c't magazin für computer technik

Kollisionen und Sensoren für den c't-Sim, neues Verhalten für den c't-Bot

Drängelnde Spielgefährten

Praxis & Tipps | Praxis .. Uhr
Benjamin Benz, Peter König, Lasse Schwarten

Schluss mit der Einsamkeit im Robotersimulator! Die neue Version des c't-Sim beherbergt ganze Rudel von Bots und registriert Zusammenstöße der virtuellen Blechkumpane. Neue Sensoren warnen rechtzeitig vor Stürzen in den Abgrund oder erlauben es, die Farbe des Bodens auszuwerten; ein Roboter-Knigge gibt der Maschine Richtlinien für ihr Verhalten vor.

Beim Roboterprojekt c't-Bot können sich sowohl Programmierer als auch Bastler austoben - die einen schrauben und löten ihre eigenen mechatronischen Spielgefährten selbst zusammen, die anderen schicken virtuelle Roboter in den Simulator c't-Sim. Während im vorangegangenen Artikel der Reihe die Hardware und die Montage des Roboters im Mittelpunkt stand, liegt der Schwerpunkt diesmal wieder auf der Software, die eine Reihe von Verfeinerungen erfahren hat - aktuelle Code-Versionen stehen wie immer auf der Projektseite [1] zum Download bereit.

Auch wer weniger an Robotern, dafür aber mehr am Bau künstlicher Welten interessiert ist, kommt hier auf seine Kosten: Der Artikel führt in Konzepte und Details der verwendeten Java3D-Bibliothek ein. Beispiele aus dem c't-Sim zeigen, wie sich die vorgefertigten Klassen geschickt zur Modellierung neuer Sensoren verwenden lassen. Abschließend wendet sich die Aufmerksamkeit dem Verhalten der Roboter zu: Ein neues Framework für Verhaltensroutinen erleichtert die Lösung einfacher Aufgaben, ob in der simulierten oder der realen Welt. Gleichzeitig ist es flexibel genug, später aufwendigere Systeme und Entscheidungsbäume aufnehmen zu können.

Raum für Java

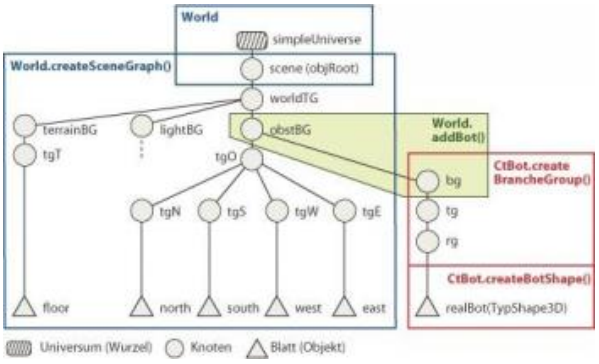
Wie bereits im ersten Artikel zum c't-Sim [2] beschrieben, verwendet der Simulator zur Modellierung und Darstellung der künstlichen Welt die Grafikbibliothek Java3D. Diese enthält eine Reihe nützlicher Methoden, die beispielsweise prüfen, ob Objekte der modellierten Welt miteinander kollidieren. Mit Hilfe fertiger Klassen kann der Betrachter aber auch seinen Blickpunkt im künstlichen Universum versetzen. Begnügte sich der c't-Sim in der ersten Version mit einer starren Aufsicht auf das Geschehen in der Roboterwelt, kann man jetzt nach Herzenslust den Blick schweifen lassen oder seinen Lieblingsbot in Großaufnahme betrachten.

Intern sortiert Java3D die virtuelle Welt streng hierachisch als Baum, auf dessen Zweigen alle Elemente ihren Platz finden: sichtbare und unsichtbare Körper, Texturen und Lichtquellen, Blickpunkte, aber auch Verschiebungen oder Drehungen. Das gesamte Gebilde wird Szenegraph genannt; seine Wurzel bildet das künstliche Universum, seine Blätter die einzelnen sichtbaren Objekte der Simulationswelt wie Boden, Wände oder Bots. Dazwischen liegen Knoten, die für Transformationen sorgen oder das Verhalten und Aussehen von Objekten bestimmen.

Auch die Klassen des Pakets Java3D spiegeln deutlich die zugrunde liegende Baumstruktur: Die Namen der abstrakten Klassen wie `Node` oder `Leaf` orientieren sich an den Begriffen aus der Graphentheorie. Diese Klassen dienen als Basis für abgeleitete Typen wie `Shape3D` (räumliche Körper) oder `TransformGroup` (Verschiebungen und Drehungen), die für den praktischen Einsatz in dreidimensionalen Szenen geeignet sind. Die ausführliche Dokumentation zu den einzelnen Klassen und Methoden des Java3D-API kann man im Netz nachlesen (Links auf der Projektseite [1]).

Simple Universen

Wenig Mühe macht dem Programmierer die Erschaffung einer eigenen Welt, wenn er auf dem Standardtyp `SimpleUniverse` aus dem Paket `com.sun.j3d.utils.universe` zurückgreift: Dieser bringt bereits einen vorgefertigen Szenegraphen mit, dem man lediglich den Zweig der sichtbaren Objekte hinzufügen muss. Für das Beispiel unseres Robotersimulators zeigt die Abbildung unten die Struktur dieses Astes.



