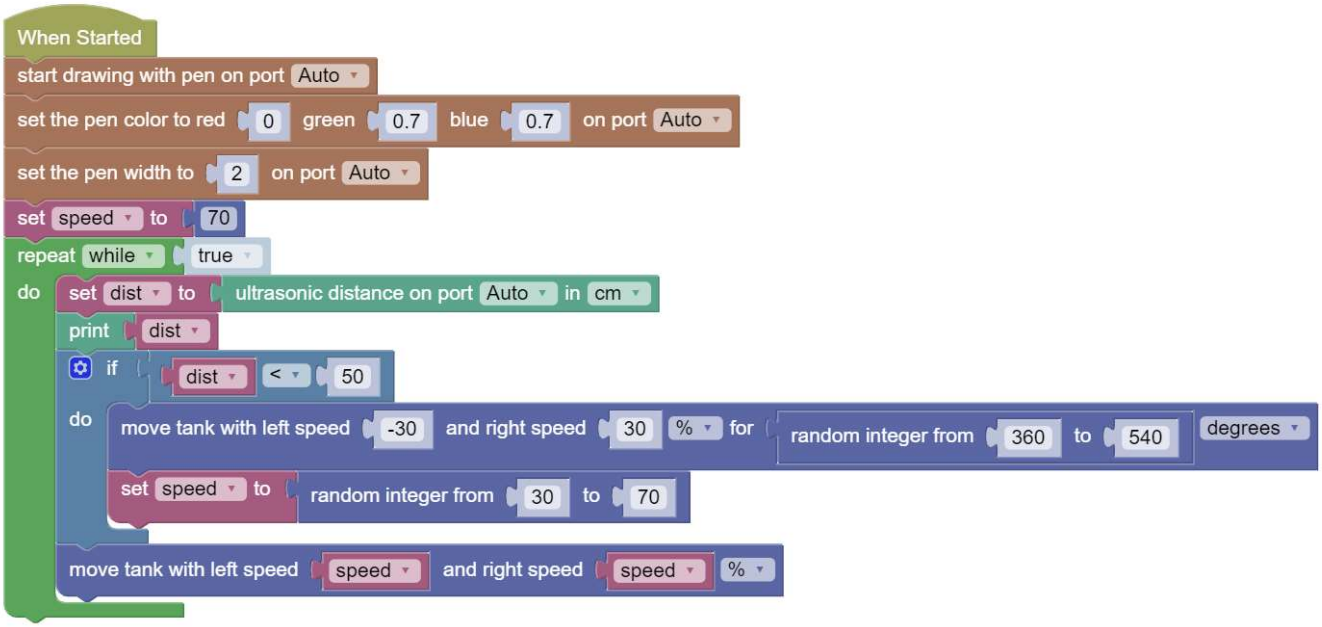
Aufgabe 1:



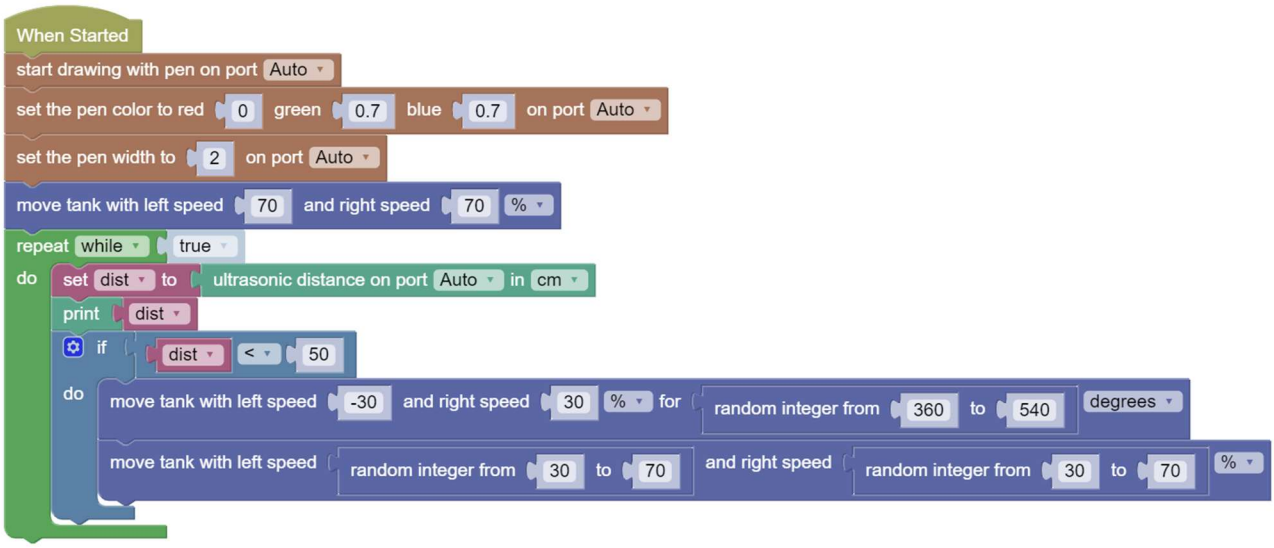
Aufgabe 2:

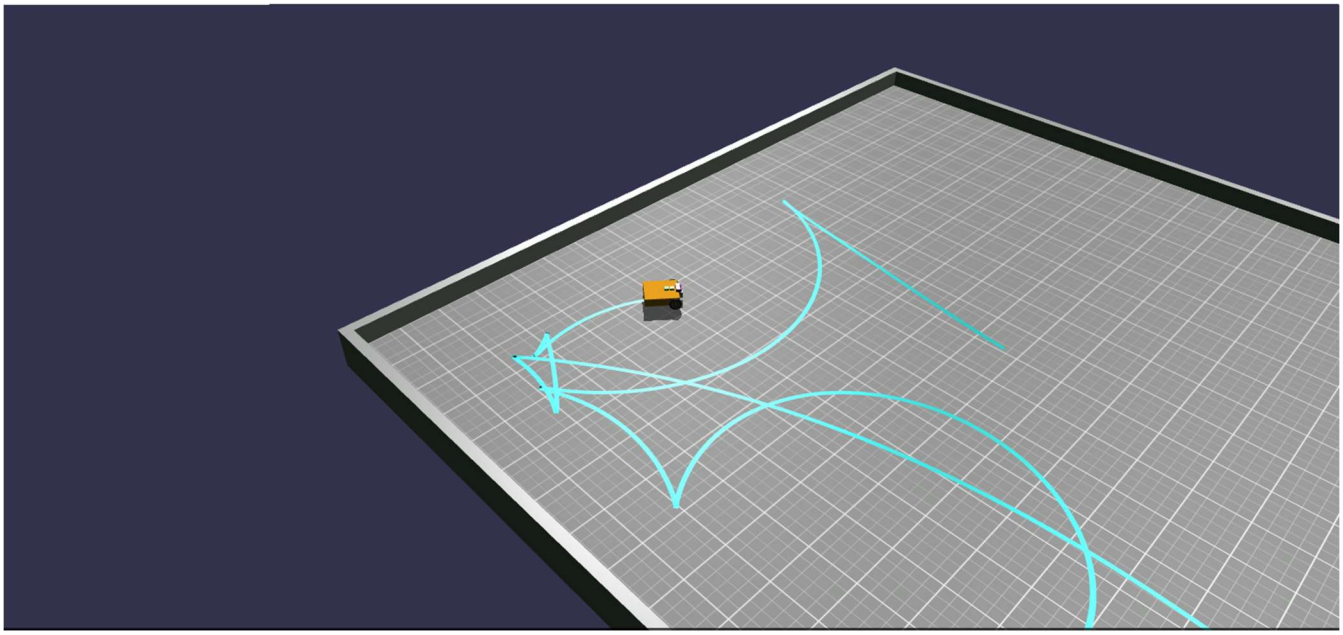


Aufgabe 3:

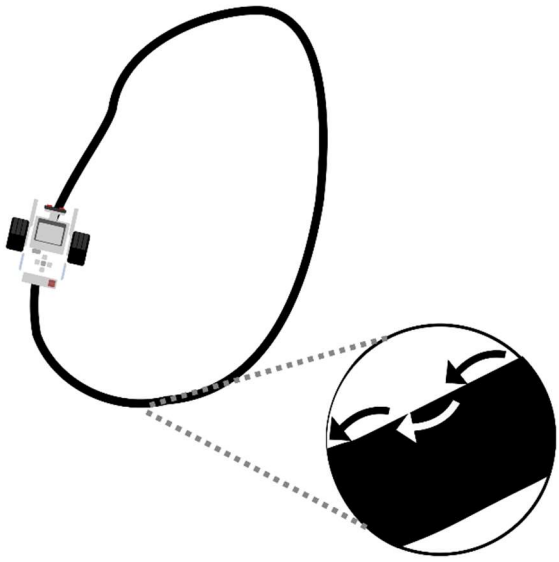

When Started
start drawing with pen on port Auto
set the pen color to red 0 green 0.7 blue 0.7 on port Auto
set the pen width to 2 on port Auto
set speed to 70
repeat while true
do
set dist to ultrasonic distance on port Auto in cm
print dist
if dist < 50
do
move tank with left speed -30 and right speed 30 % for random integer from 360 to 540 degrees
set speed to random integer from 30 to 70
move tank with left speed speed and right speed speed %

Aufgabe 4:


When Started
start drawing with pen on port Auto
set the pen color to red 0 green 0.7 blue 0.7 on port Auto
set the pen width to 2 on port Auto
move tank with left speed 70 and right speed 70 %
repeat while true
do
set dist to ultrasonic distance on port Auto in cm
print dist
if dist < 50
do
move tank with left speed -30 and right speed 30 % for random integer from 360 to 540 degrees
move tank with left speed random integer from 30 to 70 and right speed random integer from 30 to 70 %

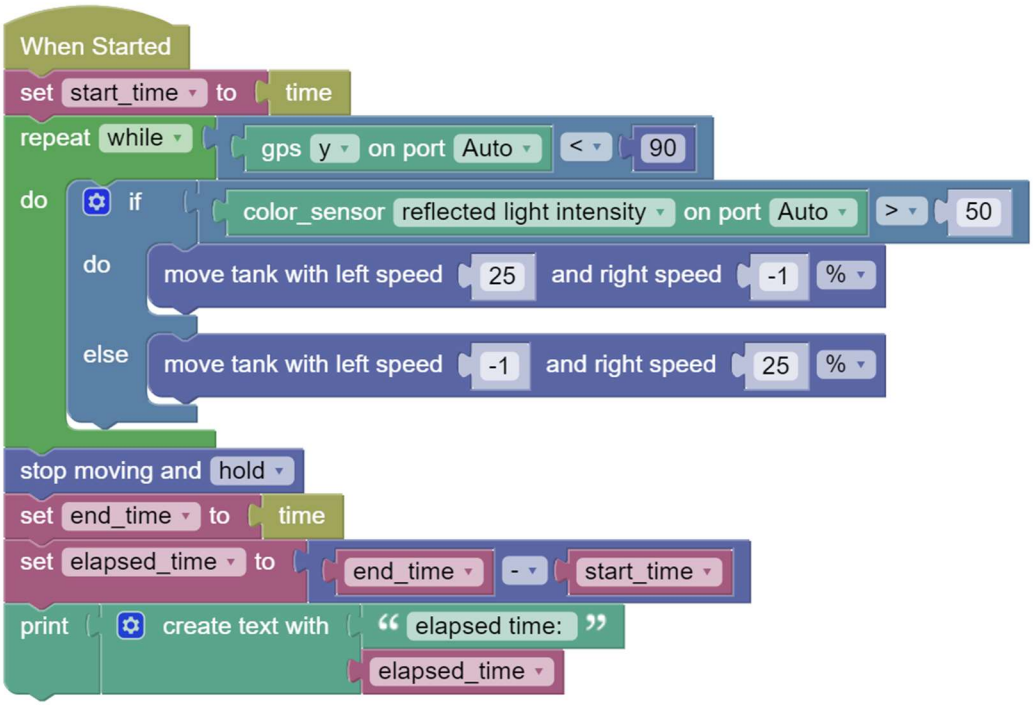


Zurück zum Linienfolger



Starten wir das Programm, dann sehen wir, dass der Linienfolger tatsächlich der Linie folgt, wenn auch nicht sehr flüssig. Interessanterweise ist der Linienfolger auch robust im Bezug auf Vertauschung des **do**- und **else**-Zweiges. Warum? Analysiert man das Verfahren genauer, dann erkennt man, dass der Linienfolger streng genommen nicht wirklich der Linie folgt, sondern der hell-dunkel-Grenze auf einer Seite der Linie, siehe Grafik. Vertauscht man nun do- und **else**-Zweig, dann fährt der Linienfolger einfach auf der anderen Seite der Linie.

Bauen wir zuletzt in den Linienfolger noch eine Zeitmessung ein.



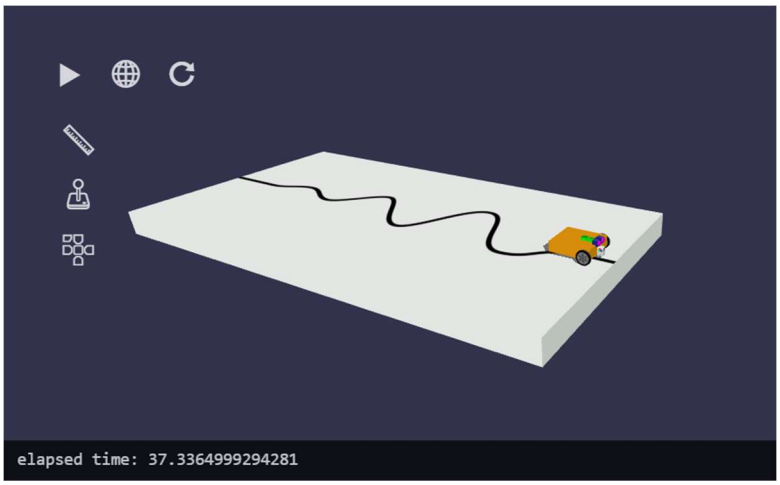
Erklärung

Vor dem Ausführen der **while**-Schleife wird der Variablen **start_time** die aktuelle Systemzeit zugewiesen. Vorher ist eine Variable dieses Namens in der Kategorie Variables anzulegen. Der time-Block wird aus der Kategorie Control geholt. Nach dem Beenden der Schleife wird noch einmal die Systemzeit geholt und in der Variablen **end_time** gespeichert.

Die verstrichene Zeit wird dann einfach mit Endzeit minus Anfangszeit ermittelt. Das funktioniert, weil Zeitwerte einen eigenen Datentyp haben und der „**-**“-Operator eine eigene Variante für Zeitwerte besitzt. Erstmals verwendet wird hier auch der **create-text**-Block (Kategorie **Text**). Mit diesem

Block kann man zwei oder mehr Texte zu einem Text zusammenfügen. Die Konfiguration des Blocks kann wieder über das kleine Zahnrad erfolgen.

