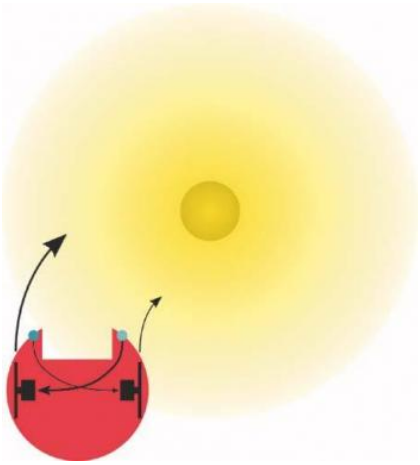
c't-Bots bewältigen komplexe Aufgaben

Hohe Schule

Praxis & Tipps | Praxis .. Uhr
Christoph Grimmer

Roboter haben Höheres im Sinn, als nur vor Wänden, Artgenossen und bodenlosen Abgründen zurückzuschrecken. Als ob sie sich auf die nächsten olympischen Spiele vorbereiten wollten, trainieren c't-Bots jetzt Slalomfahrten - ganz ohne vorher wissen zu müssen, wie der Kurs sich drehen und wenden wird.

Sensoren erlauben es einem Roboter, seine Umgebung wahrzunehmen und unmittelbar auf Reize zu reagieren. So bewegten sich die Stammväter aller mobilen Roboter, die kybernetischen Schildkröten von W. Grey Walter aus den Jahren 1948/49, zielgerichtet auf Lichtquellen zu. Die Steuerung besorgte je ein Analogrechner aus zwei Röhren; seine Leistung reichte allerdings aus, dass die Ur-Roboter ihrem Hunger nach Licht folgen konnten [1]. Komplizierte Algorithmen fallen bei einer derartigen Reise ins Helle nicht an: Wollen c't-Bots das Verhalten ihrer Ahnen nachahmen, müssen sie lediglich ihre Lichtsensoren direkt an den jeweils gegenüber liegenden Motor koppeln (siehe Abbildung).

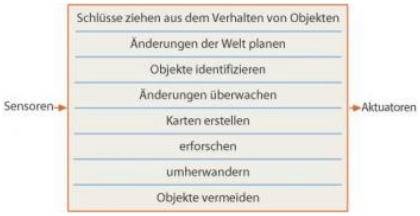


Je mehr Helligkeit ein Lichtsensor misst, desto schneller läuft der daran gekoppelte Motor: Der Bot bewegt sich zum Licht.

Damit der Roboter nicht ungebremst in die Lampe hineinstürmt wie die Motte ins Licht, muss er kurz vor dem drohenden Zusammenprall sein Verhalten ändern: Melden die Abstandssensoren, dass der festgelegte Sicherheitsabstand zum Lampenfuß unterschritten wird, schalten die Motoren auf Schubumkehr. Solche differenzierten Handlungsweisen sind mit festen Verdrahtungen zwischen Sensoren und Aktuatoren nicht mehr so leicht zu erzeugen - flexiblere Steuerungsprinzipien machen hier Bots und Programmierern das Leben leichter.

Der c't-Bot verfügt zu diesem Zweck über eine Liste für mehrere Verhaltensanweisungen, die er ihrer Priorität nach aufruft. Sie bildet eine erste Architektur für den Roboter - alle Routinen, die aus Sensordaten Kommandos für die Aktuatoren berechnen, fügen sich in dieses Schema ein. Eine solche Architektur richtet sich nach der vorhandenen Hardware, vor allem aber nach dem geplanten Einsatz des Roboters: Ein stationärer Industriehelfer erfordert eine andere Programmierung als ein automatischer Rasenmäher.

Neben der bereits in einem vorangegangenen Artikel [2] vorgestellten priorisierten Verhaltensliste eignen sich auch alternative Entwürfe hervorragend, um auf der Grundversion des c't-Bot implementiert zu werden: Einer davon ist die so genannte Subsumptions-Architektur (siehe Kasten "Subsumptions-Architektur" unten). Sie geht genügsam mit dem Speicher um und bringt auf elegante Weise die Daten vieler verschiedener Sensoren unter einen Hut.