Der Szenegraph organisiert die Objekte eines Java3D-Universums streng hierarchisch. Die „Blätter" (Dreiecke ganz unten) enthalten alle sichtbaren Gegenstände der virtuellen Welt.

Der Konstruktor der Klasse World aus dem c't-Sim erzeugt ein `SimpleUniverse` und ruft dann die Methode `World.createSceneGraph()` auf, die alle ortsgebundenen Körper der Welt erzeugt und ihnen ihren Platz anweist. Die gesamten Objekte fasst zunächst die Transformationsgruppe `worldTG` zusammen. Java3D rechnet mit Abmessungen in Metern, der Transformationsvektor (`0.0, 0.0, −2.0`) der Gruppe schiebt somit alle Objekte unterhalb von worldTG um zwei Meter in Richtung der negativen Z-Achse des Koordinatensystems. Dadurch rückt die Roboterwelt ein Stück vom Betrachter weg. Normalerweise würde der Simulator die Kamera so platzieren, dass der gesamte Boden in das Bildschirmfenster passt; ohne die zusätzliche Verschiebung der Perspektive fielen die Außenwände der Welt aus dem Rahmen und wären nicht mehr zu sehen.

Um später gezielte Zugriffspunkte auf den Boden, die Lichtquellen und die Hindernisse zu bieten, erzeugt `createSceneGraph()` drei separate Zweige (`terrainBG, lightBG` und `obstBG`). Die Transformationsgruppe `tgT` sorgt dafür, dass die Fußbodenoberfläche der Roboterwelt auf der relativen Höhe Z=0 liegt, `tgO` lässt die Unterkanten aller Hindernisse bündig mit der Unterkante des Bodens abschließen. Die äußeren Begrenzungen des Bot-Spielplatzes bilden die vier massiven Mauern `north`, `south`, `west` und `east`, sie werden jeweils im Rahmen einer eigenen

Transformationsgruppe (wie `tgN` oder `tgS`) an die richtige Position geschoben. Die Hindernisse sind vom Typ `Box`, der aus dem Paket `com.sun.j3d.utils.geometry` stammt.

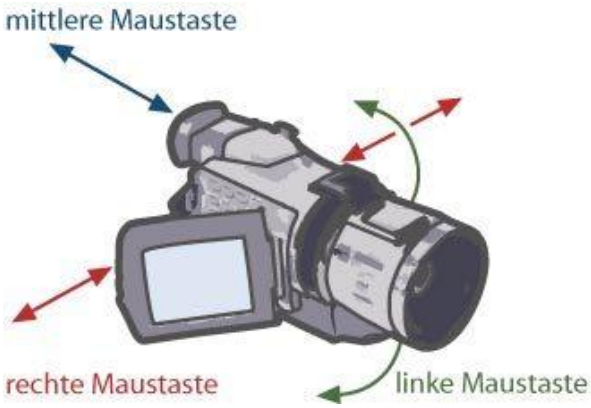
Bots auf Bäumen

Die Bots erzeugen ihre eigenen kleinen Zweige für den Graphen (über `CtBot.createBranchGroup()`). Diese enthalten Knoten für die Verschiebung der Position (`tg`) sowie Rotation (`rg`) und enden mit einem dreidimensionalen Körper (Typ Shape3D), den die Methode `CtBot.createBotShape()` zusammenbaut.

Die Klasse `World` fügt diese Bot-Zweige der Gruppe der Hindernisse (`obstBG`) hinzu. Die Bots sitzen somit auf dem gleichen Ast des Szenebaums wie die Wände - entscheidend ist hierbei die gemeinsame Eigenschaft, dass sie anderen Bots im Weg stehen und ihnen die freie Fahrt nehmen können, was für die Kollisionserkennung eine wesentliche Rolle spielt.

Zu jedem virtuellen Roboter führt ein eindeutiger Pfad durch den Szenegraphen, der bei der Wurzel (dem Universum) beginnt und der alle Knoten durchläuft, die den Zustand des Bot bestimmen: `scene` ordnet ihn der Menge der darzustellenden Objekte zu, `worldTG` lässt den Bot nicht in der Luft hängen, wenn die Welt unter seinen Rädern weggeschoben wird, und `obstBG` klassifiziert ihn als potenzielles Hindernis.

Ein weiterer, in der Abbildung nicht dargestellter Ast des Szenegraphen organisiert den Einblick in das künstliche Universum. Er legt unter anderem Position und Blickrichtung der virtuellen Kamera fest und sorgt für die Darstellung der Welt innerhalb der grafischen Programmoberfläche. Wer den c't-Sim erweitern will, muss sich um diesen Zweig allerdings weniger kümmern: Das verwendete `SimpleUniverse` bringt bereits eine vorgefertigte Sicht auf die Welt mit, deren Standardeinstellungen sich bisher als durchweg brauchbar erwiesen haben.