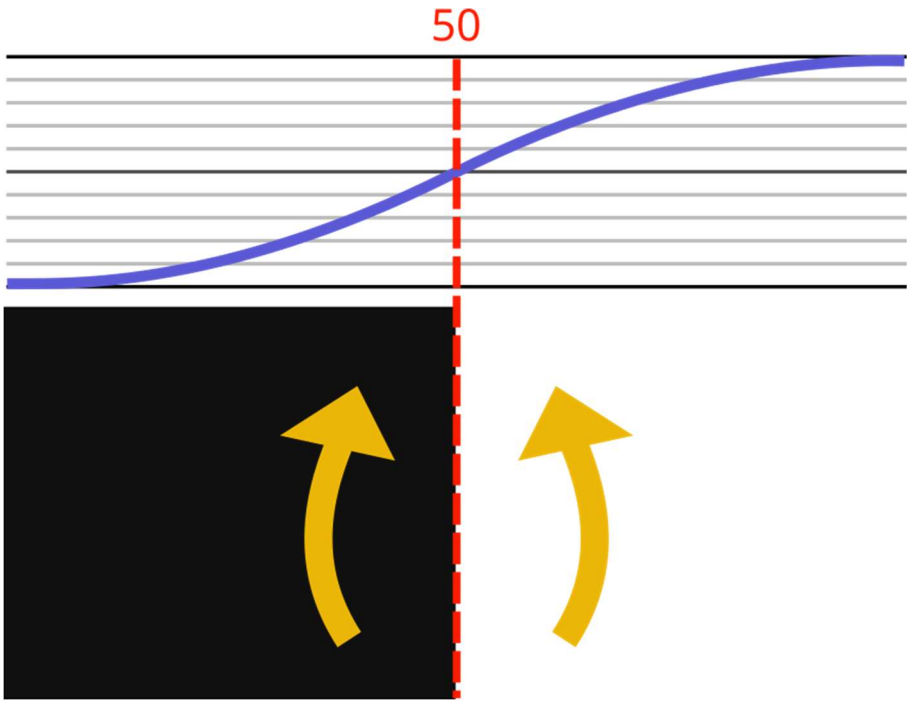
Ergebnis



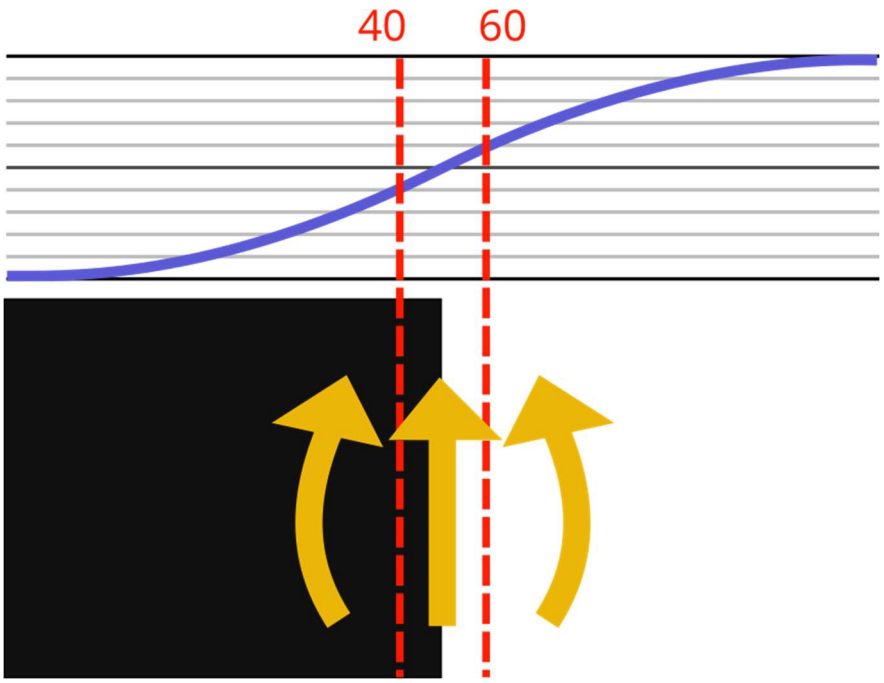
Linienfolger II (verbesserte Version)

Der oben gezeigte Linienfolger hat funktioniert, aber das Fahrzeug bewegte sich nicht sehr flüssig und brauchte fast 40 Sekunden für die kurze Strecke.

Wir überlegen uns deshalb eine Verbesserung. Der einfache Linienfolger beruht darauf, dass man den Helligkeitswert 50% als Schwellwert nimmt und leicht nach rechts steuert, wenn der Wert unterschritten wird und leicht nach links steuert, wenn der Wert von 50% überschritten wird. Die blaue Linie deutet den Helligkeitsverlauf an, wie ihn der Sensor misst.



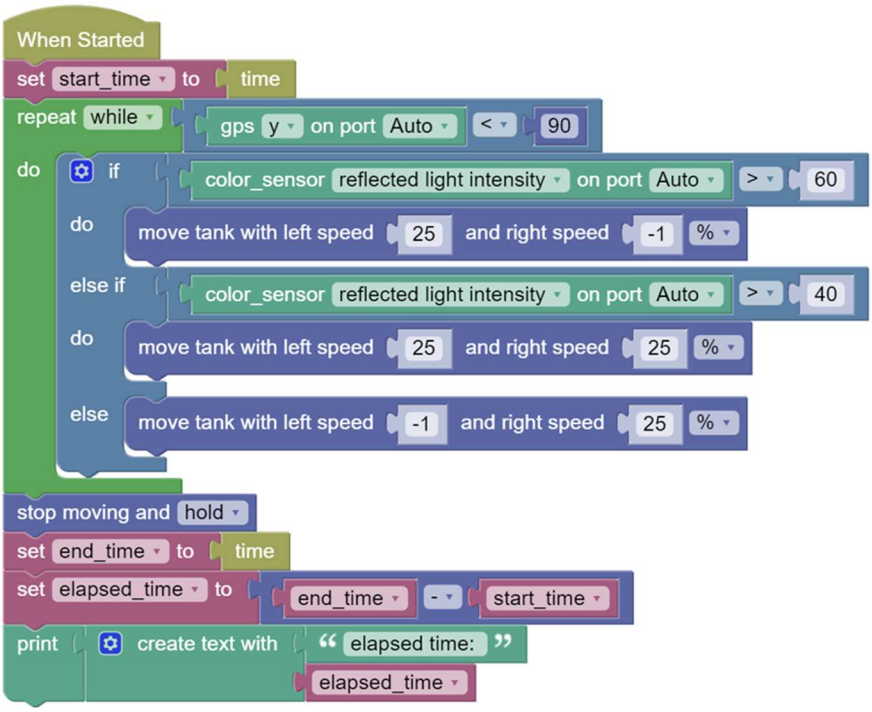
Das Verfahren wollen wir jetzt so wie in der Grafik unten modifizieren. Es kommt jetzt nicht mehr ein Schwellenwert zum Zug, sondern zwei. Ist der Helligkeitswert kleiner als 40% dann wird leicht nach rechts gelenkt, ist der Helligkeitswert größer als 60% dann wird leicht nach links gelenkt. Ist der Helligkeitswert zwischen 40% und 60%, dann wird geradeaus gelenkt.



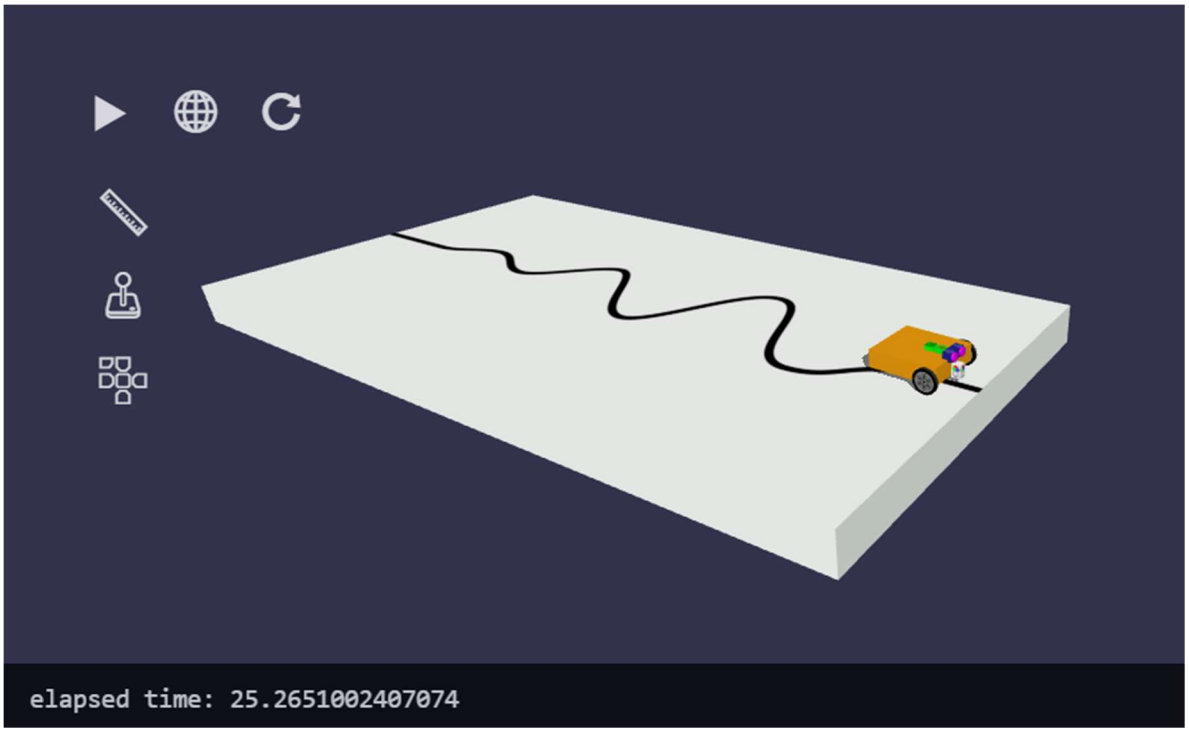
Aufgabe

Kannst du diesen Plan selbst umsetzen? Es sind nur relativ kleine Änderungen am Programm des einfachen Linienfolgers notwendig.

Lösung



Ergebnis

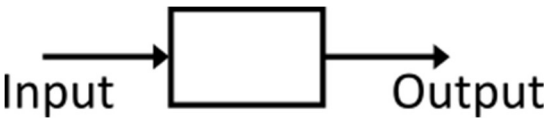


Das Fahrzeug bewegt sich deutlich flüssiger und schneller.

Steuerung und Regelung

Steuerung

Weiter oben haben wir gehört, dass die Steuereinheit Sensordaten empfängt, diese verarbeitet und aus den Ergebnissen Steuerbefehle für die Aktoren erzeugt. Diese Aufgabe führt die Steuereinheit im oben schon erwähnten Sense-Plan-Act (SPA) Zyklus fortlaufend durch. Im einfachen Fall kann man sich das vorstellen wie in der folgenden Abbildung:



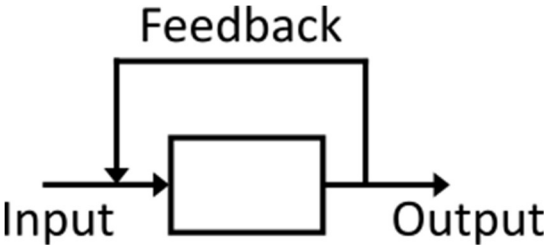
Man spricht auch von einem offenen Wirkungsweg oder einer Steuerkette oder „Open Loop Control“. Beispiele dafür sind:

- Ein stationärer Roboter misst das Umgebungslicht. Am Motor ist ein Propeller montiert. Je höher die gemessene Lichtstärke, desto schneller dreht sich der Propeller.
- Ein autonomes Fahrzeug bewegt sich auf einen vordefinierten Rundkurs. Auf diesem Rundkurs misst das Fahrzeug die Temperatur und erzeugt über einen Lautsprecher einen Ton. Die Frequenz des Tons ist desto höher, je wärmer die gemessene Temperatur ist.
- Eine Bewässerungsanlage schaltet sich zu gewissen Tageszeiten für eine bestimmte Zeit ein, um einen Rasen zu bewässern.
- Am Dach eines Bahnhofs ist ein Helligkeitssensor. Wenn das Licht einen gewissen Schwellenwert unterschreitet, wird das Licht im Warteraum eingeschalten. Wenn das Licht diese Schwelle wieder überschreitet, wird das Licht wieder ausgeschaltet.

Eine ganze Reihe von Aufgaben können mit einer Steuerung gelöst werden.

Regelung

Von einer Regelung oder „**`Feedback-Control`**“ spricht man, wenn Wirkungen der Aktoren Einfluss auf die gemessenen Sensorwerte haben. Das bezeichnet man auch als Rückkopplung.



Man spricht auch von einem geschlossenen Wirkungsweg oder „Closed Loop Control“.

Beispiele:

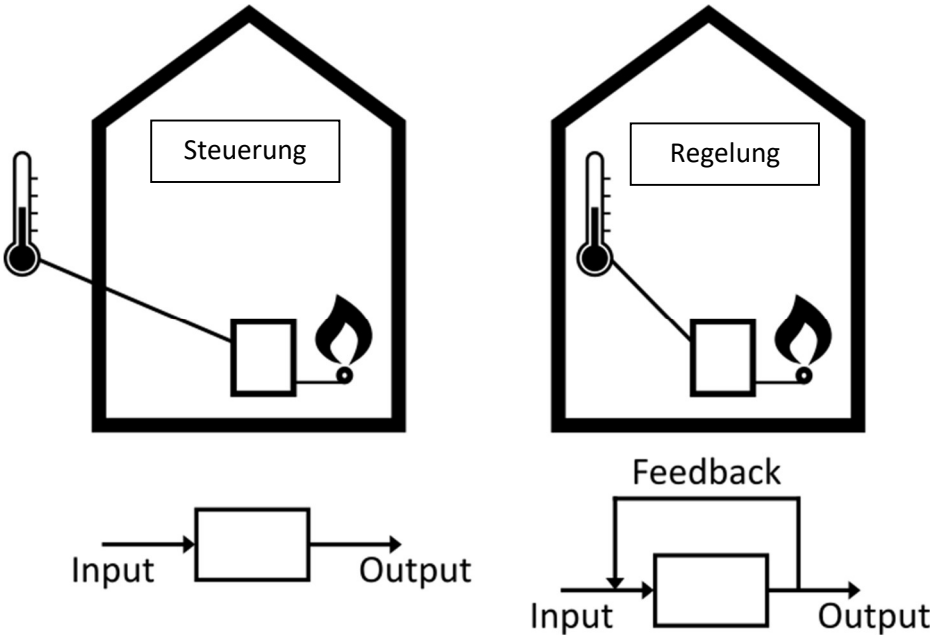
- Eine Bewässerungsanlage schaltet die Bewässerung ein, wenn ein durch einen Sensor gemessener Feuchtigkeitswert einen gewissen Wert unterschreitet.
- Ein Fahrzeug misst mit einem Sensor die Himmelsrichtung seiner Fahrtrichtung. Steuert das Fahrzeug zu weit links von einem Vorgabewert, dann lenkt das Fahrzeug leicht nach rechts und umgekehrt.
- Ein Fahrzeug steuert mit 70% Leistung auf eine Wand zu. Wenn der durch einen Abstandssensor gemessene Wert D 70 cm unterschreitet, dann steuert das Fahrzeug mit D% Leistung auf die Wand zu. Wenn D < 20 ist, stoppt das Fahrzeug abrupt.

