c't-Bot und c't-Sim

Sven Sakowski
Hochschule für Technik Stuttgart

17. Dezember 2006

Inhaltsverzeichnis

1 Einleitung

Beim Thema Roboter denken viele an die zwei sympathischen Roboter aus „Star Wars“, C3PO und R2D2. Oder man denkt an die industrielle Massenproduktion, bei der Roboter am Fließband produzieren und mit höchster Präzision immer die gleichen Tätigkeiten ausführen. Möchte man sich mit der Programmierung von Robotern befassen hat man eine reihe von Bausätze zur Auswahl, die es ermöglichen, Erfahrungen mit der Programmierung von Robotern zu sammeln. Ein Nachteil dieser Roboter-Bausätze wie sie von Lego oder Fischer-Technik angeboten werden ist, dass man hier voll und ganz auf die Schnittstellen des Herstellers angewiesen ist.
Flexible Schnittstellen und Erweiterbarkeit bietet der c't-Bot, der vom Heise Verlag initiiert und betreut wird. Beim c't-Bot handelt es sich um einem Bausatz für einen voll programmierbaren, autonomen Roboter. Bei der Konstruktion des c't-Bots wurde darauf geachtet, dass das Zusammenbauen auch für ungeübte Bastler in ein paar Stunden möglich ist.
Die Software für den c't-Bot wird in der Programmiersprache C geschrieben. Der gesamte Quelltext ist frei verfügbar und darf verändert und erweitert werden. Dadurch erhält man einen umfassenden Einblick in die Roboterprogrammierung und kann eigenen Ideen selbst ausprobieren. Gerade für die, die sich noch nie mit der Programmierung von Robotern beschäftigt haben bietet der c't-Bot die Möglichkeit dies ohne großen Aufwand zu tun.
Um die Programmierung des c't-Bot zu testen, steht ein Simulator zu Verfügung. Hiermit lassen sich alle Funktionen des Bot testen, ohne das Gefahr besteht die Hardware zu beschädigen.
Der herfür verwendete c't-Sim ist ein in Jave geschriebener Simulator. Hier kann genau der Quelltext getestet werden der auch auf dem c't-Bot ausgeführt wird. Der Quelltext des c't-Sim ist frei verfügbar und darf beliebig verändert werden. Der c't-Sim veranschaulicht grafisch wie der c't-Bot zum Beispiel auf Hindernisse reagiert oder ob er einen Abgrund als solchen erkennt.
Der große Vorteil der Kombination c't-Bot und c't-Sim ist, dass man sofort anfangen kann zu programmieren und zu testen, auch wenn man die Investition in den c't-Bot scheut. Die Software für den c't-Bot und den c't-Sim kann kostenlos aus dem Internet geladen werden. Ebenso sind alle benötigten Compiler und Werkzeuge kostenlos im Internet erhältlich.
Im Folgenden soll beschrieben werden wie man den c't-Bot und der c't-Sim programmiert. Hierfür sind grundlegende Programmierkenntnisse in C und Java erforderlich. Später wird auf die Installation der benötigten Software eingegangen.
Danach wird in unabhängigen Teilen auf die Programmierung des c't-Bot sowie auf die Programmierung des c't-Sim eingegangen.
Aufgrund der Komplexität des Themas, können nur Teilbereiche der Programmierung aufgezeigt werden. Für Informationen zu allen Teilbereichen sei auf die Projektseite verwiesen.
Im weiteren wird der c't-Bot einfach als Bot und der c't-Sim als Sim bezeichnet.

2 Einstieg

Dieses Kapitel ist der Einstieg in das „c't-Projekt - c't-Bot und c't-Sim“. Hier wird der Aufbau des Bot beschrieben. Danach wird die Aufgabe des Sim erläutert. Zuletzt werden die Werkzeuge vorgestellt, die zur Programmierung benötigt werden.

2.1. Was ist der Bot

Der Bot ist ein zylindrischer, autonomer Roboter auf zwei Rädern. Er kann nur als Bausatz (Preis ca. 220 Euro für die Basisversion) erworben werden und muss selbst in C programmiert werden. Auf der Projektseite[1] stehen Schaltpläne, Dokumentation, ein Forum und Quelltext zur freien Verfügung.
Der Bot ist so Aufgebaut, dass seine Hardware erweitert werden kann. Man kann Erweiterungen für den Bot kaufen und eigene Konstruktionen integrieren. Im Folgenden werden die wichtigsten Bauteile aufgezeigt:
Die Grundplatte des Bot ist eine Metallscheibe im Durchmesser von 12 cm. Auf der einen Seite hat die Scheibe eine rechteckige Aussparung, die so groß ist, dass sie einen Golfball aufnehmen kann. Neben der Aussparung sind Schlitze in der Grundplatte für die beiden Räder. Damit der Bot nicht kippt, ist gegenüber der Aussparung auf der Unterseite der Metallplatte ein Halbkugel förmiger Kunststoffgleiter montiert. Der Akku-Block, bestehend aus fünf Mignon-Zellen, ist über dem Gleiter montiert, so dass der Bot nicht nach vorne kippen kann.