Eine virtuelle Kamera blickt in die Simulatorwelt: Klicken der linken Maustaste und anschließendes Ziehen schwenkt sie um die eigene Achse, die rechte Taste verschiebt ihre Position quer zur Blickrichtung, die mittlere reguliert die Entfernung.

Gleich knallts

Java3D kann eine künstliche Welt aber nicht nur einfach darstellen. Fertige Klassen liefern auf bequeme Weise detaillierte Informationen zu Überschneidungen von Objekten. Das Prinzip: Die Klasse `PickInfo` bildet zwischen einem ausgewählten Körper (vom Typ `PickShape`) und allen Objekten aus einem bestimmten Zweig des Szenegraphen die Schnittmenge. `PickInfo` liefert anschließend je nach Wunsch die Entfernung vom Ursprung des `PickShape` bis zum ersten Schnitt zurück oder auch eine Liste aller geschnittenen Objekte nach Abstand sortiert. Solche Schnittinformationen nutzt der c't-Sim für die Modellierungen verschiedener Sensoren - und für die handfeste Umsetzung von Kollisionen eines Bot mit Wänden oder seinen Artgenossen.

In der ersten Version des c't-Sim blieb der Roboter zwar an Mauersteinen hängen,

Zusammenstöße unter Bots blieben allerdings unerkannt. Wer testweise mehr als einen Roboter in den Simulator schickte, konnte glauben, Gespenster zu sehen: Die Bots fuhren körperlos durcheinander hindurch, und auch ihre Abstandsensoren nahmen keinerlei Notiz von Kollegen.

Abhilfe schafft in der neuen Version des c't-Sim die Methode `world.checkCollision()`, die Bots aus `updateStats()` heraus aufrufen. Streng genommen überprüft der Simulator allerdings nicht, ob eine Kollision vorliegt, sondern testet, ob sich der Körper des Bot mit einem Hindernis überschneiden würde, wenn er die angestrebte neue Position einnähme. In diesem Fall verharrt er am bisherigen Ort, zur optischen Anzeige der Kollision färbt sich seine Karosserie blau.

Objekte, deren Überschneidung geprüft werden soll, müssen in Java3D über die Methode `setPickable()` als „greifbar" deklariert werden. Das gilt für den Boden, die Wände und auch die Bots - mit Ausnahme des einen, der die Prüfung selbst ausgelöst hat. Dazu versteckt sich dieser in der Methode `world.checkCollision()` kurzfristig vor der Welt:

```
pickShape = new PickBounds(bounds);
synchronized (obstBG) {
  botBody.setPickable(false);
  pickInfo = obstBG.pickAny(
    PICK_BOUNDS, NODE, pickShape);
  botBody.setPickable(true);
  }
```

Die Methode `pickAny()` lässt sich auf jede `BranchGroup` anwenden und liefert seit Version Java3D 1.4 ein Objekt des Typs `PickInfo` zurück - dieses enthält Angaben zu Überschneidungen zwischen den Objekten des gewählten Zweigs (hier die Gruppe der Hindernisse) und dem übergebenen `PickShape`, in diesem Fall die Grenzen (`bounds`) des Bot. Gibt es keine Überschneidungen, so zeigt `PickInfo` ins Leere (`null`); andernfalls liegt eine Kollision vor, und der Bot wird an weiterer Fortbewegung gehindert. Während der Bot selbst gerade nicht sichtbar ist, darf kein anderer Thread (insbesondere kein anderer Bot) auf die Gruppe der Hindernisse zugreifen (was `synchronized` sicherstellt), sonst nehmen sich mehrere Roboter im Simulator nicht mehr zuverlässig gegenseitig wahr - sie verstecken sich voreinander.

Auf der Kippe

Der c't-Sim greift an vielen Stellen auf vorhandene Klassen aus Java3D zurück: Drei weitere `PickInfo`-Objekte beantworten die Frage, ob der Roboter noch festen Boden unter seinen Rädern hat. Die Grundplatte des simulierten Bot befindet sich im Normalfall 15 mm über dem Boden (wie viele andere Eigenschaften des Bot spiegelt sich dieses Maß in einer Konstanten der Klasse `CtBot` wider, in diesem Fall `BOT_GROUND_CLEARANCE`). Vom Zentrum der Räder und des Gleitpin aus wird jeweils mittels eines `PickRay` (Auswahlstrahl) senkrecht nach unten überprüft, ob dort irgendwo fester Boden zu finden ist, und falls ja, ob die Höhe des Sensors über Grund höchstens der vorgesehenen Bauchfreiheit des Bot entspricht. Andernfalls gibt die Methode false zurück, und der Bot kippt um:

```
pickShape = new PickRay(pos, heading);
pickInfo = terrainBG.pickClosest(PICK_GEOMETRY, CLOSEST_DISTANCE, pickShape);
if (pickInfo == null) {
```