Closed Loop Controls sind flexibler, mächtiger und robuster gegen störende Einflüsse. Wenn es zum Beispiel in Strömen regnet, dann schaltet die Bewässerungsanlage nicht ein. Regelungen haben aber auch ein kompliziertes dynamisches Verhalten.

Zur Verdeutlichung des Unterschieds folgt nun ein Beispiel.

Heizungssteuerung

Auf beiden Seiten unten ist ein Haus mit Heizung zu sehen. Ein Temperaturmessfühler misst die Temperatur und schaltet die Heizung nach einer gewissen Vorgabe ein oder aus. Im Fall links handelt es sich um ein Außenthermometer im Fall rechts um ein Raumthermometer.



Trotz der auf den ersten Blick ähnlichen Situation handelt es sich im linken Fall um eine Steuerung im rechten Fall um eine Regelung. Im linken Fall hat die Heizung keinerlei Einfluss auf die gemessene Größe im Gegensatz zum rechten Fall wodurch es zur beschriebenen Rückkopplung (Feedback-Effekt) kommt. Auch das Programm der beiden Systeme wird sich stark unterscheiden.

Programm für die Steuerung	Programm für die Regelung
Wenn die Außentemperatur > 20°, dann schalte die Heizung ab. Wenn die Außentemperatur zwischen 10° und 20° dann schalte die Heizung zu jeder vollen Stunde für 15 min ein. Wenn die Außentemperatur zwischen 0° und 10° ist, dann schalte die Heizung zu jeder Stunde für 30 min ein. …	Wenn die Raumtemperatur unter 18° fällt, dann schalte die Heizung ein. Wenn die Raumtemperatur über 22° steigt, dann schalte die Heizung aus.

Das rechte Programm ist kürzer und kann eventuell die Raumtemperatur besser einstellen als das linke Programm. Es hat aber auch Nachteile:

- Wenn zum Beispiel das Raumthermometer an einer ungünstigen Stelle montiert ist, wo es länger dauert, bis die Wärme vordringt, dann schaltet die Heizung zu spät aus und es wird wärmer als gewünscht.
- Wenn Ein- und Ausschalttemperatur zu nahe beieinander gewählt sind, dann schaltet die Heizung unter Umständen sehr oft ein und läuft nur kurz. Das ist möglicherweise ineffizient und schlecht für die Heizung.
- Regelungen haben unter Umständen ein komplizierteres dynamisches Verhalten als Steuerungen.
- Durch die Rückkoppelung kann es zu ungewollten Aufschaukelungen kommen, ähnlich einer Rückkopplung in der Akustik.

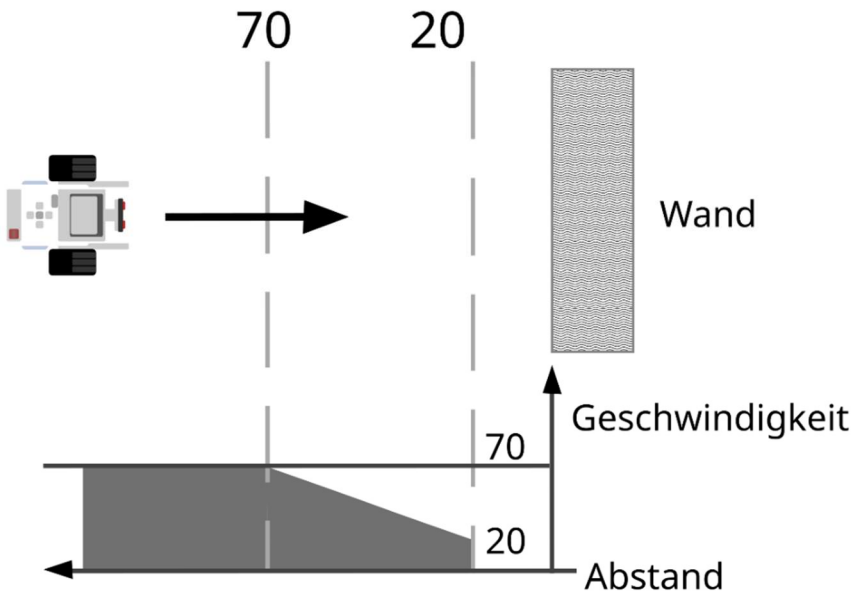
Begriffsklärung

Sehr oft werden die behandelten Begriffe falsch verwendet. Deshalb nochmals zusammenfassend:

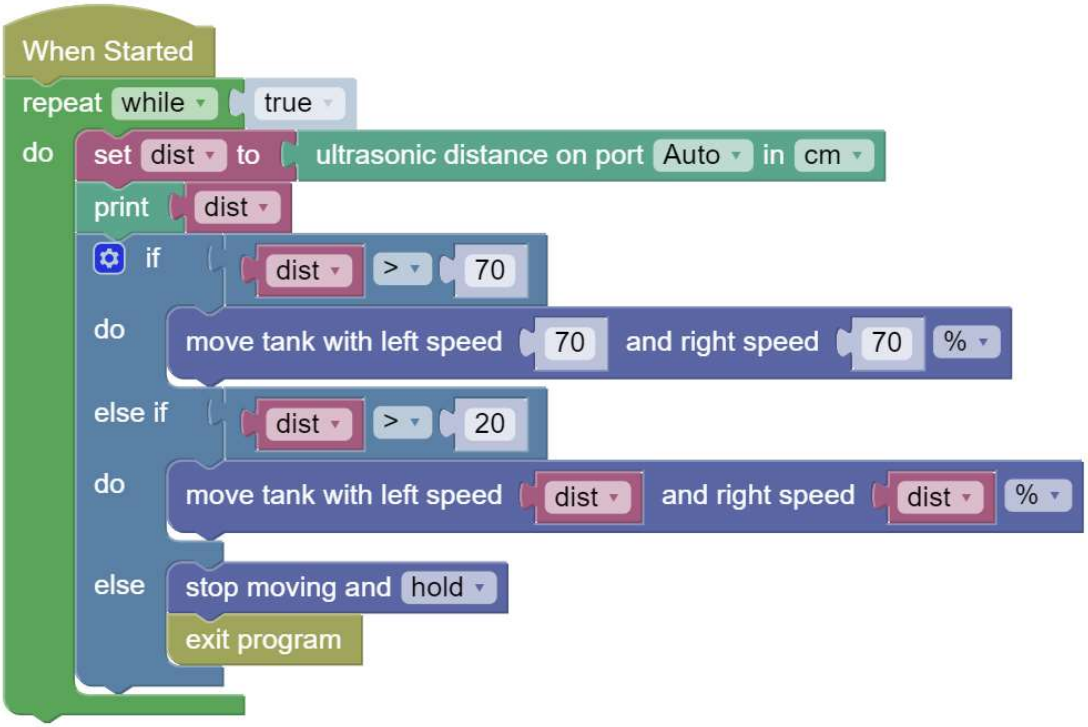
- Control System / Open Loop Control: **Steuerung**
- Feedback Control / Closed Loop Control: **Regelung**
- Feedback: Rückkopplung

Sanftes Bremsen

Setzten wir das Regelungsbeispiel von oben um. Ein Fahrzeug steuert mit 70% Leistung auf eine Wand zu. Wenn der durch einen Abstandssensor gemessene Wert D 70 cm unterschreitet, dann steuert das Fahrzeug mit D% Leistung auf die Wand zu. Wenn D < 20 ist, stoppt das Fahrzeug abrupt.



Lösung

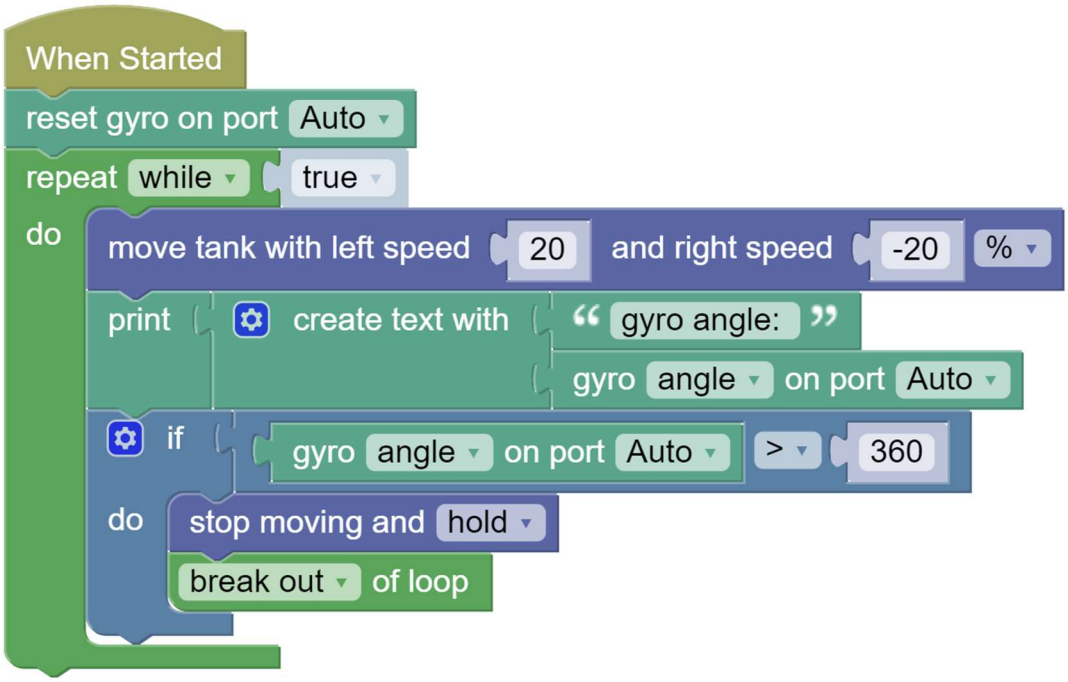


Warum ist dieses Beispiel eine **Regelung** und keine Steuerung?

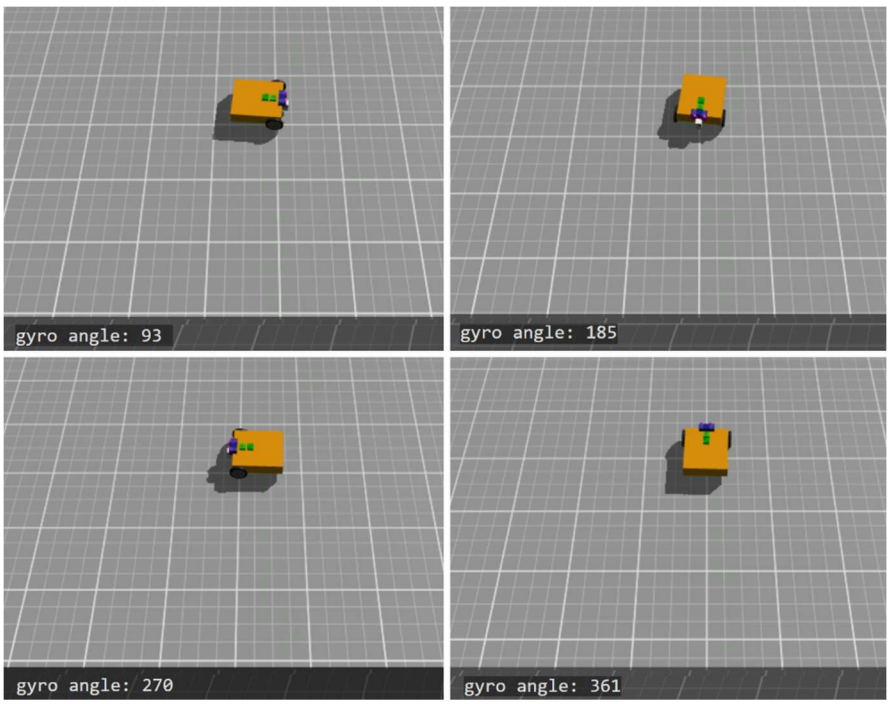
Erinnern wir uns: Für eine Regelung ist eine Rückkopplung (Feedback) gefordert, das heißt, die Wirkungen der Aktoren hat einen Einfluss auf die gemessenen Sensorwerte. Die Aktoren sind in unserem Fall die Motoren, die Sensorwerte sind die gemessenen Entfernungen zur Wand. Hat die Wirkung der Motoren (die Bewegung) Einfluss auf die gemessenen Entfernungen? Ja!

Korrigierte Geradeausfahrt mit dem Gyro-Sensor

Betrachten wir zunächst folgendes Programm. Hier verwenden wir den gyro-Block (Kategorie Sensors) mit der Option „angle“. Abbruchbedingung ist, wenn der gelieferte Wert 360 übersteigt.