Innerhalb der Subsumptions-Architektur bauen abstrakte Verhaltensweisen auf konkreter programmierten Routinen auf (nach [4]).

Dieser Artikel stellt die Grundidee des Subsumptions-Ansatzes vor, anschließend zeigt der zweite Teil am Beispiel einer Slalomfahrt, wie man den c't-Bot dressieren kann, auch komplizierte Dinge zu tun. Der Code dazu ist wie immer im CVS-Archiv auf der Projektseite [3] erhältlich. Da reine Lehren in der täglichen Praxis oft an ihre Grenzen stoßen, haben wir den c't-Bot nicht komplett auf das neue Schema umgestellt, sondern das Grundprinzip der Subsumptions-Architektur mit den Mitteln der vorhandenen Prioritätenliste umgesetzt. Damit bleibt sämtlicher Verhaltenscode, der für das bisherige Framework programmiert wurde, uneingeschränkt verwendbar.

Die Java-Komponente des c't-Sim wurde ebenfalls einigen Änderungen unterzogen. Die aktuelle Version

0.3 verwendet Konstrukte aus dem Java-Standard 5.0 - Voraussetzung für den Betrieb ist also ein JDK, das diese Java-Version unterstützt. In der Entwicklungsumgebung Eclipse muss zusätzlich im Menü Window/Preferences unter Java/Compiler/JDK Compliance die Standardeinstellung 1.4 auf 5.0 umgeschaltet werden.

Nach oben offen

Die Subsumptions-Architektur verfolgt den eleganten Ansatz, komplexes Verhalten auf einem Fundament simpler Verhaltensbausteine zu konstruieren. Die einfachen Benimm-Blöcke (wie „Objekte vermeiden“) werden von abstrakteren Handlungsweisen (wie „erforschen“) eingefasst oder subsumiert - daher der Name des Modells (siehe Abbildung). Was auch immer auf der höheren Ebene geschieht, das Fundament seines Verhaltens, die Bausteine seines guten Benehmens, vergisst der Roboter dadurch nie. Sind die Grundregeln möglichst allgemein gehalten, reagiert er flexibel auf eine sich permanent verändernde Umwelt. Der Roboter handelt in jeder Situation gezielt, aber sicher: Ein plötzlich auftauchendes Hindernis gefährdet den vorwitzigen mechatronischen Forscher keineswegs, wenn die Steuerung fürs Auskundschaften den Reflex mit einschließt, potenziell gefährliche Objekte zu vermeiden.

Die Subsumptions-Architektur fordert, dass ein Roboter über eine Reihe definierter Zustände verfügt; der aktuelle Zustand des Roboters ist stets eindeutig. Wechselt der Bot aufgrund von äußeren Umständen von einem Zustand in den anderen, ändert sich auch sein Verhalten: Beispielsweise bricht er die Erkundung des Geländes ab und kehrt zur Ladestation zurück, sobald sein Akku Spannungsabfall meldet.

Je tiefer die Ebene innerhalb der Architektur ist, auf der eine Verhaltensweise seinen Platz findet, desto direkter knüpft sie an Steuerimpulse und Sensordaten der Roboter-Hardware an. Dabei lohnt es sich, die Befehle etwa an die Motoren ebenfalls in kleine Module zu verpacken. Den Roboter kann man dadurch später erweitern oder umbauen, ohne dass man die Verhaltensweisen (auch Behaviours genannt) auf höherer Ebene komplett neu schreiben muss.

Nach oben hin bleibt der Ansatz prinzipiell offen: Auf den höchsten Ebenen finden theoretisch aufwendige Tätigkeiten wie Kartenzeichnen oder das Schmieden komplizierter Pläne ihren Platz. Verzichtet man darauf, reagiert der Roboter im Rahmen seiner Behaviours unmittelbar auf die Umgebung und fährt in vielen Fällen sehr gut damit. Auch ohne innere Karte kann man einen Roboter gezielt nach Slalomstangen suchen lassen und ihn anschließend auf den Bogenkurs schicken.