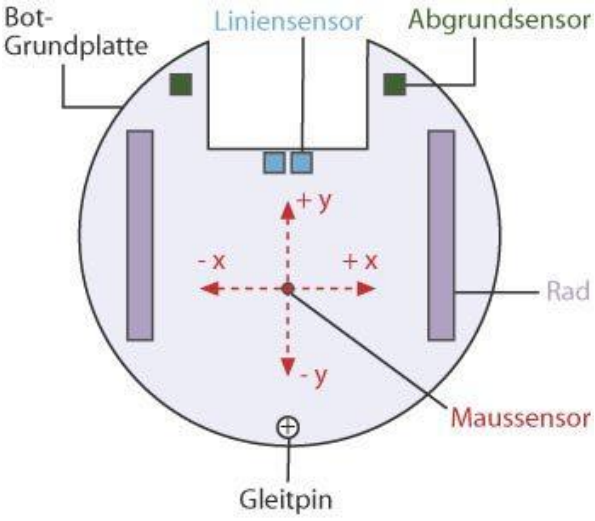
```
  return false;
} else if (pickInfo.getClosestDistance() > groundClearance) {
  return false;
} else return true;
```

Während der echte Bot in diesem Fall unweigerlich den Gesetzen der Physik folgt und in die Tiefe stürzt, begnügt sich der Simulator mit einem Farbwechsel: Schlägt das leuchtende Roboter-Rot in Grün um, so hat der Bot den Boden unter den Rädern verloren. Gleichzeitig werden auf der Kontrolltafel die Kontrollkästchen „setzen“ aktiviert, und der Wackelkandiat kann mit Hilfe der Schieberegler wieder auf sicheres Terrain versetzt werden.

Im Unterschied zur Kollision mit Wänden (bei der alle fraglichen Objekte geprüft werden) interessiert bei der Abstandsmessung zum Boden ausschließlich der Schnitt des `PickShape` mit dem nächstliegenden Objekt. Deshalb kommt hier die Methode `pickClosest()` statt `pickAny()` zum Einsatz. Wird beim Aufruf der Parameter `CLOSEST_DISTANCE` mit übergeben, lässt sich aus dem `PickInfo` direkt die kürzeste Distanz zum nächstgelegenen Objekt herauslesen.

Abstand halten!

In der Version 0.1 billigte der c't-Sim seinen Bots lediglich die Rad-Encoder und zwei IR-Entfernungssensoren zu [2], welche die Entfernung zum nächsten Objekt innerhalb ihres streng begrenzten Blickwinkels messen. Sie funktionieren ganz ähnlich wie die Kontrolle des festen Stands, allerdings kommt in `world.watchObstacle()` ein konischer Auswahlstrahl (`PickConeRay`) zum Einsatz. Er entspringt am Ort des jeweiligen IR-Sensors, seine Längsachse liegt parallel zur Blickrichtung des Bot und sein Öffnungswinkel beträgt drei Grad.



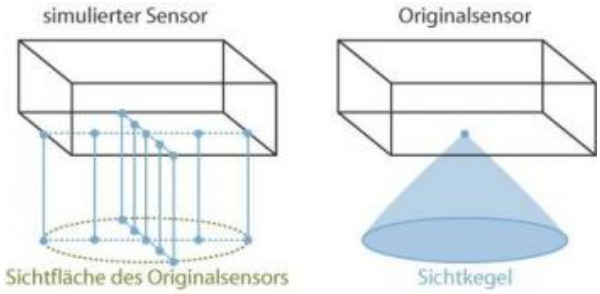
Abgrund- und Liniensensoren sitzen unterhalb der Grundplatte des c't-Bot und reagieren sowohl auf die Entfernung als auch auf die Farbe des Bodens, der Maussensor registriert die Bewegung über Grund.

Wie sein Vorbild aus Aluminium, Halbleitern und Draht verfügt der c't-Bot in der neuen Version des Simulators über weitere Sensoren. Vor dem drohenden Absturz in die Tiefe schützen zwei Abgrundfühler, die vorne links und rechts neben dem Fach des Bots unter der Grundplatte sitzen und senkrecht nach unten die Entfernung des Bodens im Auge behalten (siehe Abbildung oben). Im Original bestehen sie aus je einer Leuchtdiode und einem Phototransistor, der misst, wie viel des ausgestrahlten Lichts vom Boden reflektiert wird [3]. Tiefe Blicke sind nicht die Sache dieses Sensors: Bereits ab etwa drei Zentimetern Entfernung erkennt er keine Objekte mehr. Außerdem lässt er sich von einer perfekt schwarzen Fläche irritieren, die alles Licht verschluckt.

Würde man den beachtlichen Einfallswinkel der Abgrundsensoren von rund 80 Grad analog zu den IR-Entfernungsmessern über ein PickConeRay-Objekt modellieren, erhielte man immer nur die kürzeste Distanz zum nächsten

Objekt - der Sensor würde (anders als sein reales Vorbild) eine Stufe erst melden, wenn sein gesamter Einfallskegel ins Nichts blickt.