Ergebnis



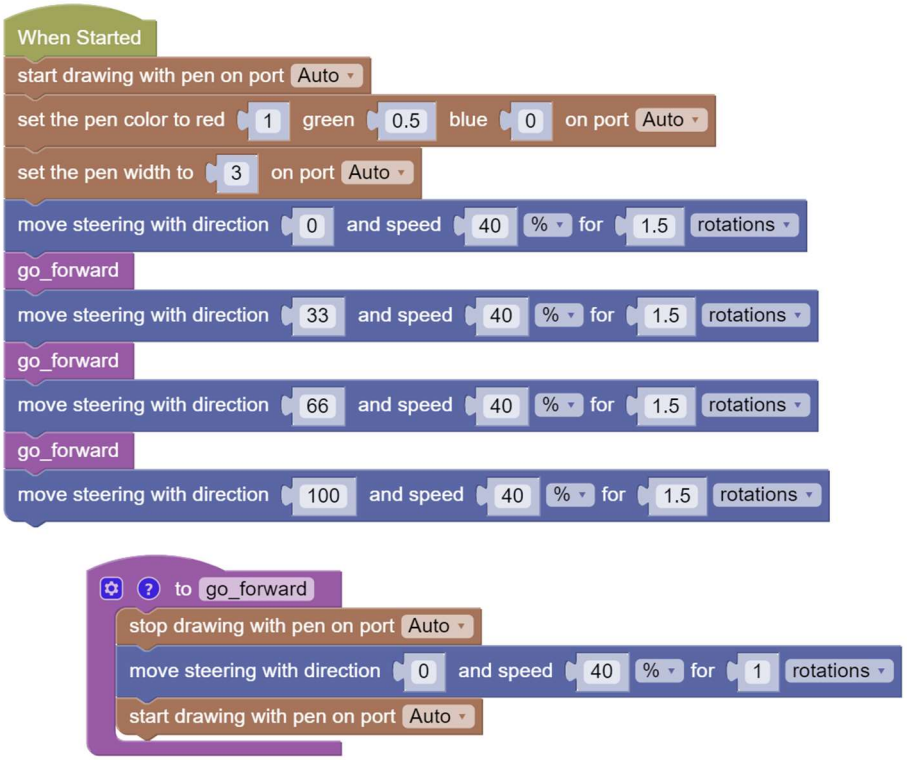
Wie wir sehen, gibt der Sensorwert mit der Option „**angle**“ die Verdrehungsrichtung in Grad an, wobei der Referenzpunkt (0) gleich dem Startpunkt des Fahrzeugs war, genau aus dem Grund, weil dort **reset-gyro** (Kategorie Sensors) aufgerufen wurde.

move-steering-Block

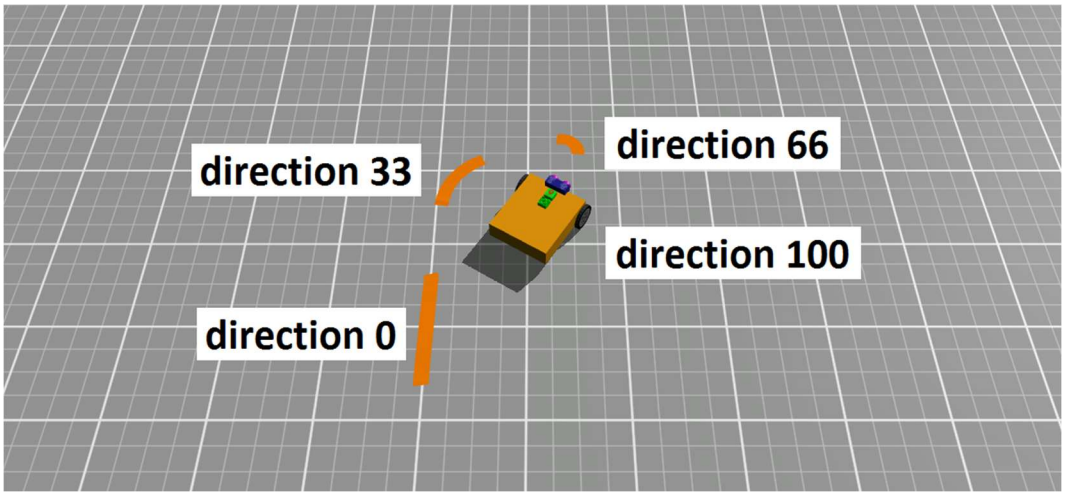
Bis jetzt haben wir diesen Block bewusst nicht verwendet. Unbedingt notwendig ist er nicht, der **move-tank**-Block würde vollkommen ausreichen. Warum gibt es ihn dann? In der Programmierung ist es meist so, dass man Aufgabenstellungen auf viele unterschiedliche Arten lösen kann. Bis zu einem gewissen Teil ist das Geschmacksache und Programmierende haben unterschiedliche Programmierstile, auch wenn sie gleiche Aufgaben bewältigen. Andererseits gilt für den **move-steering**-Block, dass er auf den ersten Blick nicht ganz so anschaulich ist, wie der **move-tank**-Block, aber manche Aufgaben lassen sich mit ihm etwas eleganter lösen als mit dem **move-tank** Block. Dafür ist die korrigierte Geradeausfahrt mit dem Gyro-Sensor ein sehr gutes Beispiel.

Funktionen

Auch für Funktionen (Kategorie `Functions`) gilt, dass sie für eine vollständige Programmiersprache nicht unbedingt erforderlich wären, aber sie sind sehr nützlich, den Programmcode zu strukturieren. Funktionen bekommen einen Namen und Programmcode wird in die Funktion ausgelagert. Wird die Funktion aufgerufen, dann wird der dort befindliche Programmcode ausgeführt und nach Ausführung springt der Programmfluss wieder zurück an die Stelle des Funktionsaufrufes und setzt dort fort. Die hier gezeigte Verwendung ist der einfachste Fall ohne Übergabeparameter und ohne Rückgabewert.



Ergebnis



Erklärung

Der **`direction`**-Parameter des **`move-steering`**-Block hat, wie oben ersichtlich, folgende Bedeutung:

0	Geradeausfahrt
< 50	leichter Bogen nach rechts
> 50	stärkerer Bogen nach rechts
100	Drehung am Stand
negative Werte	wie oben aber links statt rechts

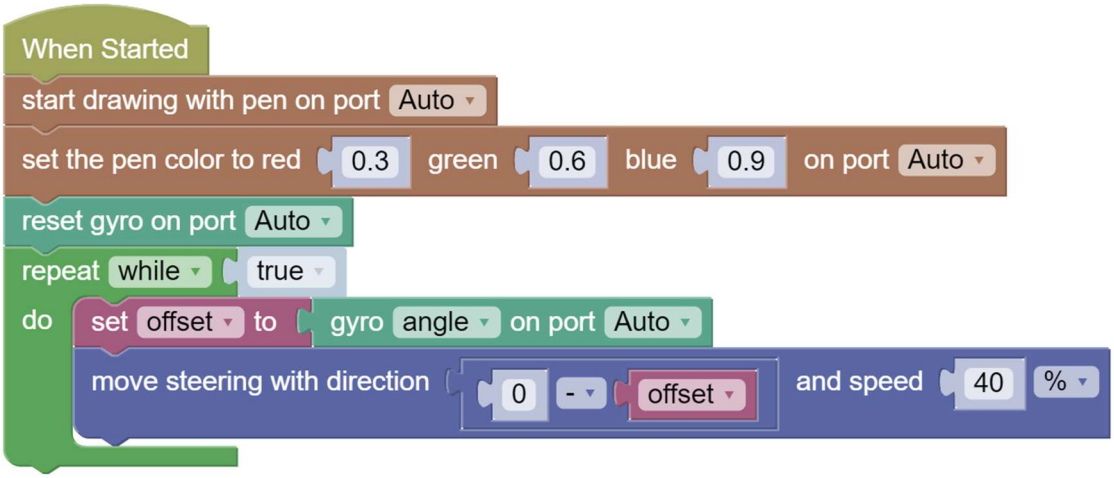
Der **`direction`**-Wert darf zwischen -100 und 100 sein. Andere Werte führen zu einer Fehlermeldung.

Die Funktion wird in unserem Fall also verwendet, um den Stift anzuheben und gerade weiterzufahren. Dadurch entstehen die Unterbrechungen der Linie.

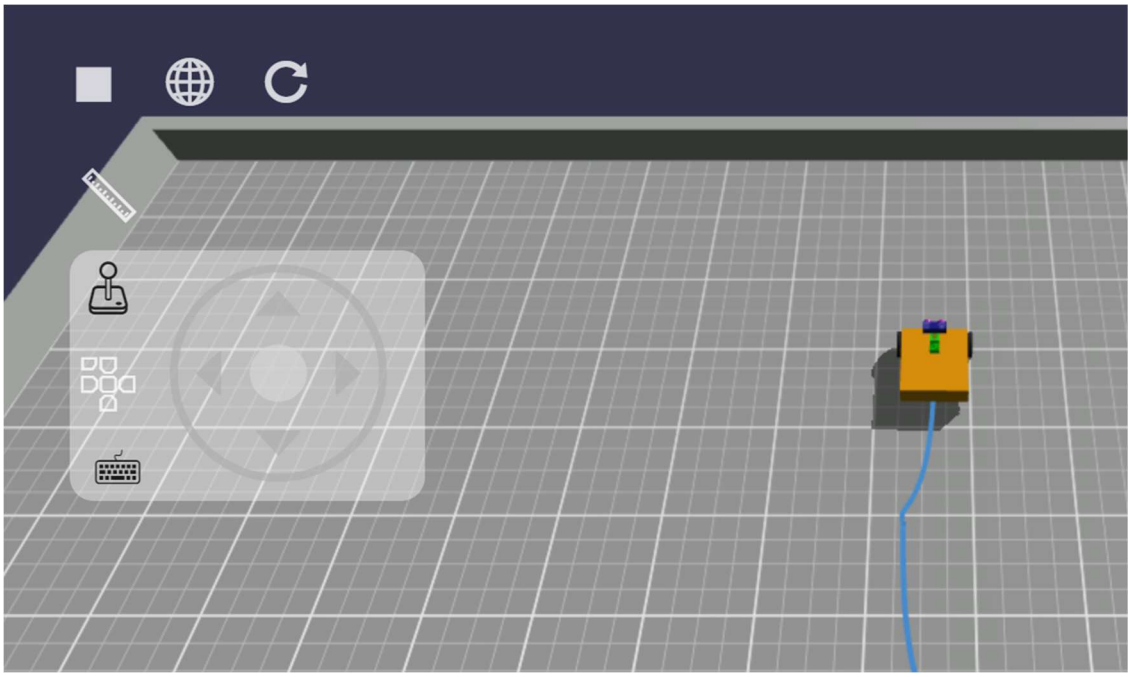
Nach diesen beiden Veranschaulichungen können wir am eigentlich geplanten Projekt der Geradeausfahrt mit dem Gyro-Sensor weiterarbeiten.

Weiter oben haben wir gesagt: Ein wichtiger Bestandteil der Roboterprogrammierung ist es, Ungenauigkeiten und Abweichungen möglichst klein zu halten oder diese entsprechend zu korrigieren.

Die Geradeausfahrt mit dem Gyro-Sensor ist ein sehr gutes Beispiel für diese Strategie der Korrektur.



Der Gyro-Sensor wird durch das Kommando **`reset-gyro`** auf die Zielrichtung eingestellt. Diese ist dann als Richtung 0 festgelegt. Das müsste nicht notwendigerweise so sein aber dadurch vereinfacht sich die Berechnung. Die Anzeige des Gyro-Sensors entspricht dann automatisch der Abweichung von der Zielrichtung (=Offset). Gibt es keine Abweichung (Offset = 0), dann ergibt 0 minus offset gleich 0, das heißt das Fahrzeug fährt gerade. Andernfalls ändert sich die Lenkrichtung im Ausmaß der Abweichung. Das ist genau das gewünschte Verhalten. Auch ob das Vorzeichen richtig ist, kann man sich noch überlegen. Ausprobieren ist in dem Fall auch erlaubt, da es nur 2 Möglichkeiten gibt. Andernfalls könnte man statt 0 minus offset auch offset minus 0 schreiben. Aber eine theoretische Überlegung zeigt, dass das gar nicht notwendig ist, weil ein positives offset zu einer Linkskurve und ein negatives offset zu einer Rechtskurve führen sollte. Die Testfahrt im Bild unten bestätigt, dass der Ausdruck wie im Programm zielführend ist.



Das Fahrzeug fährt los und obwohl man mit dem Steuerknüppel heftige Lenkmanöver nach links und rechts durchführt, kehrt das Fahrzeug von selbst wieder zur Fahrtrichtung 0 zurück. Die Eingriffe mit

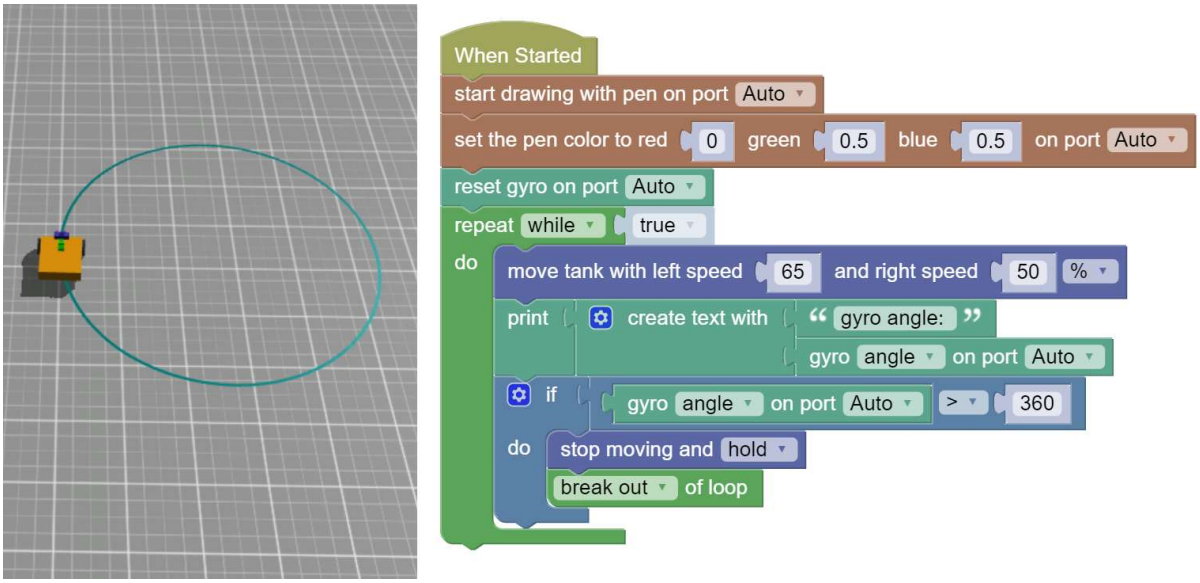
dem Steuerknüppel sollen sehr unebenes oder rutschiges Terrain simulieren. Theoretisch könnte das Fahrzeug über eine drehende Platte fahren und würde Kurs halten, solange sich die Platte nicht zu schnell dreht.

Aufgaben

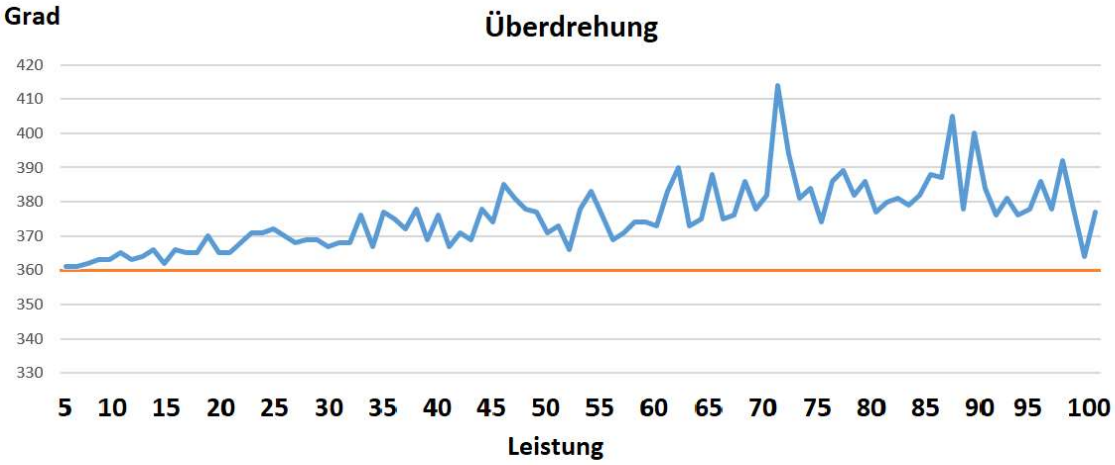
Das Fahrzeug soll einen großen Kreis fahren und mithilfe des Gyro-Sensors erkennen, wann dieser vollendet ist. Dann soll das Fahrzeug anhalten.