Stein auf Stein zum Gipfel

Die neue Version des c't-Bot-Code bringt eine ganze Werkzeugkiste voll flexibel einsetzbarer Module mit. `bot_avoid_harm()` verhindert beispielsweise, dass der Roboter gegen Wände oder in Abgründe fährt, die Routine liest Entfernungs- und Abgrundsensoren aus und weist die Motoren im Notfall in den Rückwärtsgang. Dann liefert die Funktion `True (=1)` zurück, bei freier Fahrt `False (=0)`.

Roboter mit omnidirektionalem Antrieb können einfach einen Schritt zur Seite treten - will der c't-Bot es ihnen gleichtun, muss er dafür eine Reihe von Bewegungen verketten: Er dreht sich zuerst um 90° zur Seite, fährt die gewünschte Distanz geradeaus und rotiert anschließend um -90° zurück. Komplizierte Bewegungen erhält man aber nicht nur durch solche Ketten von Grundbausteinen, sondern auch durch Mischungen aus mehreren Einzelbewegungen. Wer einen Bogen beschreibt, fährt geradeaus und dreht sich dabei gleichzeitig. Ein bisschen mehr Fahrt geradeaus weitet den Bogen, etwas weniger lässt den Roboter eine enge Schleife ziehen oder dreht ihn sogar auf der Stelle. Dadurch konstruiert man aus einer überschaubaren Zahl von Modulen alle gewünschten Bewegungen.

Ganz im Sinn des Subsumptions-Paradigmas abstrahiert beispielsweise `bot_drive()` von der konkreten Steuerung. Für die Kurvenfahrt müssen keine getrennten Befehle für die Drehzahlen beider Motoren gegeben werden, der Parameter `curve` bestimmt mit Werten von -127 (scharf links) über 0 (geradeaus) bis 127 (scharf rechts) die Krümmungsrichtung. Der Parameter stellt bewusst keine Gradangabe dar: Einerseits würde dies eine Präzision vortäuschen, die der Bot in der Praxis nicht einhalten kann, andererseits kosten die dafür nötigen Operationen auf Gleitkommazahlen den Microcontroller viele Ressourcen.

`bot_drive()` schließt die Routine `bot_avoid_harm()`mit ein, die den Roboter vor Schaden bewahrt:

```
void bot_drive(int8 curve, int speed){
  if(bot_avoid_harm()) return;
  if(curve < 0) {
    speedWishLeft = speed*(1.0 + 2.0*(curve/127));
    speedWishRight = speed;
  } else if (curve > 0) {
    speedWishRight = speed*(1.0 - 2.0*(curve/127));
    speedWishLeft = speed;
  } else {
    speedWishLeft = speed, speedWishRight = speed;
  }
}
```

Das Rad auf der Außenseite der Kurve hält die übergebene Geschwindigkeit, das innere wird je nach Krümmungsfaktor `curve` gebremst oder sogar in die Gegenrichtung bewegt - bei Werten von ±127 läuft es sogar mit vollem `speed` gegensinnig und der Bot dreht auf der Stelle.

Ein weiterer Benimm-Block auf der untersten Ebene ist `bot_drive_distance()`, der das Abfahren einer festgelegten Strecke möglich macht - angegeben in Zentimetern, nicht in Rad-Encoder-Schritten, womit wieder von der Hardware abstrahiert wird (konstante Werte wie Radumfang in Millimetern oder die Anzahl der Markierungen auf dem Rad schreibt `bot_local.h` fest). Auch `bot_turn()` funktioniert prinzipiell sehr ähnlich und dreht den Bot um die übergebene Gradzahl nach rechts (bei negativen Zahlen) oder nach links (positive Zahlen).