Der c't-Sim beschreitet daher in den Methoden `World.sensGroundReflectionCross()` und `sensGroundReflectionLine()` einen anderen Weg: Der Sensor misst mit einer festgelegten Anzahl von parallelen Peil-Strahlen die Entfernung zum nächsten Objekt. Die Strahlen sind dabei kreuzförmig angeordnet und decken eine Fläche ab, die in etwa dem Schnitt des realen Einfallskegels des Sensors mit dem Boden entspricht (siehe Abbildung unten). Die Ergebnisse der einzelnen Punkt-Lotungen werden über das arithmetische Mittel zu einem einzigen Messwert zusammengefasst.



Verrechnete Messwerte einzelner Peilstrahlen (links) sorgen für die Simulation eines Sensors mit breitem Blickwinkel (rechts).

Nicht nur die Entfernung des Bodens, auch dessen Farbe hat Einfluss auf das Messergebnis des Sensors - in der Realität wie in der Simulation:

```
shape = (Shape3D) pickInfo.getNode();
shape.getAppearance().getMaterial().getDiffuseColor(color);
absorption += 1-(color.x + color.y + color.z)/3;
```

Die Methode räumt den drei Farbkanälen des RGB-Modells vereinfachend gleiches Gewicht für die Helligkeit eines Farbtons ein - zumindest für Grautöne ergeben sich durch diese Implementierung realistische Werte. Damit kann die gleiche Methode im Simulator auch zum Auslesen der beiden Liniensensoren benutzt werden. Diese sind im Original tatsächlich baugleich mit den Abgrundsensoren und sitzen am hinteren Ende des Faches unterhalb der Bodenplatte (siehe Abbildung oben).

Es werde Licht

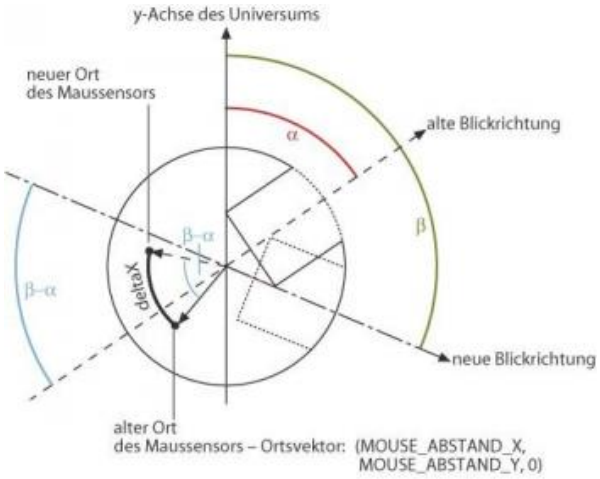
Bisher zeigte sich der Robotersimulator in gleichmäßige Helligkeit getaucht - die neue Version kann man mittels punktförmiger Lichtquellen stimmungsvoller illuminieren. Die notwendigen Leuchten enthält der Zweig `lightBG` des Szenegraphen. Sie bestehen aus zwei Komponenten: der eigentlichen Lampe (Typ `PointLight`), die von Java3D bei der Berechnung des Lichteinfalls in der grafischen Darstellung benutzt wird, und einer `Sphere`, einem hellen gelben Ball, welcher der Lichtquelle einen virtuellen Körper verleiht.

Da diese Leuchtkugel über `setPickable(true)` als greifbar definiert ist, lässt sich die Distanz zu einer Lichtquelle im Simulator genauso komfortabel ausrechnen wie die Strecke zwischen den IR-Sensoren und der nächsten Wand. Die Implementierung der beiden Lichtsensoren des Bot birgt daher kaum Überraschungen, die `Methode World.sensLight()` prüft lediglich zusätzlich, ob der Sensor nicht bereits außerhalb der Tragweite der Lichtquelle liegt, welche die Konstante `World.LIGHT_SOURCE_REACH` festlegt. Die Helligkeitssensoren können auf dem echten Roboter wahlweise nach vorne oder nach oben ausgerichtet montiert werden; im Simulator legt die Konstante `CtBot.SENS_LDR_HEADING` diese Richtung fest.

Auf Mausspur

Ein optischer Maussensor ähnelt einer winzigen Graustufen-Kamera, die in sehr kurzen Abständen Fotos von der überstrichenen Fläche schießt, jedes Bild mit dem vorherigen vergleicht und daraus errechnet, wie weit der Sensor verschoben wurde. Ein solcher Sensor sitzt 15 mm hinter der Querachse des c't-Bot und berührt beinahe den Boden (siehe [4]). Mit seiner Hilfe kann überprüft werden, ob der Bot tatsächlich den Weg zurücklegt, den sein Programm für ihn vorsieht.

Der echte Maussensor liefert Positionsveränderung des Bot in Längs- und in Querrichtung zurück, der Simulator berechnet beide Werte für jeden Schritt getrennt. Für die Bewegungkomponente parallel zur Blickrichtung (`deltaY`) kann in der `Methode CtBotSim.updateStats()` einfach die alte Position von der neuen abgezogen werden:



Der seitliche Versatz des Maussensors errechnet sich aus der geänderten Blickrichtung des Bot.

```
Vector3f vecY = new Vector3f(getPos());
vecY.sub(oldPos);
```

Anschließend wandelt die Methode `meter2Dots` dieses Maß in Metern entsprechend der optischen Auflösung des Sensors um. Die Auflösung definiert die Konstante `CtBot.SENS_MOUSE_DPI`, wie bei Scannern ist sie in Dots per Inch (DPI) angegeben.