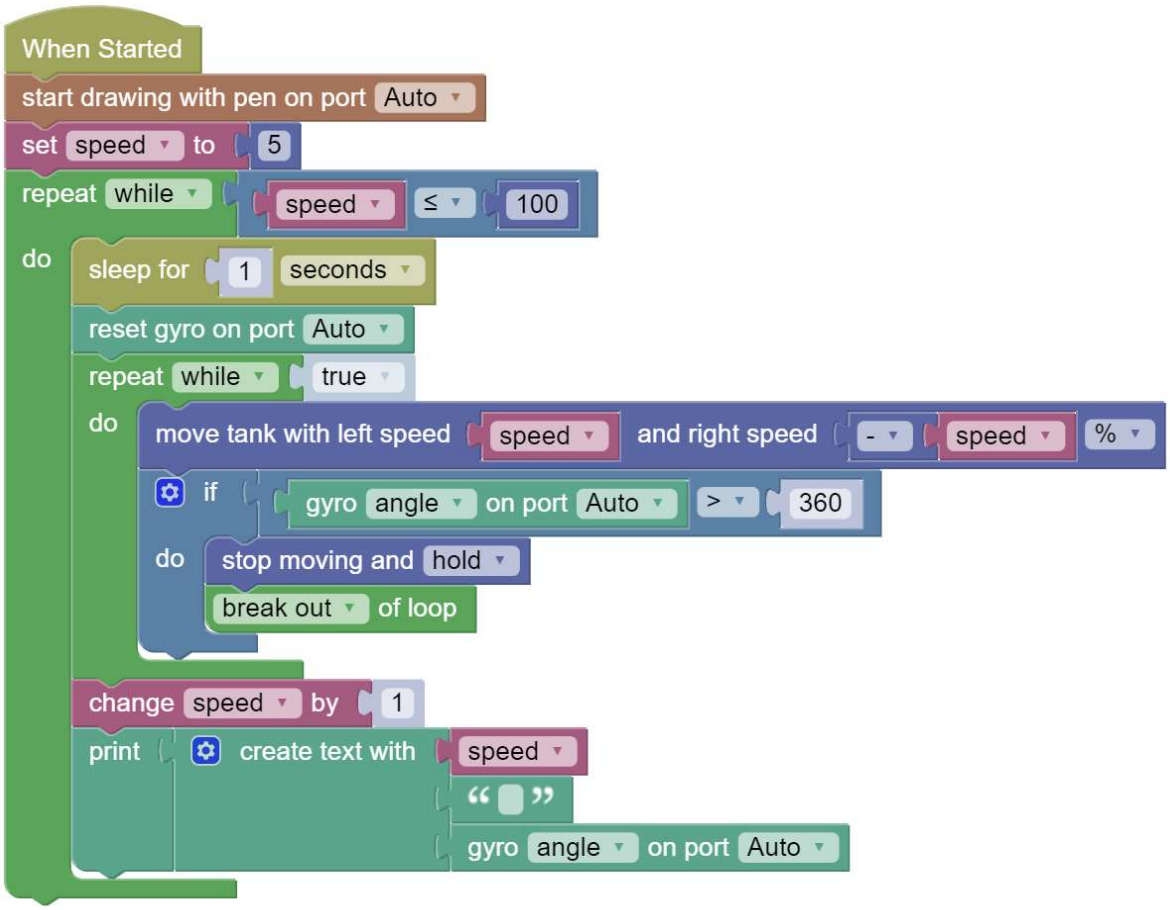
Experimentiere mit dem Programm von oben, welches das Fahrzeug am Stand dreht und den Gyro-Sensor verwendet. Probiere verschiedene Drehgeschwindigkeiten und stelle fest, wie stark sich das auf das Überdrehen auswirkt.

Lösungen



Folgende Auswertung wurde mit nachstehendem Programm erstellt:



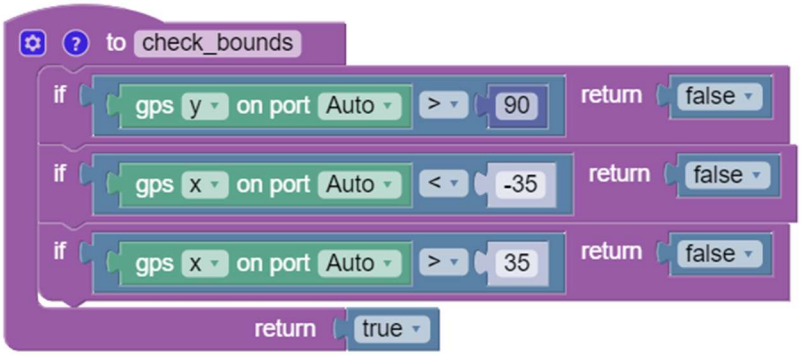


Fazit:
Die durchschnittliche Überdrehung steigt bei steigender Leistung nur leicht an. Die zufälligen Abweichungen werden aber größer.

Linienfolger III, P-Tuning [fortgeschrittener Inhalt]

Der Linienfolger hat uns hier schon in 2 Varianten begleitet, in der einfachen und in der verbesserten Version. Jetzt wollen wir mit einer alternativen, sehr gehobenen Methode, eine noch weiter verbesserte Version erstellen.

Zunächst werden wir eine Funktion kennenlernen, die einen Rückgabewert zurückliefert.

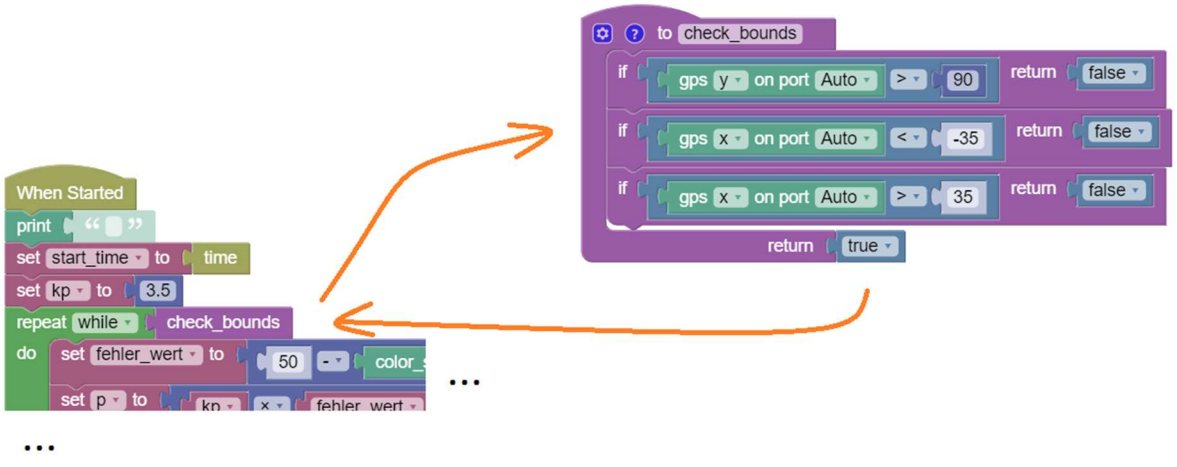


Die Funktion **`check_bounds`** stellt mithilfe des GPS-Sensors fest, ob sich die Position des Fahrzeugs innerhalb eines erlaubten Bereichs befindet. Ist das der Fall, liefert die Funktion **`true`**, ansonsten **`false`**. Diese Funktionen wird im Linienfolger III zum Einsatz kommen.

Funktionen mit Rückgabewert sind eine Erweiterung der bisher bereits verwendeten Funktionen. Mittels des `return`-Blocks wird ein Wert zurückgeliefert. Nach Aufruf des `return`-Blocks wird die Funktion sofort verlassen. Der Aufruf im aufrufenden Programm wird dann mit dem zurückgegebenen Wert ersetzt.

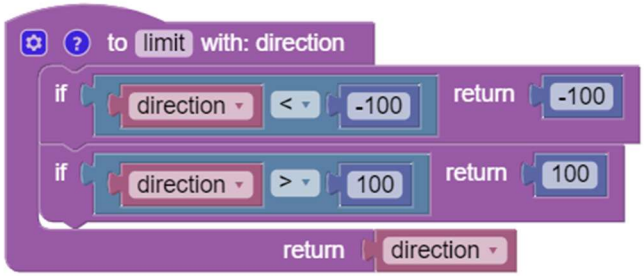
Wenn alle möglichen Rückgabewerte einer Funktion vom Datentyp Boolean sind, dann handelt es sich um einen Spezialfall innerhalb dieser Kategorie. Dann spricht man von einer Booleschen Funktion und wenn diese als Bedingungen verwendet wird, dann genügt ein Aufruf der Form `if check_bounds()`

es muss also nicht `If check_bounds() = true` geschrieben werden.

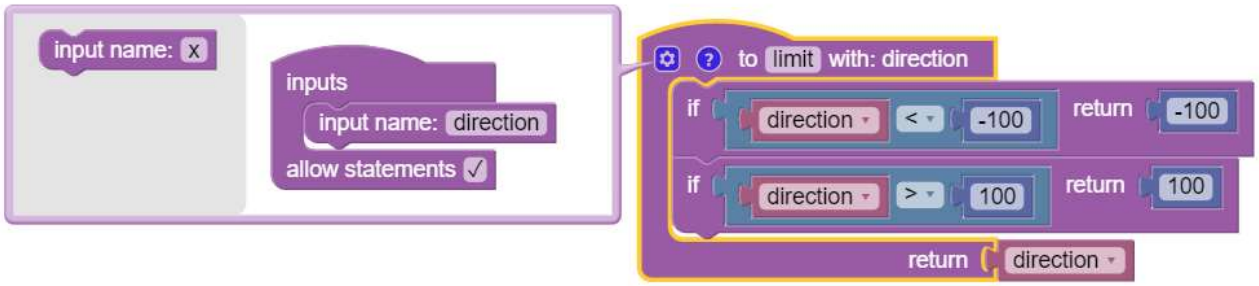


check_bounds wird aufgerufen. Der Aufruf (**`check_bounds`**) im aufrufenden Programm wird dann durch den Rückgabewert (**`true`** oder **`false`**) ersetzt.

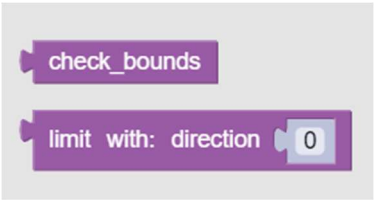
Ganz ähnlich verhält es sich auch bei der zweiten zum Einsatz kommenden Funktion. Zusätzlich kommt aber noch ein Übergabeparameter dazu (**`direction`**).



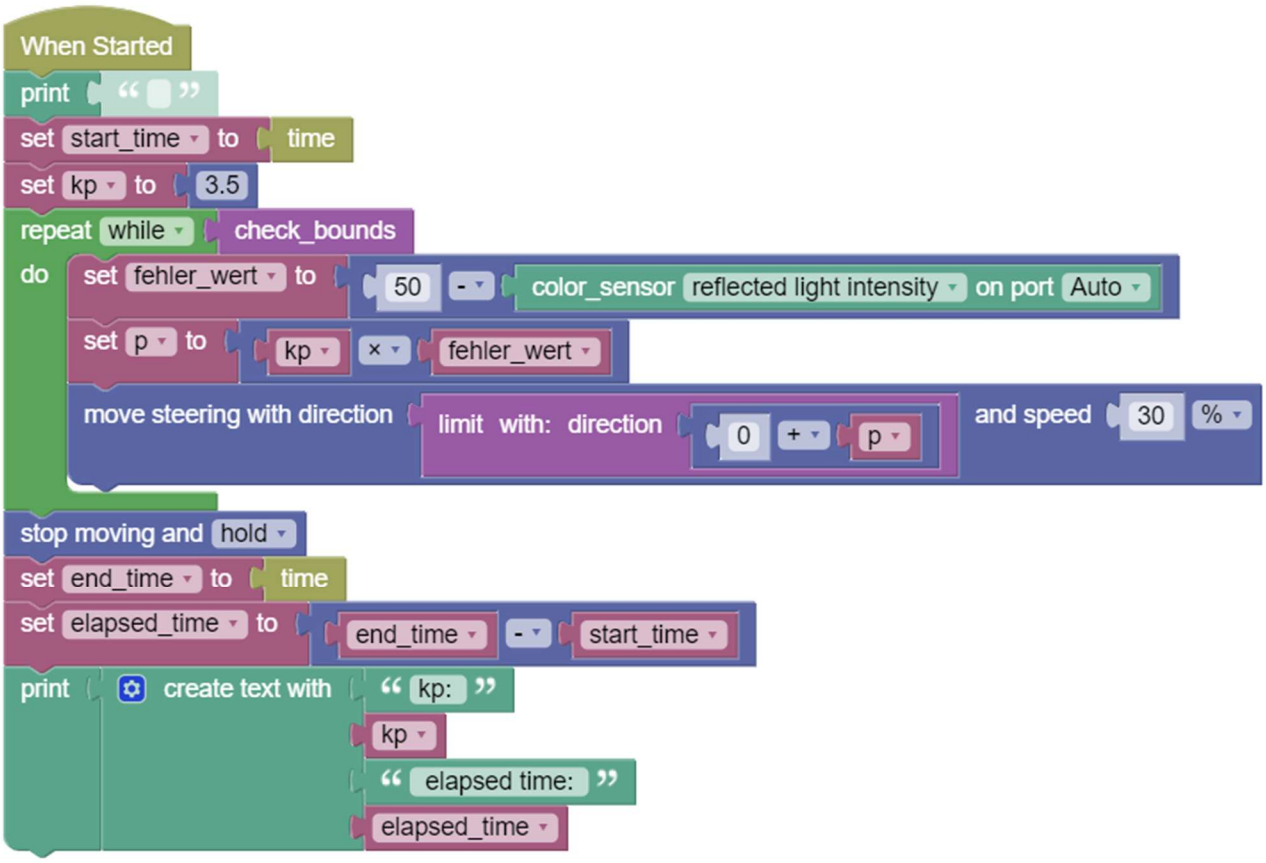
Anzahl und Namen der Übergabeparameter werden wieder über das kleine Zahnrad des Blocks festgelegt.