Im bewährten Rahmen

Die einzelnen Bausteine stehen bereit, doch wie wird die beschriebene Architektur daraus? Das bisherige Verhaltens-Framework des c't-Bot kennt keine expliziten inneren Zustände, wie sie die Subsumptions-Architektur fordert, sondern die zentrale Steuerungsfunktion `bot_behave()` arbeitet eine Liste von einzelnen Verhaltensregeln der Reihe nach ab. Jede Regel muss sich selbst möglichst schnell wieder beenden, damit sie den Prozessor nicht blockiert. Das System kann Verhaltensweisen priorisieren und auch deaktivieren.

Bisher gab es nur zwei Regeltypen: Die einen äußern über die Variablen `speedWishLeft` und `speedWishRight` absolute Wünsche für die Motorendrehzahl und unterbinden damit, dass Routinen niedriger Priorität zum Zug kommen. Regeln, die Ergebnisse von anderen lediglich modifizieren wollen, benutzen `faktorWishLeft` und `faktorWishRight`; deren Inhalt wird mit den Zielvorgaben der folgenden Regeln verrechnet.

In der neuen Version der Bot-Steuerung tauchen Regeln einer dritten Art auf: Sie verzichten darauf, selbst direkt in die Steuerung einzugreifen. Stattdessen erteilen sie Aufträge an andere Regeln. Einerseits soll die aufrufende Regel erst weiterarbeiten, wenn der Auftrag erledigt ist - jeder unvollendete Auftrag entspricht damit einem Zustand nach dem Schema der Subsumptions-Architektur. Andererseits darf der Rest des Programms nicht zum Erliegen kommen, wenn der Bot länger braucht, um den Auftrag zu erledigen. Deshalb arbeitet die Hauptroutine `bot_behave()` alle zehn bis 20 Millisekunden erneut die Liste der Behaviours ab, um regelmäßig die eingehenden Signale der Sensoren zu prüfen und Kommandos an die Motoren auszugeben.

Eine Funktion wie `bot_drive_distance()` dient nur als Bote zwischen Verhaltensregeln. Sie rettet zuerst die Zielvorgaben in globale Variablen, die das folgende Verhalten (`bot_drive_distance_behaviour()`) später ausliest:

```
void bot_drive_distance(Behaviour_t* caller, int8 curve, int speed, int cm){
  ...
  drive_distance_curve = curve;
  drive_distance_speed = speed;
  ...
  drive_distance_target = *encoder - marks_to_drive;
```

Dann teilt die Funktion dem Verhaltens-Framework mit, das aufrufende Verhalten zu deaktivieren:

```
switch_to_behaviour(caller,bot_drive_distance_behaviour,NOOVERRIDE);
```

Dieses darf zwar noch den aktuellen Aufruf beenden und seinen Zustand weiterschalten, kommt dann aber erst mal nicht mehr an die Reihe. Das Framework ignoriert nun jene Funktion, von der `bot_drive_distance()` aufgerufen wurde, so lange, bis `bot_drive_distance_behaviour()` seine Arbeit erledigt hat. Die Verhaltensweise `bot_drive_distance_behaviour()` darf dabei den Prozessor ebenfalls nicht blockieren.

Nach der Initialisierung behält die globale Variable `drive_distance_target` ihren Wert, bis der Roboter das gewünschte Ziel erreicht hat. Bei jedem Durchlauf von `bot_behave()` kommt das `bot_drive_distance_behaviour()` an die Reihe und kann weiterarbeiten. Am Ziel angekommen, deaktiviert es sich selbst und reaktiviert das aufrufende Verhalten:

```
void bot_drive_distance_behaviour(Behaviour_t* data){
  int16 *encoder;
  ...
  to_drive = drive_distance_target - *encoder;
  ...
  if(to_drive <= 0){
    return_from_behaviour(data);
```

Möchte man eigene Regeln schreiben, die Aufträge wie `bot_drive_distance()` verwenden, kann man sich am Beispiel von `bot_do_slalom_behaviour()` orientieren. Wichtig ist, den Zustand des Bot weiter zu schalten, nachdem man eine Botenfunktion aufgerufen hat. Fügt man der eigenen Regel den Aufruf `return_from_behaviour()` hinzu, so kann man sie selbst ebenfalls per Botenfunktion aktivieren und

anschließend zur aufrufenden Funktion zurückkehren.