Seitlich wandert der Maussensor in einem Kreis um den Mittelpunkt des Bot (siehe Abbildung oben); bei der Berechnung dieser Bewegungskomponente ist daher nur die Änderung der Orientierung von Belang. Die Hilfsmethode `SimUtils.getRotation()` verwandelt die alte und die neue Blickrichtung in Bogenmaße bezogen auf die Richtung der positiven y-Achse der Simulatorwelt. Deren Differenz, multipliziert mit der Distanz des Sensors vom Bot-Zentrum und wiederum von Metern in Punkte gewandelt, ergibt den gesuchten seitlichen Ausschlag (`deltaX`):

```
angleDiff = getRotation(newHeading) - getRotation(oldHeading);
vecMs = new Vector3f(SENS_MOUSE_ABSTAND_X, SENS_MOUSE_ABSTAND_Y, 0f);
deltaX = meter2Dots(angleDiff * vecMs.length());
```

Sensordaten liegen dem c't-Bot inzwischen genügend vor, aber was fängt er damit an? Unabhängig davon, ob er sich in einer virtuellen oder der realen Welt bewegt, von zentraler Bedeutung ist sein Verhalten. Man möchte dem Roboter meist einen recht abstrakten Befehl wie „Fahre von Punkt A zu Punkt B“ oder „Erkunde das Gelände“ geben. Damit der kleine Spielgefährte sich beim Ausführen des Befehls aber nicht wie ein Lemming von der Tischkante stürzt oder versucht, mit dem Kopf durch die Wand zu fahren, braucht er Entscheidungsspielraum. Während der Arbeit an seinen Aufgaben muss er die Sensoren abfragen, deren Daten auswerten und unter Umständen darauf reagieren - zum Beispiel einem Hindernis ausweichen und später auf den alten Pfad zurückkehren.

Um das Verhalten des c't-Bot kümmert sich der C-Teil des Projektes, wobei virtuelle und reale Roboter ein und denselben Code ausführen können. Die Routinen in der Java-Klasse `CtBotSimTest` dienen nur dem Funktionstest des Simulators und sind nicht als Umgebung für ausgefeilte Robotersteuerungen vorgesehen.

Das C-Projekt (ct-Bot) lässt sich entweder für den PC oder den Mikrocontroller übersetzen. Auf dem PC liefert der c't-Sim die Sensorwerte per TCP/IP-Verbindung an [2], der Mikrocontroller dagegen ermittelt sie direkt aus realen Sensoren. Haupt- und Verhaltensprogramm bekommen von diesem Unterschied aber nichts mit. Alle für die Steuerung wichtigen Code-Teile befinden sich in der Datei bot-logik.c. Die globalen Variablen `speed_l` und `speed_r` liefern die aktuelle Motordrehzahl. Sensorwerte stehen ebenfalls zur Verfügung; sensor.h gibt Aufschluss über ihre Namen und Wertebereiche. Routinen können direkt auf diese globalen Variablen zurückgreifen:

```
void bot_avoid_border(Behaviour_t *data){
  if (sensBorderL > BORDER_DANGEROUS)
    speedWishLeft=-BOT_SPEED_NORMAL;
  if (sensBorderR > BORDER_DANGEROUS)
    speedWishRight=-BOT_SPEED_NORMAL;
}
```

Die „Intelligenz“ des c't-Bot verteilt sich auf eine ganze Reihe solcher Verhaltensregeln. Ein Beispiel stammt von unserem Leser Ralph Glass: `bot_glance()` lässt den Roboter von Zeit zu Zeit nach links und rechts schauen, somit erkennt der c't-Bot, ob er an einer Wand entlang schleift. Ohne solche kurzen Blicke zur Seite entgehen ihm möglicherweise Hindernisse, auf die er im spitzen Winkel zufährt, da das Gesichtsfeld der Sensoren sehr schmal ausfällt.

```
void bot_glance(Behaviour_t *data){
  static int16 glance_counter = 0;
  glance_counter++;
  glance_counter %= (GLANCE_STRAIGHT + 4*GLANCE_SIDE);
  if (glance_counter >= GLANCE_STRAIGHT){
    if (glance_counter < GLANCE_STRAIGHT+GLANCE_SIDE){
      faktorWishLeft = GLANCE_FACTOR;
      faktorWishRight = 1.0;
    } else if (glance_counter < GLANCE_STRAIGHT+ 3*GLANCE_SIDE){
      faktorWishLeft = 1.0;
      faktorWishRight = GLANCE_FACTOR;
```