In der Kategorie *Functions* sind die neuen Blöcke dann so zu sehen:



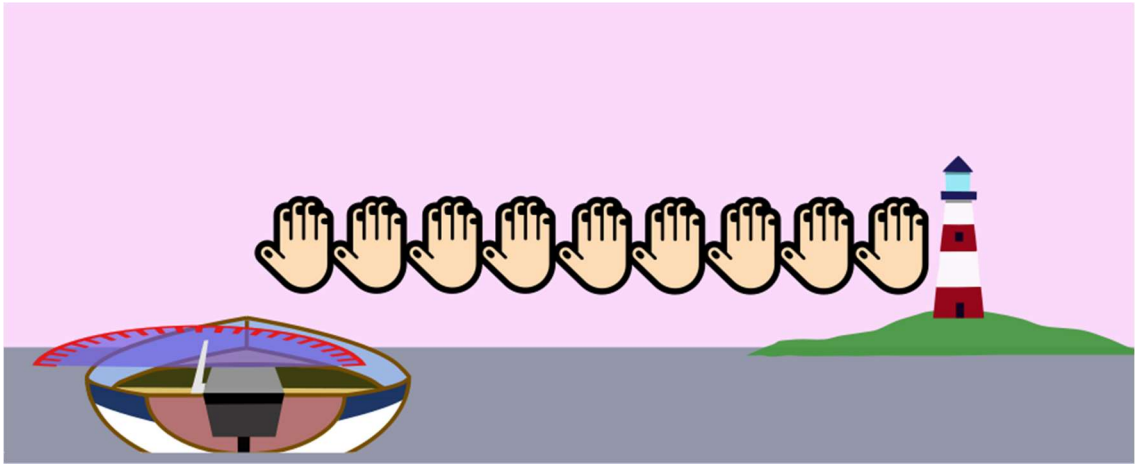
Jetzt sind wir bereit, das fortgeschrittene Linienfolger III Programm zu betrachten.



Das Fahrzeug bewegt sich, solange es im erlaubten Bereich ist (**`check_bounds`**). In jedem Durchlauf wird ein Fehlerwert berechnet. Dieser Fehlerwert ist die Differenz zum Vorgabewert von 50%. Im Idealfall würde ja das Fahrzeug genau auf der Hell/Dunkelgrenze bei 50% fahren. Dieser Fehlerwert wird nun mit einem Faktor Kp multipliziert. Aus dem Produkt ergibt sich der Korrekturwert P. Der Richtungswert wird um diesen Korrekturwert verändert. Hier haben wir einen Fall, wo der **`move-steering`**-Block dem **`move-tank`**-Block klar vorzuziehen ist.

Warum wird der Fehlerwert mit einem Faktor Kp multipliziert?

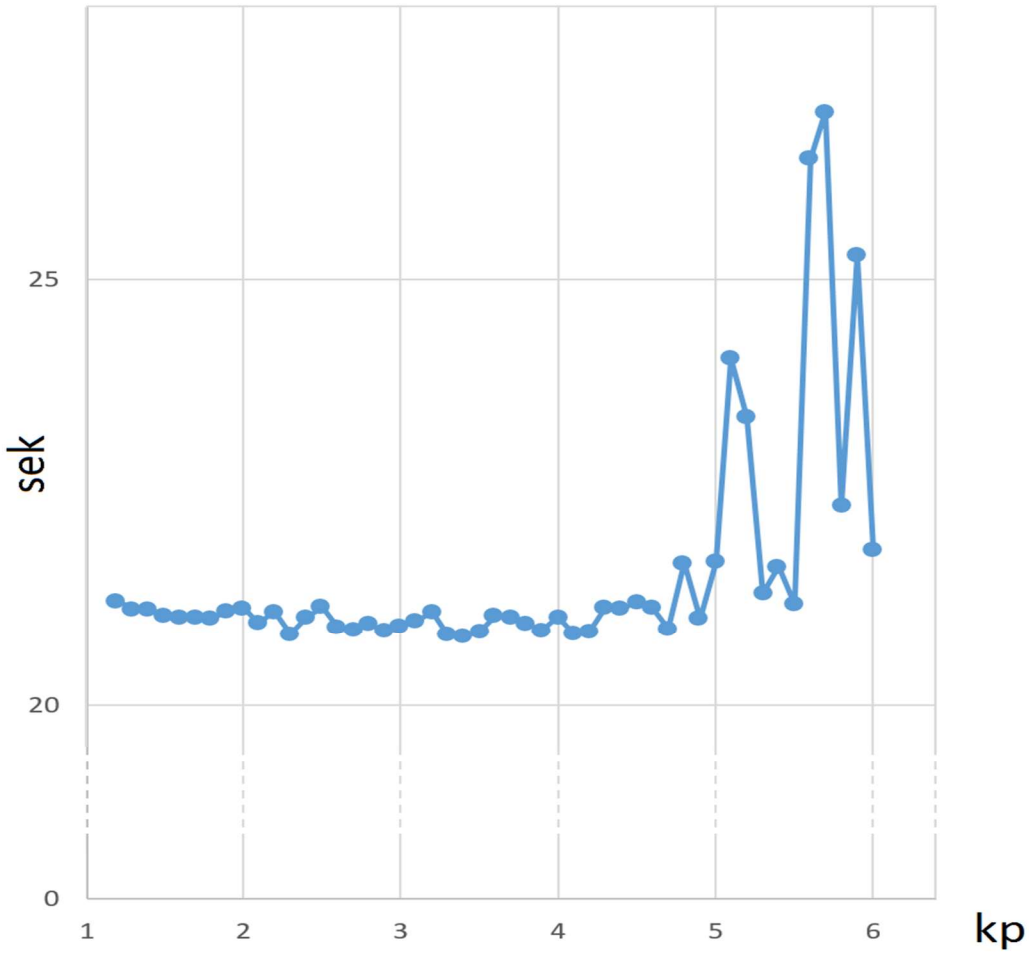
Stell dir vor du steuerst ein Boot zu einem Leuchtturm. Die Abweichung von der Zielrichtung misst du mit Handbreiten deiner ausgestreckten Hand. Der Ruderausschlag wird in Grad gemessen.



Die Idee ist nun die, dass der Ruderausschlag bei einer Abweichung nicht immer gleich groß, sondern **proportional** zur Abweichung ist. Also eine doppelte Abweichung führt zu einem doppelten Ruderausschlag, eine dreifache Abweichung zu einem dreifachen Ruderausschlag und so weiter. Begrenzt ist das jedenfalls auch durch einen maximalen Ruderausschlag. Jetzt wissen wir, warum mit dem Faktor Kp multipliziert wird.

Wie groß wird der Faktor Kp gewählt?

Das herauszufinden ist die Aufgabe des P-Tuning. Es werden verschiedene Werte für Kp durchlaufen und praktisch getestet. So wird ermittelt, welcher Wert für Kp unter den gegebenen Bedingungen ideal scheint und die kürzeste Fahrzeit in Sekunden liefert. Unten ist das Ergebnis so einer Versuchsreihe. Werte von Kp=3.5 scheinen die besten Ergebnisse zu liefern.



P-Tuning

Die Fahrt wurde im Vergleich zum Linienfolger II noch einmal erheblich beschleunigt und das Fahrzeug folgt noch gleichmäßiger der Linie.

Neben dem P-Tuning (Proportional) gibt es auch das I (Integral) und D (Differential) Tuning. Insgesamt auch PID-Tuning genannt. Dieses wird hier nicht benötigt, ist in anderen Fällen aber sehr wertvoll.

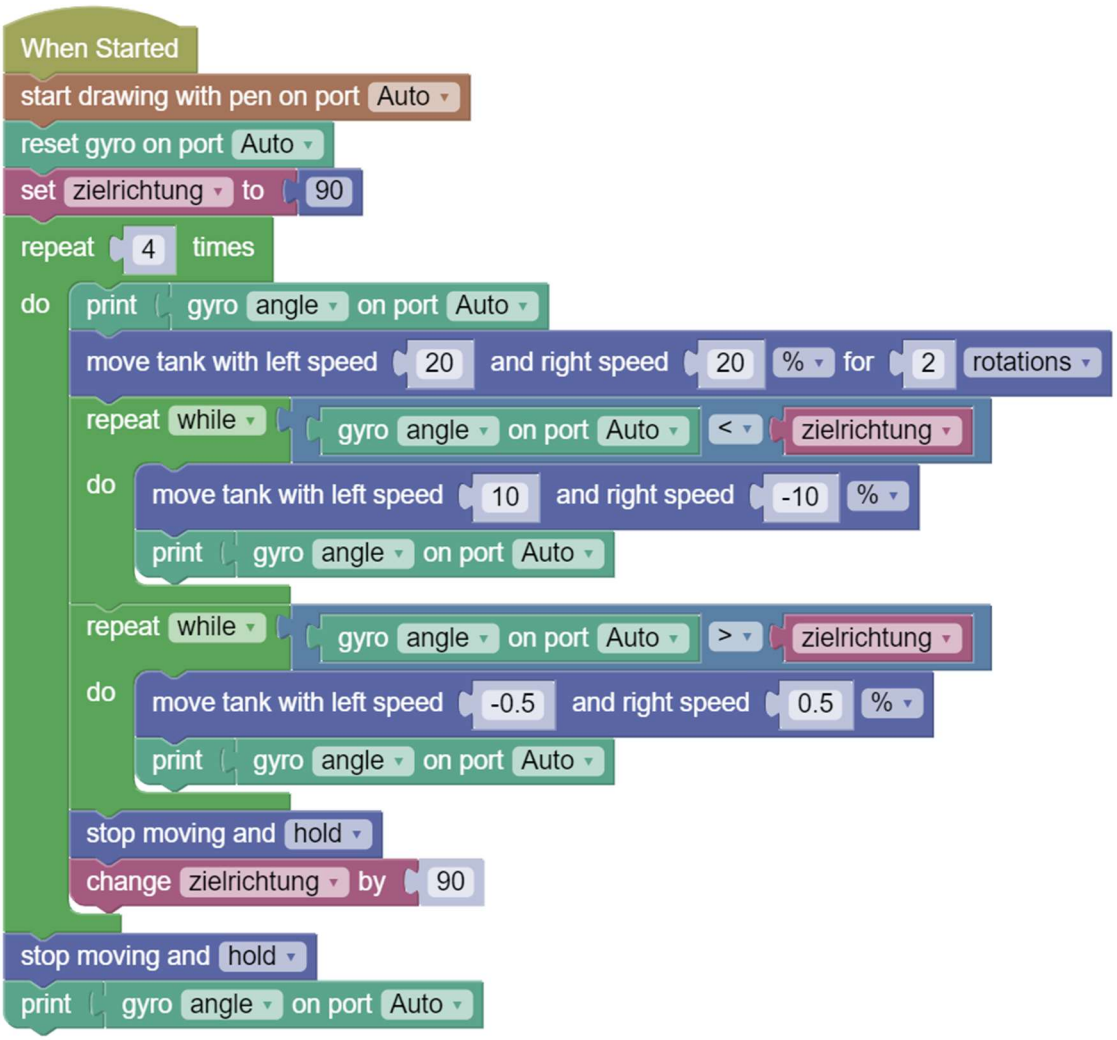
Sensor-Lag und Sensor-Drift

Unter Sensor-Drift versteht man eine verhältnismäßig langsame, ungewollte Änderung von Sensorwerten durch Alterung oder Umgebungsbedingungen. Ein Helligkeitssensor, auf dem sich Staub und Schmutz anlagert, wäre ein Beispiel dafür. Sensor-Drift kann durch Wartung, Reinigung, Neu-Kalibrierung oder Austausch des Sensors behoben werden. Wichtig ist vor allem die zeitgerechte Erkennung des Problems. Das kann teilweise automatisch erfolgen. Neben einem Helligkeitssensor könnte eine anschaltbare Referenzlichtquelle montiert sein. An der zu geringen Änderung des Messwerts bei Einschalten der Referenzlichtquelle kann das Problem erkannt werden.

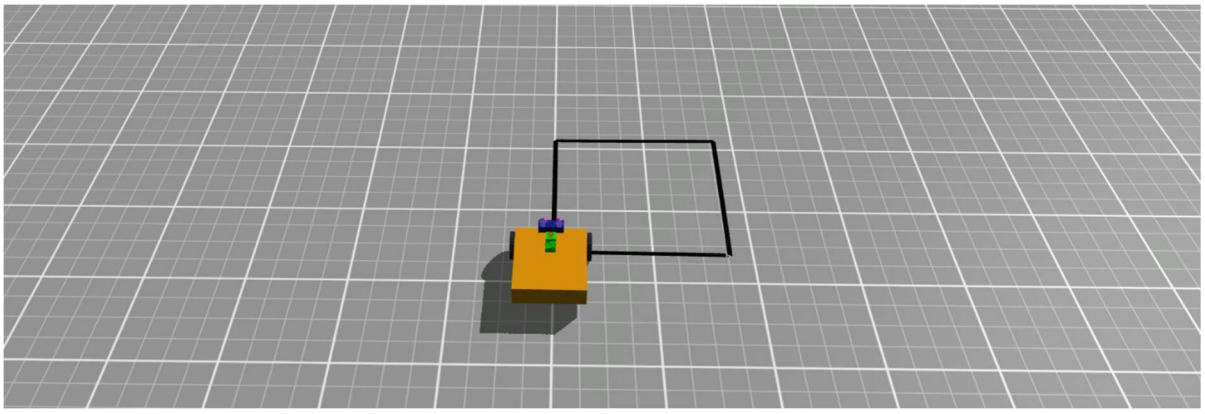
Sensor-Lag bezeichnet den Sachverhalt, dass Sensoren physikalische Größen etwas verzögert messen oder der Sensorwert erst mit einer gewissen Verzögerung eintrifft. Ein Thermometer muss unter Umständen selbst erst die Temperatur der umgebenden Luft annehmen, um den korrekten Wert zu messen. Nach der Messung wird der Wert übertragen, verarbeitet und übermittelt, was zu einer zusätzlichen Verzögerung führt. Die programmierende Person muss sich also bewusst sein, dass ein Sensorwert, der im Programm abgefragt wird, den aktuellen physikalischen Zustand nur verzögert repräsentiert. Auch die oben behandelte Abtastfrequenz spielt hier eine Rolle. In vielen Fällen ist diese Verzögerung aber so gering, dass sie vernachlässigt werden kann.

Quadratfahrt mit Gyrosensor [fortgeschrittener Inhalt]

Eines der ersten Beispiele beschäftigte sich mit der Quadratfahrt. Diese war aber noch relativ ungenau. Jetzt wollen wir wieder die Strategie der Korrektur mit Sensoren verfolgen. Durch Trägheit und Sensor-Lag werden die 4 Drehungen an den Ecken tendenziell wieder etwas zu weit drehen. Manchmal wird als Gegenmaßnahme empfohlen, die Drehung statt bei 90° schon bei 87° oder 88° einzuleiten. Das scheint uns keine gute Idee. Auch ein Reset des Gyro-Sensors, während der Abarbeitung scheint uns nicht zielführend, weil dabei Information verloren geht. Unser Programm dreht zuerst in die Zielrichtung laut Gyrosensor-Daten. Dann wird wesentlich langsamer wieder zurückgedreht, um die Überdrehung zu korrigieren. Dieses Verfahren liefert gute Ergebnisse.