Olympische Disziplin

Die aktuelle Welt im c't-Sim stellt den c't-Bot vor die Herausforderung, im Slalom durch einen Säulenparcours zu fahren. Jede Säule trägt auf ihrer Spitze eine Lichtquelle, ein eigenes olympisches Feuer. Möchte man den echten Bot aus Aluminium und Halbleitern in seiner realen Umwelt für die olympischen Spiele trainieren, kann man die Rennstrecke beispielsweise aus kleinen Tischlampen aufbauen.

Bevor der Bot mit dem Sport beginnen kann, muss er den Slalomkurs in den Weiten der Welt allerdings erst einmal finden. Erspäht er überhaupt kein Licht, durchsucht er die Gegend nach einem einfachen Muster: Er fährt geradeaus bis zur nächsten Wand und beginnt das Gelände auf konzentrischen Halbkreisen mit wachsendem Radius zu durchkämmen (siehe Abbildung). Früher oder später sollte er auf diese Weise das Licht erblicken, das er ansteuert wie seine Urahnen, die kybernetischen Schildkröten. Identifiziert er mit seinen Distanzsensoren in erleuchteter Umgebung eine Säule, beginnt der Slalomkurs. Hierbei muss der Bot eigentlich die Stangen im Auge behalten, die links und rechts an ihm vorbeiziehen, seine Entfernungsmesser blicken aber strikt nach vorne und bieten nur einen eng begrenzten Blickwinkel. Der c't-Bot löst dieses Dilemma geschickt, indem er sich auf seinem Kurs regelmäßig umschaut und neu orientiert.

Die gesamte beschriebene Verhaltensweise ist im Code unter `bot_olympic_behaviour()` subsumiert:

```
void bot_olympic_behaviour(Behaviour_t *data){
  if(check_for_light()){
    if(bot_avoid_harm() && is_obstacle_ahead(COL_NEAR)){
      bot_do_slalom(data);
    } else bot_goto_light();
  } else bot_explore(data,check_for_light);
}
```

Die Botenfunktionen fürs Erforschen (`bot_explore()`) und den Slalomkurs (`bot_do_slalom()`) aktivieren die komplexen Verhaltsmodule `bot_explore_behaviour()` beziehungsweise `bot_do_slalom_behaviour()`. Diese greifen auf die oben dargestellten Grundbausteine zurück: Die Suche nach Licht etwa besteht aus diversen Drehungen und Kurvenfahrten, für die der c't-Bot einheitlich die Funktion `bot_drive()` verwendet - lediglich der Parameter curve variiert, wodurch die Bögen allmählich raumgreifender geraten. `bot_explore_behaviour()` unterscheidet insgesamt acht Zustände, denn auch die Behaviours auf höherer Ebene dürfen den Prozessor nur kurz beanspruchen.

Solange der Bot noch keinem Hindernis ausweichen musste, fährt er forsch geradeaus. Stellt sich ihm etwas in den Weg, wechselt er seine Taktik: Er dreht sich so lange nach rechts, bis sein linker Entfernungssensor die Wand langsam aus dem Blick verliert. Anschließend fährt er 15 Zentimeter weit parallel zu der Wand an seiner Linken. Danach wechselt der Bot wieder den Zustand und dreht sich um 85 Grad nach rechts (`bot_turn(-85)`). Durch den leicht spitzen Winkel fährt der Bot danach einen etwas größeren Kreis. Nach jedem Aufruf dieses Verhaltens vor einem neuen Bogen wechselt der Parameter `curve` sein Vorzeichen und wird etwas kleiner, was die Kurve weiter öffnet:

```
case EXPLORATION_STATE_DRIVE_ARC:
  if(curve == 0){
    curve = 25;
    running_curve = True;
  } else if (running_curve == False){
    curve *= -0.9;
    running_curve = True;
  }
  ...
  bot_drive(curve, BOT_SPEED_MAX);
```