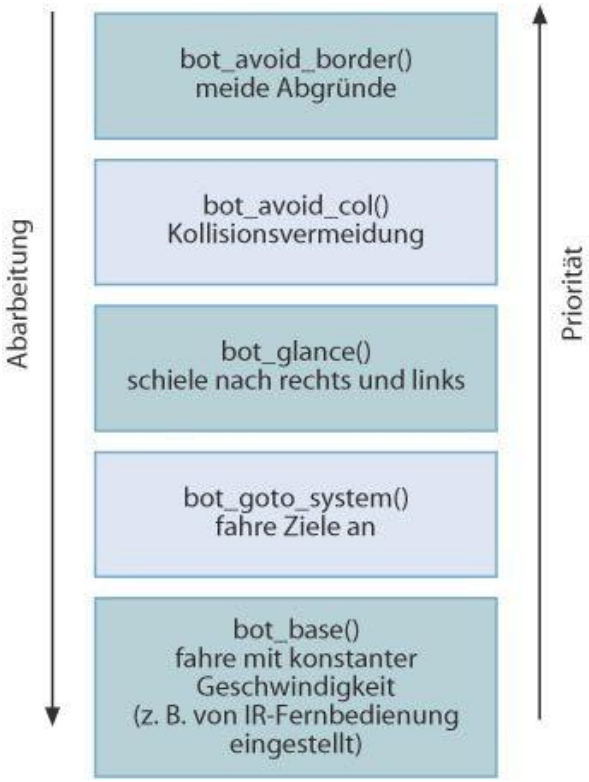
```
    } else{
      faktorWishLeft = GLANCE_FACTOR;
      faktorWishRight = 1.0;
    }
  }
}
```

Jede Verhaltensregel kann die Geschwindigkeiten der beiden Motoren absolut einstellen, indem sie die globalen Variablen `speedWishLeft` und `speedWishRight` setzt. Die Modifikatoren `faktorWishLeft` sowie `faktorWishRight` dagegen bewirken nur eine Veränderung der Geschwindigkeitswünsche der nachfolgenden Regeln.

Das `bot_goto_system()` arbeitet über mehr als einen Aufruf hinweg und merkt sich seinen eigenen Zustand. Es weist den Roboter an, eine Rad-Encoder-Position anzufahren. Der Auftrag `bot_goto(-20, 20);` befiehlt, das linke Rad 20 Encoder-Schritte (entspricht einer halben Radumdrehung) rückwärts und das rechte 20 vorwärts zu drehen, was eine Rotation um die eigene Achse ergibt. In eigenen Verhaltensregeln kann man problemlos auf die Funktion `bot_goto()` zurückgreifen, allerdings löscht jeder neue Auftrag den bisherigen.

Koordinator

Damit mehrere Verhaltensroutinen miteinander harmonieren, verwaltet die Bot-Logik das Regelwerk in einer verketteten Liste, die `bot_behave_init()` aufbaut. Typischerweise geschieht dies bereits beim Programmstart, die Liste lässt sich aber jederzeit erweitern oder verändern. Die Funktion `insert_behaviour_to_list()` sortiert Neuzugänge entsprechend ihrer Priorität ein:



Die Verhaltensroutine bot_behave() arbeitet eine verkettete Liste von Verhaltensregeln ab. Jede

Regel kann entweder die folgenden modifizieren oder direkt eine Motorgeschwindigkeit einstellen, was allerdings alle folgenden Regeln außer Kraft setzt.

```
void bot_behave_init(){
  insert_behaviour_to_list
    (&behaviour, new_behaviour(200, bot_avoid_border));
  insert_behaviour_to_list
    (&behaviour, new_behaviour(100, bot_avoid_col));
  insert_behaviour_to_list
    (&behaviour, new_behaviour(60, bot_glance));
  insert_behaviour_to_list
    (&behaviour, new_behaviour(50, bot_goto_system));
  insert_behaviour_to_list
    (&behaviour, new_behaviour(0, bot_base));
}
```

In regelmäßigen Abständen ruft das Hauptprogramm `bot_behave()` auf. Dieses arbeitet den Regel-Stapel ab und beginnt dabei mit dem Verhalten höchster Priorität. Dazu springt `bot_behave()` die Funktion an, auf die der Zeiger `job->work` verweist. Sobald eine Regel absolute Wünsche äußert (`speedWishLeft oder speedWishRight`), bricht `bot_behave()` die Verarbeitung ab und übermittelt die Geschwindigkeiten an die Motoren. Darunterliegende Regeln kommen nicht mehr zum Zug. So kann der Bot beispielsweise sofort stoppen, wenn er auf einen Abgrund zusteuert.

In vielen Situationen ist ein so rigides Durchgreifen aber weder nötig noch erwünscht: Sieht der Roboter in der Ferne ein Hindernis, so reicht es aus, zum Ausweichen ein Rad etwas zu bremsen. Dazu benutzt eine Verhaltensregel die relativen Variablen (`faktorWishLeft` oder `faktorWishRight`). `bot_behave()` merkt sich diese Modifikatoren und multipliziert sie mit den Zielgeschwindigkeiten der Regeln niedrigerer Priorität. Das Ergebnis ist eine Kombination mehrerer Verhaltensregeln. Sorgen zu viele Regeln eigenmächtig dafür, dass niedriger priorisierte Routinen übersprungen werden, ignoriert der Roboter möglicherweise die IR-Fernbedienung komplett oder weicht Hindernissen nicht mehr zuverlässig aus. Direkte Zugriffe auf die Motoren mit der Funktion `motor_set()` oder auf die globale Variable `speed_x` hebeln das Framework vollständig aus - hiervon sollte man komplett absehen.