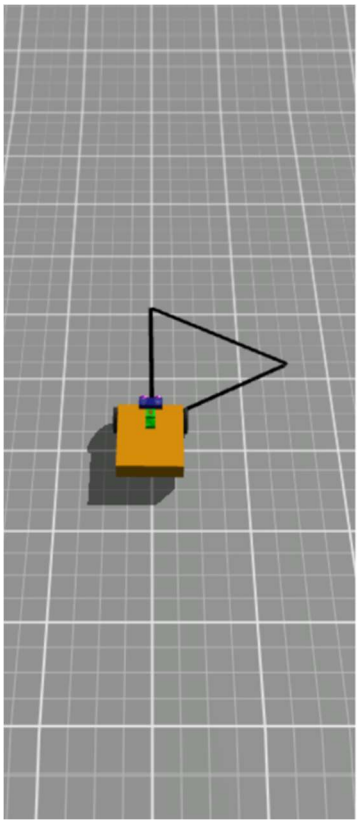
Ergebnis

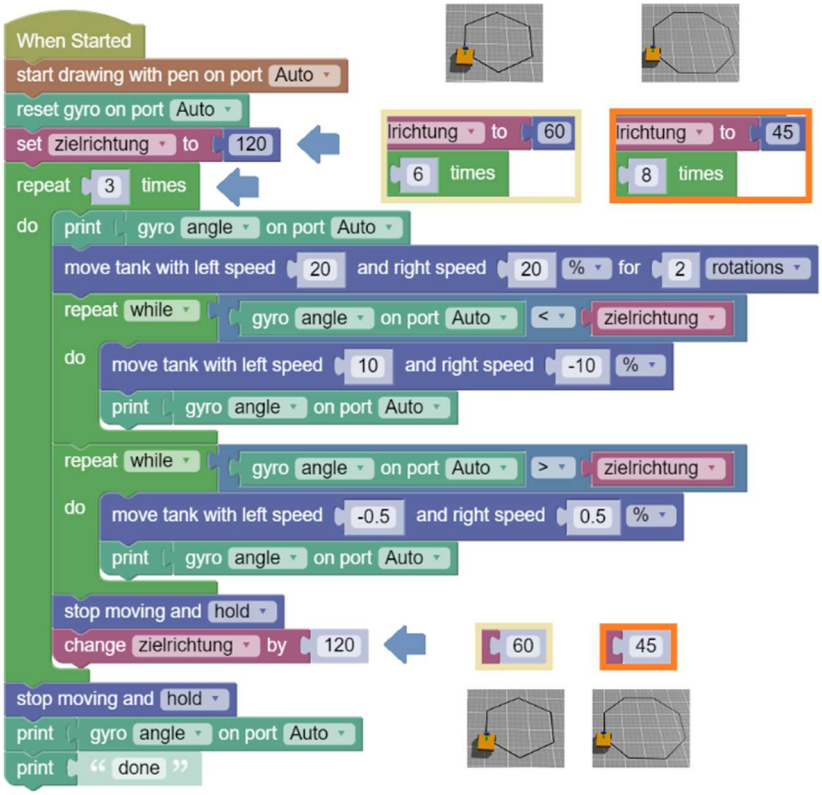


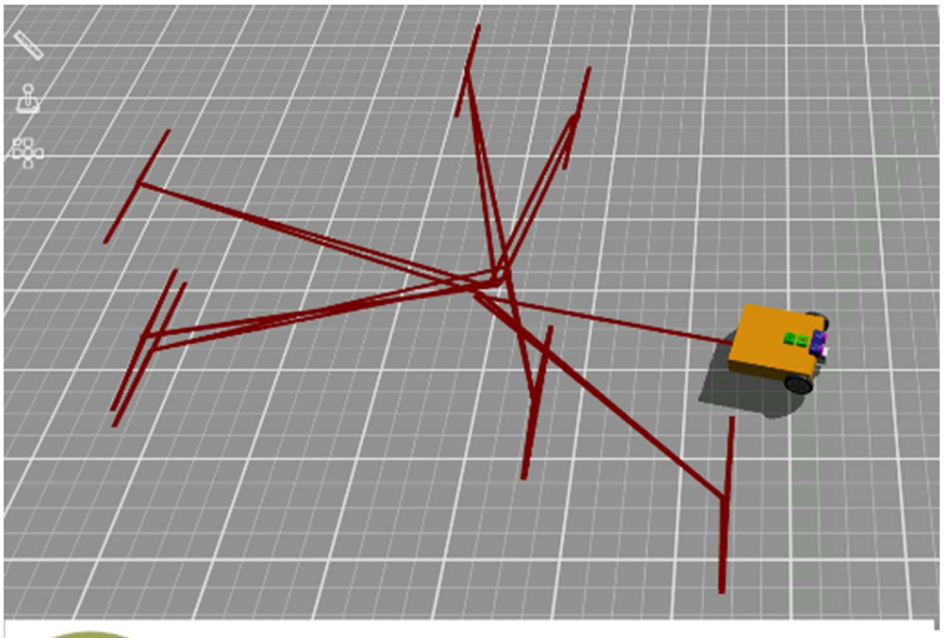
Aufgaben

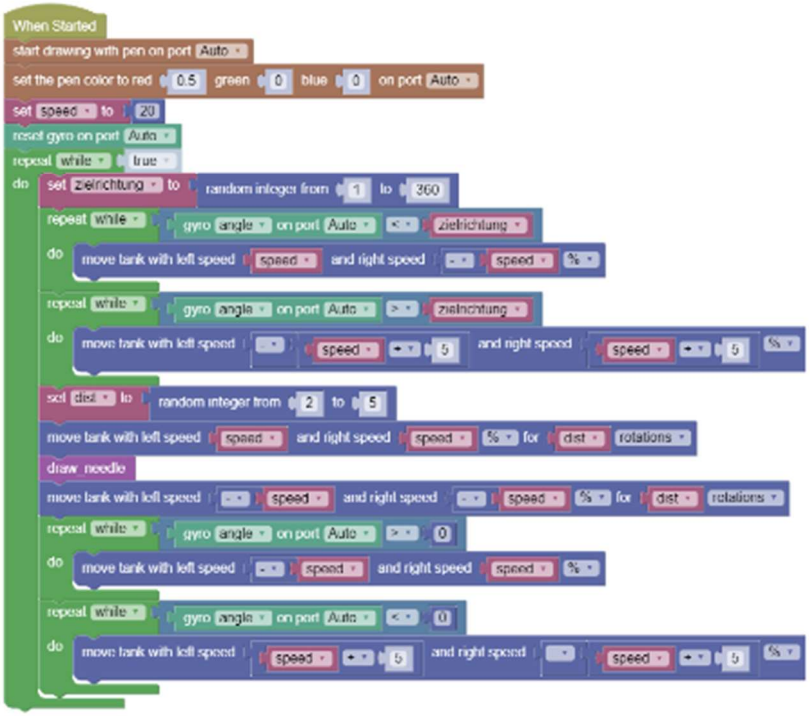
- Adaptiere das Programm oben, sodass mithilfe des Gyro-Sensors ein Dreieck, ein Sechseck oder ein Achteck abgefahren wird.
- Schreibe ein Programm mit Gyro-Sensor-Unterstützung. Verwende den **`random-integer-`** Block. Vor dem Start schaut das Fahrzeug in Himmelsrichtung 1. Drehe am Stand einen zufälligen Winkel zwischen 0 und 360 Grad. Fahre eine zufällige Strecke zwischen 2 und 5 Umdrehungen. Das Fahrzeug schaut jetzt in Himmelsrichtung 2. Drehe dich dann wieder in Himmelsrichtung 1. Fahre 1 Umdrehung nach vor, 2 Umdrehungen zurück und dann wieder 1 Umdrehung nach vor. Drehe dich wieder in Himmelsrichtung 2 und fahre rückwärts zum Ausgangspunkt. Wiederhole das in einer Schleife und mit eingeschaltetem Stift.

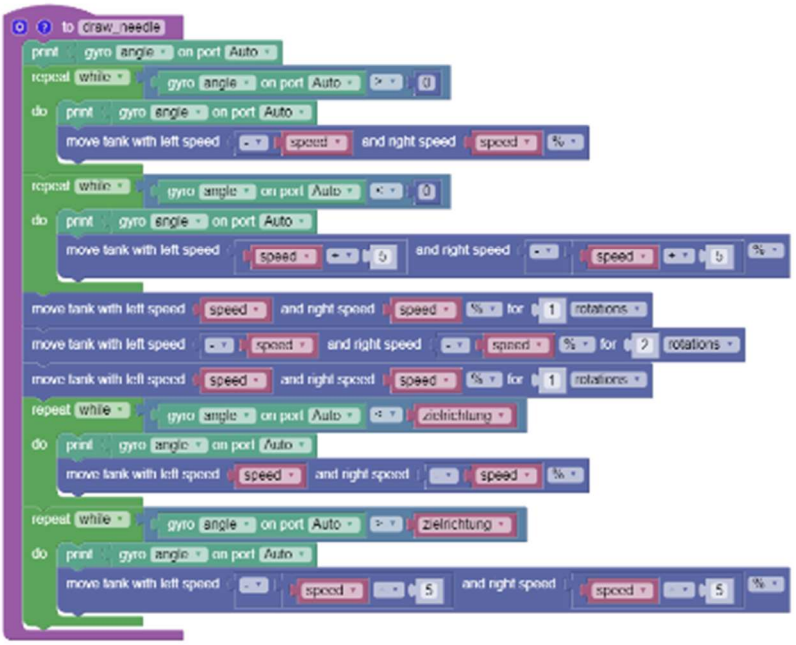
Lösungen




When Started
start drawing with pen on port Auto
reset gyro on port Auto
set zielrichtung to 120
repeat 3 times
do
print gyro angle on port Auto
move tank with left speed 20 and right speed 20 % for 2 rotations
repeat while gyro angle on port Auto < zielrichtung
do
move tank with left speed 10 and right speed -10 %
print gyro angle on port Auto
repeat while gyro angle on port Auto > zielrichtung
do
move tank with left speed -0.5 and right speed 0.5 %
print gyro angle on port Auto
stop moving and hold
change zielrichtung by 120
stop moving and hold
print gyro angle on port Auto
print " done "
lrichtung to 60
6 times
lrichtung to 45
8 times
60
45




When Started
start drawing with pen on port Auto
set the pen color to red 0.5 green 0 blue 0 on port Auto
set speed to 20
reset gyro on port Auto
repeat while true
do
set zielrichtung to random integer from 1 to 360
repeat while gyro angle on port Auto < zielrichtung
do
move tank with left speed speed and right speed - speed %
repeat while gyro angle on port Auto > zielrichtung
do
move tank with left speed - speed - 5 and right speed speed - 5 %
set dist to random integer from 2 to 5
move tank with left speed speed and right speed speed % for dist rotations
draw_needle
move tank with left speed - speed and right speed - speed % for dist rotations
repeat while gyro angle on port Auto > 0
do
move tank with left speed - speed and right speed speed %
repeat while gyro angle on port Auto < 0
do
move tank with left speed speed - 5 and right speed - speed - 5 %


to draw_needle
print gyro angle on port Auto
repeat while gyro angle on port Auto > 0
do
print gyro angle on port Auto
move tank with left speed - speed and right speed speed %
repeat while gyro angle on port Auto < 0
do
print gyro angle on port Auto
move tank with left speed speed - 5 and right speed - speed - 5 %
move tank with left speed speed and right speed speed % for 1 rotations
move tank with left speed - speed and right speed - speed % for 2 rotations
move tank with left speed speed and right speed speed % for 1 rotations
repeat while gyro angle on port Auto < zielrichtung
do
print gyro angle on port Auto
move tank with left speed speed and right speed - speed %
repeat while gyro angle on port Auto > zielrichtung
do
print gyro angle on port Auto
move tank with left speed - speed - 5 and right speed speed - 5 %

Zusammenarbeit

Die bisher besprochenen Programme waren eher klein und überschaubar. Bei größeren Programmen und besonders, wenn mehrere Personen am gleichen Projekt arbeiten, ist eine bessere Planung und eine schrittweise Vorgangsweise zu empfehlen:

- Genaues Analysieren der Aufgabe
- Entwerfen einer Lösung
- Schreiben eines Programms
- Testen und Verbessern eines Programms

Bei der gemeinsamen Arbeit an einem Roboter-Projekt sind Planung, Kommunikation und eine gute Aufgabenzuteilung sehr wichtig.

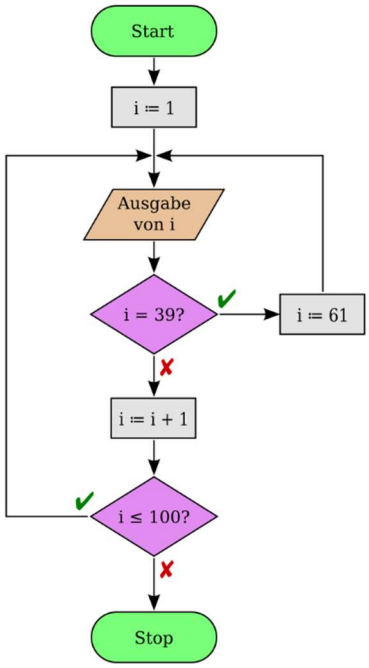
Alle Maßnahmen, die gutes Teamwork fördern sind zu begrüßen, Problemfaktoren sollten frühzeitig erkannt und wenn möglich vermieden werden.

Zum Entwurf oder zur internen Kommunikation, kann man formale Beschreibungsmethoden einsetzen. Hier zwei sehr einfache Beispiele:

Flussdiagramm (Flow Chart)

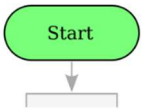
Ein Flussdiagramm stellt einen Algorithmus grafisch dar.

Betrachten wir folgendes Diagramm (die Farben sind optional und nicht Teil des Formalismus):

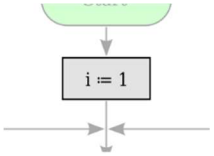


Quelle: abgewandelt von de.wikipedia.org/wiki/Programmablaufplan

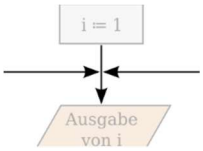
Das Flussdiagramm modelliert eine Zählschleife, die von 1 bis 100 zählt und diese Zahlen ausgibt. Die Zahlen von 40 bis 60 werden aber übersprungen. Das “**:=**” wird für eine Zuweisung verwendet. Erklärung der verwendeten Elemente:



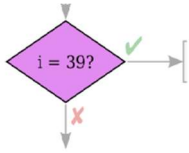
Das ovale Symbol stellt ein Terminalsymbol (Start, Stop, ...) dar.



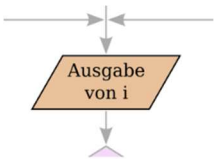
Ein Rechteck steht für eine Operation (=Tätigkeit).



Pfeile oder Linien stellen die Verbindung zum nächstfolgenden Element dar.



Die Raute steht für Verzweigung oder Entscheidungen.



Das Parallelogramm symbolisiert Ein- und Ausgaben.