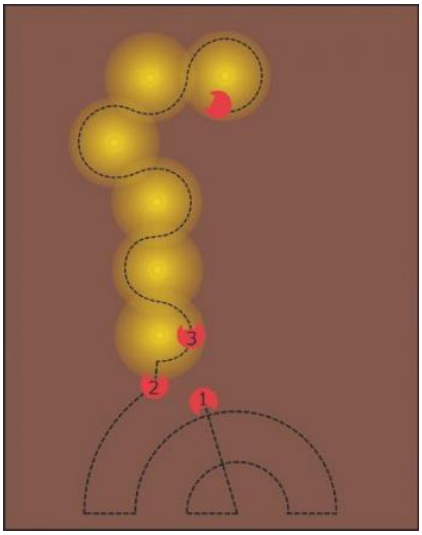
Brüder, zur Sonne

Wurde der Roboter bei seiner Expedition im Dunkeln fündig, lässt ihn `bot_goto_light()` auf die entdeckte Lichtquelle zufahren. Dabei bestimmt der Helligkeitsunterschied (Gradient) zwischen den beiden Lichtsensoren den Krümmungsparameter `curve`:

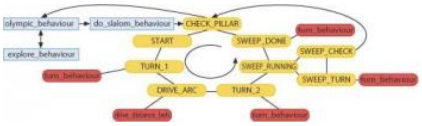
```
void bot_goto_light(void){
  ...
  curve = (sensLDRL - sensLDRR)/1.5;
  ...
  bot_drive(curve, speed);
}
```

Ist der Roboter endlich nahe genug an ein Hindernis herangekommen und melden die Lichtsensoren

ausreichende Helligkeit, geht der Bot davon aus, dass es sich um eine Slalomstange handelt. Er fährt dann um das Hindernis herum, bis das nächste auftaucht. Die Anzahl der notwendigen internen Zustände (siehe Abbildung unten) reduziert ein einfacher Kniff: Jeder Zustand ist auch für spiegelsymmetrische Situationen nutzbar. Verliert der Bot die aktuelle Slalomstange aus den Augen, wechselt er wieder in den Zustand, in dem er `bot_olympic_behaviour()` verlassen hat:



Solange der Bot kein Licht sieht, erkundet er systematisch die Gegend (1). Entdeckt er eine Lampe, fährt er darauf zu (2), um neben der Lichtquelle in den Slalomkurs einzuschwenken (3).

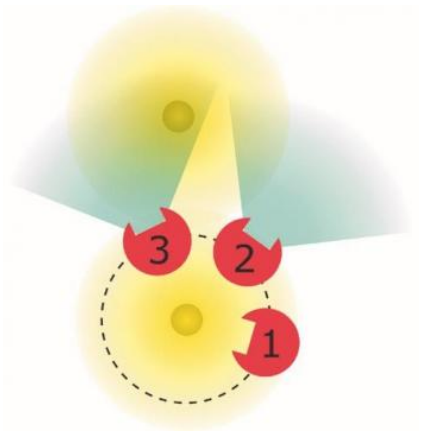


Bei seiner Slalomfahrt durchläuft der c't-Bot eine Reihe von inneren Zuständen (gelb) und benutzt geschickt Verhaltensweisen der unteren Architekturebenen (rot).

```
void bot_do_slalom_behaviour(Behaviour_t *data){
  ...
   switch(state){
   case SLALOM_STATE_CHECK_PILLAR:
     if(check_for_light()){
       if(bot_avoid_harm()){
         state = SLALOM_STATE_START;
       } else bot_goto_light();
     } else {...
       return_from_behaviour(data);
     } break;
```