Als Übergabeparameter bekommen alle Verhaltensroutinen einen Zeiger auf ihren Listeneintrag (`data`). Dort findet sich unter anderem das Feld `active`, mit dem sich eine Funktion nach getaner Arbeit selbst deaktivieren kann:

```
typedef struct _Behaviour_t {
   void (*work) (struct _Behaviour_t *data);
   uint8 priority;
   char active:1;
   struct _Behaviour_t *next;
} Behaviour_t;
```

Die Reise ins Labyrinth

Mit dem neuen Rahmen für Verhaltensregeln und den erweiterten Sensoren ist der virtuelle Bot gerüstet für Ausflüge in schwierigeres Gelände. Deshalb haben wir die Welt um einen kleinen Hindernisparcours erweitert. Die im Beispiel-Code veröffentlichten Verhaltensregeln geben einen Anhaltspunkt, wie man den Roboter programmieren kann. Clevere Routinen zu erfinden, die den Bot durch einen Eingang in das kleine Labyrinth lotsen und ihn in möglichst kurzer Zeit den anderen Ausgang finden lassen, überlassen wir Ihnen. Bringt Ihr Roboter den Hindernislauf unbeschadet hinter sich, veröffentlichen wir Ihre Steuerroutinen gerne auf der Projektseite [1]. Über gut kommentierte und pfiffige Patches auch zu anderen Aspekten freuen wir uns immer und pflegen diese auch gerne in die Codebasis ein. Ersten Erweiterungen von Leserseite (wie der Schieberegler für die Zeitlupe) haben bereits Eingang in die offizielle Programmversion gefunden.

Die nächste Folge der Artikelserie richtet sich wieder verstärkt an die Schrauber und Löter, auf deren Werkbänken inzwischen die ersten echten c't-Bots aus Metall und Silizium das Licht der Welt erblicken - Details in der Ansteuerung von Sensoren und Aktuatoren werden im Blickpunkt stehen. Mit theoretischen Modellen, Roboter-Architekturen und deren verschiedenen Implementierungen beschäftigt sich ein späterer Artikel der c't-Bot-Reihe. (**pek[1]**)

Literatur

[1] **Webseite zum c't-Bot-Projekt[2]**

[2] **Benjamin Benz, Peter König, Virtuelle Spielgefährten, Simulator für c't-Bots[3]**

[3] **Benjamin Benz, Carl Thiede, Thorsten Thiele, Spielgefährten, Roboter für Löter, Simulator für Soft-Werker[4]**

[4] **Benjamin Benz, Carl Thiede, Thorsten Thiele, Hallo Welt, Aufbau und Inbetriebnahme des c't-Bot[5]**

Soft-Link[6]

Die Java3D-Bibliothek

Java3D ist eine Bibliothek für die Modellierung und Darstellung dreidimensionaler Objekte in Java-Programmen. Ursprünglich stammt sie (wie die bekannteste Java-Implementierung) aus dem Hause Sun, der Code wurde aber bereits vor einiger Zeit zur Weiterentwicklung im Netz freigegeben.

Die Programmierschnittstelle Java3D bewegt sich auf einer etwas höheren Ebene als der Standard OpenGL (Open Graphics Library) oder die DirectX-Komponente Direct3D von Microsoft. Da Java3D wahlweise auf eine der anderen beiden Bibliotheken aufsetzt, benötigt das Paket selbst keinen direkten Zugang zu Hardware-Treibern. Der Preis für diese Plattformunabhängigkeit besteht in der geringeren Geschwindigkeit der Darstellung; für die Zwecke des Roboter-Simulators c't-Sim sollte das Tempo von Java3D aber auch auf etwas betagteren Maschinen ausreichen (sofern die Zahl der eingesetzten Bots überschaubar bleibt). Voraussetzung ist allerdings eine Grafikkarte, die OpenGL 1.3 oder höher unterstützt.

Derzeit gibt es keine konkreten Pläne für Erweiterungen der Schnittstelle zur Anwendungsprogrammierung (API, Application Programming Interface) um neue Klassen und Möglichkeiten. Die Entwickler arbeiten derzeit mit Hochdruck an Version 1.4, die Freigabe der finalen Version ist für Ende Februar 2006 geplant.

URL dieses Artikels:

http://www.heise.de/-290334

Links in diesem Artikel:

[1] mailto:pek@ct.de

[2] https://www.heise.de/ct/artikel/c-t-Bot-und-c-t-Sim-284119.html

[3] https://www.heise.de/ct/artikel/Virtuelle-Spielgefaehrten-290294.html

[4] https://www.heise.de/ct/artikel/Spielgefaehrten-290274.html

[5] https://www.heise.de/ct/artikel/Hallo-Welt-290314.html

[6] http://www.heise.de/ct/06/05/links/224.shtml