Pseudocode

Sehr beliebt ist auch Pseudocode, um technologieunabhängig Algorithmen zu formulieren. Obiger Algorithmus würde wie folgt aussehen:

```
i:=1
while i < 101
    print(i)
    if i == 39 then
        i = 61

    i= i+1
```

Aufgaben

Analysiere folgende Programme und beschreibe was passieren wird.

```
reset_gyro()
repeat 2 times
  move_left_right(30,30) for 2 rotations
  while gyro(angle) < 180
     move_left_right(1,-1)
  reset_gyro()


while true
   if color_sensor(reflected_light) > 50%
       move_left_right(25,25)
   else
       stop motors
       break from loop
```

Lösungen

Programm 1:

Der Gyrosensor wird zurückgesetzt. Momentane Orientierung wird als 0-Wert festgelegt.
Der folgende Abschnitt wird 2 mal ausgeführt:
Das Fahrzeug fährt 2 Radumdrehungen vorwärts.
Das Fahrzeug dreht sich im 180° um die eigene Achse.
Der Gyro wird wieder zurückgesetzt.

Zuletzt sollte das Fahrzeug wieder am Ausgangspunkt stehen mit der Ausgangsorientierung.

Programm 2:

Solange der Helligkeitssensor mehr als 50% misst, fährt das Fahrzeug geradeaus mit 25% Leistung. Sobald die gemessene Helligkeit unter 50% fällt, schalten die Motoren ab und das Programm terminiert.

Gefahren durch Roboter und ethische Aspekte

Gesetze von Asimov

Schon 1942 wurden vom Autor Isaac Asimov folgende Gesetze vorgeschlagen:

1. Ein Roboter darf kein menschliches Wesen (wissentlich) verletzen oder durch Untätigkeit (wissentlich) zulassen, dass einem menschlichen Wesen Schaden zugefügt wird.
2. Ein Roboter muss, den ihm von einem Menschen gegebenen Befehlen gehorchen – es sei denn, ein solcher Befehl würde mit Regel eins kollidieren.
3. Ein Roboter muss seine Existenz beschützen, solange dieser Schutz nicht mit Regel eins oder zwei kollidiert.

Schaden abwenden

Oberstes Ziel ist, dass Roboter Menschen keinen Schaden zufügen. Teil des Einsatzes von Robotern ist deshalb eine Risikobeurteilung von potenziellen Gefahren. Eventuell ist der gesamte Arbeitsbereich von Robotern sichtbar abgegrenzt oder gänzlich abgeschirmt gegenüber unbefugtem Betreten. Sensoren können den Zutritt von Personen erkennen und den Roboter abschalten. Zusätzlich kann eine manuelle Notabschaltung vorhanden sein.

Erhöhte Gefahr bei Test, Wartung Reparatur

Die Erfahrung zeigt, dass die meisten Unfälle mit Robotern nicht im Normalbetrieb passieren, sondern zur Entwicklungszeit, während dem Test, beim Service oder der Reparatur. Sicherheitskonzepte sollten das berücksichtigen.

Zunehmende Zusammenarbeit Mensch-Roboter

Die völlige Trennung von Robotern und Menschen könnte aber durch eine zu erwartende höhere Verbreitung von Robotern schwieriger werden. Szenarien, in denen Menschen und Roboter eng zusammenarbeiten, könnten auch häufiger werden. Es wird deshalb intensiv daran gearbeitet, dass Roboter selbst die Gegenwart von Menschen oder Hindernissen frühzeitig und verlässlich erkennen und dann jeglichen Zusammenstoß vermeiden.

Risikoabwägung

Unfälle, so wie sie beim autonomen Fahren passieren können, werden wohl nie ganz vermieden werden können, aber der Einsatz von autonomen Fahrzeugen scheint gerechtfertigt, wenn ein autonomes Fahrzeug im Schnitt viel weniger Unfälle produziert als eine Person, die das Farzeug steuert.

Entscheidungsregeln

Es kann sein, dass Roboter in unvorhergesehene Situationen kommen, in denen sie Entscheidungen treffen müssen. Ein autonomes Fahrzeug könnte einen völligen Bremsausfall haben und das System müsste entscheiden, ob das Fahrzeug gegen eine Betonmauer oder in eine Menschengruppe gelenkt werden soll. Es besteht die Gefahr, dass Hersteller Roboter mit Entscheidungsregeln zugunsten zahlungsfähiger, am Kauf interessierter Personen aber zum Nachteil Unschuldiger ausstatten.

Solche Entscheidungsregeln…

- sollten Gegenstand gesellschaftlicher Diskussion sein
- sollten durch einen breiten Konsens entschieden und juristisch festgeschrieben werden
- dürfen in einem im Betrieb befindlichen Gerät nicht manipuliert werden können (analog zum Beispiel zur Aufhebung der Drosselung eines E-Bikes)
- sollten nach einem Unfall zweifelsfrei und transparent untersucht werden können (Black-Box Prinzip)

Klare Verantwortungsbereiche

Bei jeder Fehlleistung eines Roboters muss klar geregelt sein, wer dafür die Verantwortung trägt. Andernfalls besteht die Gefahr, dass sich Hersteller, Personen im Besitz solcher Systeme und Personen, die solche Systeme entwickeln, gegenseitig die Verantwortung zuschieben oder sich vielleicht gar nicht bewusst sind, welche Verantwortung sie tragen.

Arbeitsmarkt

Roboter könnten unsere gesamte Arbeitswelt revolutionieren. Viele schwere, unangenehme oder ermüdende Tätigkeiten könnten von Robotern übernommen werden. In Kombination mit Methoden der Künstlichen Intelligenz können aber auch sehr gehobene Expertentätigkeiten, wie zum Beispiel chirurgische Eingriffe von Robotern ausgeführt werden. Wie werden Beschäftigte, die von Robotern von ihren Arbeitsplätzen verdrängt werden, in Zukunft ihr Geld verdienen? Ähnliche Befürchtungen bei der Einführung von Computern in den Büroalltag haben sich zwar nicht bewahrheitet, aber ganz unberechtigt scheint die Frage nicht.

ICDL Digitales Lernen Robotik Syllabus 1.0

In diesem Modul werden die Grundprinzipien der Robotik, der Zusammenbau, die Programmierung und die Steuerung eines einfachen Roboters erläutert.

Ziele des Moduls

Personen, die dieses Modul absolvieren, können:

- Schlüsselkonzepte im Zusammenhang mit Robotern und Robotiksystemen verstehen und Beispiele für Roboter identifizieren,
- die wesentlichen Teile eines Roboters und ihre Funktion erkennen, einschließlich Mikrocontroller, Antriebe, Sensoren und Energiequellen,
- die Elemente eines einfachen Steuerungssystems verstehen und ein Steuerungssystem testen,
- grundlegende Programmierkonzepte verstehen sowie ein Programm in einer visuellen Programmiersprache erstellen und ausführen,
- einen Roboter einrichten, Bewegungen implementieren und den Roboter in einer Umgebung steuern.

Kategorie	Fähigkeiten	Referenz	Task Item
1 Grundkonzepte der Robotik	1.1 Roboter und automatisierte Systeme	1.1.1	Definition der Begriffe Roboter und Robotiksysteme
		1.1.2	Verstehen, dass Roboter ferngesteuert, teilautonom oder autonom sein können
		1.1.3	Verstehen, dass Roboter stationär oder mobil sein können
	1.2 Verwendung von Robotern	1.2.1	Häufige Anwendungen von Robotern in verschiedenen Umgebungen kennen, wie: Zuhause, Schule, Produktion, Gesundheitswesen.
		1.2.2	Fortgeschrittene Anwendungen von Robotern kennen, wie: selbstfahrende Autos, robotergestützte Operationen
		1.2.3	Ethische Probleme in der Verwendung von Robotern kennen, wie: Menschen Schaden zufügen
2 Robotik Teile	2.1 Wesentliche Teile und Komponenten	2.1.1	Wesentliche Teile von Robotern erkennen, wie:

			Antrieb, Mikrocontroller, Sensoren, Energiequellen
		2.1.2	Komponenten eines Roboter-Sets kennen, wie: Chassis, Elektronikteile, Kabel, Werkzeuge und Teile für den Zusammenbau
	2.2 Mikrocontroller	2.2.1	Wissen, dass der Mikrocontroller Informationen von Eingabegeräten wie Sensoren sammelt, Programme ausführt und Ausgabegeräte wie LEDs und Audiogeräte steuert
		2.2.2	Wesentliche Anschlüsse von Mikrocontrollern kennen, wie: Stromanschluss, USB, kabellos, Ein- und Ausgang
	2.3 Antriebsysteme	2.3.1	Hauptteile von Aktuatoren kennen, wie: Schalter und Motor
		2.3.2	Verstehen, dass der Aktuator elektrische Energie in mechanische Energie umwandelt, um den Roboter anzutreiben.
	2.4 Sensoren	2.4.1	Verstehen, dass ein Sensor Änderungen in der Umgebung wahrnehmen kann, wie: Lichtstärke, Entfernung, Winkel
		2.4.2	Die Funktion von verschiedenen Sensortypen kennen, wie: Licht, Klang, Gyroskop
	2.5 Fortbewegung und Energiequellen	2.5.1	Teile kennen, welche die Bewegung eines Roboters unterstützen, wie: Arme, Räder
		2.5.2	Energiequellen kennen, wie: Batterien, Sonnenenergie
3 Einfache Steuerungssysteme	3.1 Übersicht der Steuerungssysteme	3.1.1	Elemente eines Steuerungssystems kennen; grundlegende Arten einer Steuerung verstehen, wie: offener

			und geschlossener Regelkreis
		3.1.2	Anbindungen an den Mikrocontroller verstehen, wie: Taster, Strom, Motor, USB-Eingang, kabellose Technologien, Sensoren, Ausgabegeräte
		3.1.3	Verbindungen zum Mikrocontroller in einem Blockdiagramm verstehen
		3.1.4	Ein einfaches Steuerungssystem aufsetzen, z. B mit Energiequellen, Motoren und Sensoren
	3.2 Einfache Steuersysteme testen	3.2.1	Vordefinierte Programme ausführen, um Ausgabewerte bereitzustellen, wie: Lichtstärke, Klang, Entfernung, Winkel.
		3.2.2	Verstehen, dass es zwischen Eingang und Ausgabe der Daten eine Verzögerung gibt
		3.2.3	Verstehen, dass das Verändern von Variablen in einem Programm die Ausgabe beeinflusst
4 Visuelle Programmierung	4.1 Programmiergrundlagen	4.1.1	Definition der Begriffe Programm und Programmiersprache
		4.1.2	Wissen, dass Blöcke grundlegende Elemente in einer visuellen Programmiersprache sind; häufige Blockkategorien kennen, wie: Ereignisse, Steuerung.
		4.1.3	Typische Tätigkeiten bei der Erstellung eines Programms kennen, wie: Analysieren einer Aufgabe, Entwerfen einer Lösung, Schreiben eines Programms, Testen und Verbessern eines Programms

		4.1.4	Grundlegende Elemente eines Programms kennen, wie: Ablauf, Entscheidungen, Schleifen
		4.1.5	Verstehen wie ein Flussdiagramm verwendet werden kann, um die Schritte einer Lösung darzustellen
	4.2 Konstante, Variable	4.2.1	Zwischen den Begriffen Konstante und Variable im Kontext eines Programms unterscheiden
		4.2.2	Neue Variablen erstellen und passende Werte zuweisen
	4.3 Ereignisse, Steuerung	4.3.1	Verwendung eines Ereignisblocks in einem Programm, wie: wenn
		4.3.2	Verwendung eines Steuerungsblocks in einem Programm, wie: warten, warten bis
		4.3.3	Eine Schleife oder endlose Fortsetzung mit Blöcken implementieren, wie: für immer, wiederholen
		4.3.4	Bedingungen mit Blöcken implementieren, wie: wenn, dann, sonst
		4.3.5	Logische Operatoren verwenden, wie: und, nicht, oder
	4.4 Erstellen und Ausführen eines Programms	4.4.1	Einen Plan skizzieren und ein Problem lösen, wie: Steuerung einer Ausgabe, eine Reihe von Aktionen durchführen
		4.4.2	Zeichnen eines Flussdiagramms, um die Schritte einer Lösung abzubilden
		4.4.3	Erstellen eines Programms in einer visuellen Programmiersprache, um ein Problem zu lösen, wie: Steuerung einer

			Ausgabe, eine Reihe von Aktionen durchführen
		4.4.4	Verstehen, dass es mehr als einen Weg gibt ein Programm zu schreiben, um dasselbe Problem zu lösen
		4.4.5	Ausführen eines Programms; Identifizieren und Lösen von Fehlern in einem Programm
5 Arbeiten mit Robotern	5.1 Einrichten	5.1.1	Sicherheitsrichtlinien verstehen und implementieren, wie: sicherer Umgang mit elektronischen Teilen und Werkzeug; Bewusstsein um die eigene Sicherheit und um die anderer
		5.1.2	Zusammenbauen eines Roboters mit dem vorhandenen Werkzeug
	5.2 Implementierung der Roboterbewegung	5.2.1	Implementierung von einfachen Roboterbewegungen, wie: Stopp, Vorwärts- und Rückwärtsbewegung, Drehen
		5.2.2	Verständnis für die Zusammenhänge zwischen Energie, Entfernung, Geschwindigkeit und Zeit in der Roboterbewegung.
		5.2.3	Anwenden von Konzepten wie Energie, Entfernung, Geschwindigkeit und Zeit, um Bewegungen zu steuern: Vorwärts- und Rückwärtsbewegung; verstehen, dass Schwung und Reibung die Bewegung beeinflussen können
		5.2.4	Zusammenhänge von Energie, Rotationsgeschwindigkeit und Winkel der Rotation

			in der Roboterbewegung verstehen
	5.3 Implementierung der Robotersteuerung	5.3.1	Verwendung eines Roboters, um Sensordaten zu sammeln, wie: Entfernung, Klang, Winkel, Licht
		5.3.2	Bauen, Testen und Verbessern eines Programms, um den Roboter mittels eines Sensors zu steuern, wie: Licht, Klang, Gyroskop.
		5.3.3	Die Wichtigkeit des Testens verstehen, um Fehler zu beseitigen
		5.3.4	Verstehen, dass manche Fehler zufällig auftreten, wie: Staub, unbekannte Variablen.
	5.4 Steuerung in einer Umgebung	5.4.1	Navigation eines Roboters in einer Umgebung, um Aufgaben mithilfe verschiedener Funktionalitäten abzuschließen, wie: einer Linie folgen oder ausweichen; einem Objekt / einem Hindernis folgen oder ausweichen; eine Rampe hinauf- oder hinunterfahren
		5.4.2	Navigation eines Roboters in einer Umgebung, um verschiedene Szenarien mithilfe passender Kombinationen von Bewegung und Funktionalitäten abzuschließen
		5.4.3	Wissen, dass Teamwork bei der gemeinsamen Arbeit an einem Roboter wichtig ist; Bedeutung von —Fähigkeiten/Skills kennen, wie: Planung, Kommunikation, Aufgabenzuteilung