Umsichtig

An der ersten Säule fährt der Bot rechts vorbei (`orientation == SLALOM_ORIENTATION_RIGHT`), dazu dreht er sich zurerst um 90 Grad nach rechts und läuft anschließend eine 20 Zentimeter lange Linkskurve um die Säule (siehe Abbildung). Dann riskiert er einen Blick zur anderen Seite und dreht sich 45 Grad nach außen (`SLALOM_STATE_TURN_2`). Anschließend streift sein Blick in sechs Schritten über einen Sektor von insgesamt 90 Grad:



Die beleuchtete Säule nimmt der Bot fest ins Visier (1). Nach Drehung und Bogenfahrt von 20 Zentimetern prüft er, ob sich im grünen Sektor eine weitere Slalomstange befindet (2) - falls nicht, setzt er die Fahrt

fort und prüft anschließend erneut (3).

```
case SLALOM_STATE_SWEEP_RUNNING:
  if(sweep_steps == 0){
    sweep_state = SWEEP_STATE_CHECK;
  }
  if(sweep_steps < 6) {
    if(sweep_state == SWEEP_STATE_CHECK){
      if(is_good_pillar_ahead() == True){
        state = SLALOM_STATE_CHECK_PILLAR;
        orientation = (orientation == SLALOM_ORIENTATION_LEFT) ? SLALOM_ORIENTATION_RIGHT : SLALOM_OF
        sweep_steps = 0;
      } else {
        sweep_state = SWEEP_STATE_TURN;
      }
    }
    if(sweep_state == SWEEP_STATE_TURN) {
      turn = (orientation == SLALOM_ORIENTATION_LEFT) ? 15 : -15;
      bot_turn(data,turn);
      sweep_state = SWEEP_STATE_CHECK;
      sweep_steps++;
```

Die einzelnen Schritte dieses Rundblicks verteilen sich wiederum auf zwei Zustände, in denen sich der Bot entweder dreht (`SWEEP_STATE_TURN`) oder prüft, ob er eine hübsch beleuchtete Säule in guter Slalomdistanz vor sich hat (`SWEEP_STATE_CHECK`). Ist das der Fall, wählt er diese als neues Slalomziel (`SLALOM_STATE_CHECK_PILLAR`) und kehrt die Orientierung um. Andernfalls wendet sich der Bot wieder reumütig der alten Säule zu und beginnt die Prozedur von neuem mit dem nächsten Bogenstück von 20 Zentimetern Länge.

Hausaufgaben

Mit der beschriebenen Programmierung kommt der c't-Bot auch mit recht unordentlich gesteckten Kursen zurecht. Trotzdem bleiben noch viele Stellen und Parameter im Programm, an denen sich herumschrauben lässt, um das Verhalten des Bot weiter auszufeilen. Vielleicht erweist es sich als günstiger, wenn er sich nicht so häufig umschaut oder wenn er die Säulen im engeren Bogen nimmt? (Aber Vorsicht, unterhalb von acht Zentimetern Distanz funktionieren die Distanzsensoren nicht mehr zuverlässig!) In den Kommentaren zum Code finden sich an einigen Stellen Hinweise darauf, wo es sich anbietet, Verhaltensweisen zu erweitern und zu verbessern. Codepatches, die den Bot im Slalom auf der Ideallinie halten, veröffentlichen wir wie immer gerne auf der Projektseite [3]. Der vorgestellte Rahmen einer einfachen Subsumptions-Architektur kann aber auch leicht als Grundlage für völlig anderes Verhalten benutzt werden. (**pek[1]**)

Literatur

[1] W. Grey Walters kybernetische Schildkröten

[2] **Benjamin Benz, Peter König, Lasse Schwarten, Drängelnde Spielgefährten, Kollisionen und Sensoren für den c't-Sim, neues Verhalten für den c't-Bot[2]**

[3] **Webseite zum c't-Bot-Projekt[3]**

[4] Rodney A. Brooks, A Robust Layered Control System for a Mobile Robot, MIT AI Memo 864, September 1985 (PDF siehe Soft-Link)

Soft-Link[4]

Projekt c't-Bot

Unser Selbstbau-Roboter zeigt, dass mobile Roboter kein Privileg gut ausgestatteter Forschungslabors sind, sondern sich auch im eigenen Wohnzimmer wohl fühlen. Der c't-Bot bildet eine relativ preiswerte und flexible Plattform für eigene Entwicklungen; die begleitende Artikelserie in c't erleichtert den Einsteig in das spannende Feld der Robotik. Wer nicht selbst löten und schrauben möchte, der erschafft mit dem Simulator c't-Sim virtuelle Roboter - Steuerungsprogramme lassen sich zwischen Simulator und echtem Bot beliebigaustauschen.

Seit c't-Ausgabe 2/06 erscheint in jedem Heft ein neuer Artikel zum Projekt. Einzelne Teile der Serie kann man auf www.heise.de/kiosk/ als PDF nachbestellen, ältere Artikel stehen im Volltext auf der Projektseite [3] bereit, wo auch der Quellcode für Roboter und Simulator sowie Anleitungen, Tipps und Links zu finden sind. Bausätze für Bots verkaufen **eMedia[5]** und **Segor Electronics[6]**.

Über Änderungen informiert die Mailingliste **ct-bot-entwickler@listserv.heise.de[7]**, Anmeldung unter

www.heise.de/bin/newsletter/listinfo/ct-bot-entwickler[8].

Subsumptions-Architektur

Auf Grey Walters kybernetische Schildkröten folgten ganze Generationen von Robotern mit erheblich leistungsfähigeren „Elektronengehirnen“. Anstatt nur ins Licht zu streben, verfolgten sie ganz andere Pläne - im Wortsinn. Denn sie planten ihre nächsten Schritte zuerst, bevor sie handelten - mit Hilfe detaillierter interner Repräsentationen ihrer Umwelt. Auf Rechnern der siebziger und achtziger Jahre konnte das eine Weile dauern - hatte sich die Umgebung inzwischen verändert, mussten die Roboter von vorne anfangen. Erfolg hatten sie damit in vereinfachten Laborwelten, die penibel von Störungen frei gehalten wurden.

Mitte der achtziger Jahre provozierte der Roboter-Pionier Rodney Brooks seine Kollegen mit der Hypothese, dass ein Roboter auf eine Karte, einen Plan und sogar auf ein Gedächtnis verzichten könnte und trotzdem in der Lage wäre, seiner Umwelt geschickt zu begegnen. Gegen Roboter herkömmlicher Art, die lange rechneten und dann wenig taten, führte Brooks seine Idee einer „Intelligenz ohne Repräsentation“ [4] ins Feld. Ab Mitte der achtziger Jahre entwarf er erste Maschinen nach dem neuen Prinzip, das er als „Subsumption Architecture“ bezeichnete. Seine neuen Roboter mussten sich nicht lange besinnen, was zu tun sei - sie fuhren einfach los, reagierten unmittelbar auch auf eine dynamische Umwelt und waren selbst durch das Chaos in manchen Büros des MIT nicht nachhaltig zu verwirren: Sie waren „fast, cheap and out of control“, wie Brooks eine seiner Veröffentlichungen betitelte.

URL dieses Artikels:
`http://www.heise.de/-290392`

Links in diesem Artikel:
[1] `mailto:pek@ct.de`
[2] `https://www.heise.de/ct/artikel/Draengelnde-Spielgefaehrten-290334.html`
[3] `https://www.heise.de/ct/artikel/c-t-Bot-und-c-t-Sim-284119.html`
[4] `http://www.heise.de/ct/06/06/links/218.shtml`
[5] `http://www.emedia.de/`
[6] `http://www.segor.de/`
[7] `mailto:ct-bot-entwickler@listserv.heise.de`
[8] `http://www.heise.de/bin/newsletter/listinfo/ct-bot-entwickler`