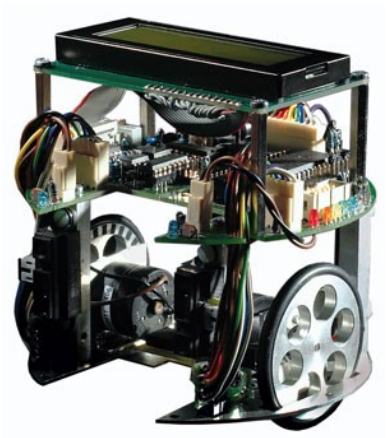

Abbildung 1: c't-Bot [Bild1]

Angetrieben wird der Bot durch zwei, auf der Grundplatte befestigten, 6 Volt Gleichstrommotoren mit angeschlossenem Untersetzungsgetriebe (33:1). Dadurch entwickelt der Antrieb ein Drehmoment von0,03 Nm. Bei einem Raddurchmesser von 57 mm und einer maximalen Umdrehungszahl von 151 U/min erreicht der Bot eine Geschwindigkeit von 0,45 m/s (1,6 km/h).
Über dem Antrieb befindet sich die Hauptplatine, die die gleiche Form wie die Grundplatte hat. Hier befindet sich der Mikrocontroller (Atmega32), der die Motoren steuert und die Sensordaten auswertet. Der Controller hat eine Taktfrequenz von 16 MHz und kann viele Befehle innerhalb eines Taktzyklus ausführen. 32 KByte[2] Flash-Speicher nehmen den eigenen Programmcode auf. Für Variablen stehen 2 KByte SRAM[3] zu Verfügung und 1 KByte EEPROM[4] speichert Parameter auch ohne Versorgungsspannung.
Der Mikrocontroller zeigt nicht an in welchem Zustand oder Programmteil er sich befindet. Um dies zu visualisieren besitzt der Bot acht frei programmierbare LEDs[5]. Mehr

1http://www.heise.de/ct/ftp/projekte/ct-bot/
2Kilo Byte
3Static Random Access Memory = statischer Speicher
4Electrically Erasable Programmable Read Only Memory = programmierbarer Speicherbaustein
5Light Emitting Diode; Leuchtdiode

Informationen liefert ein LC-Text-Display[1] mit 4x20 Zeichen. Hiermit lassen sich ganze Statustexte ausgeben, die Informationen über den Bot beinhalten. So können zum Beispiel Sensordaten ausgegeben werden oder es kann angezeigt werden welcher Programmteil gerade ausgeführt wird. Sucht man Fehler in der eigenen Programmierung, helfen diese Informationen um den Fehler einzugrenzen.
Damit der Bot seine Umwelt wahrnehmen kann, trägt er unterschiedliche Sensoren mit sich:
Auf der Innenseite der Räder kleben Kodierscheiben mit abwechselnd schwarzen und weißen Feldern. Zwei Reflexlichtschranken (CNY70) können so die Drehgeschwindigkeit der Räder messen. Diese Einheit nennt man Rad-Encoder.
Die Rad-Encoder können nur die Drehgeschwindigkeit der Räder ermitteln. Drehen die Räder auf rutschigem Untergrund durch, nehmen sie dies nicht wahr. Deshalb ist in die Bodenplatte des Bot der Sensor einer optischen Maus eingebaut. Damit kann bestimmt werden ob sich der Bot fortbewegt oder ob die Räder durchdrehen.
Damit der Bot nicht gegen Gegenstände fährt, hat er zwei nach vorne gerichtete Abstandssensoren (GP2D12). Diese Sensoren nehmen Hindernisse in einem Abstand zwischen zehn und 80 Zentimeter wahr. Kommt der Bot zu nahe an einen Gegenstand, liefern die Sensoren falsche Werte. Dabei handelt es sich um einen Schachpunkt in der Konstruktion, den man durch eine bessere Positionierung der Sensoren vermieden hätte.
Im vorderen Bereich des Bot tasten zwei Lichtschranken (CNY70) den Boden ab. Dadurch erhält man Informationen ob der Bot auf einen Abgrund zu fährt.
Eine weitere Lichtschranke (CNY70) tastet den Boden unter dem Bot ab. Damit ist es zum Beispiel möglich den Bot dazu zu veranlassen einer schwarzen Schnur zu folgen.
Der Bot kann mit Hilfe zweier Photosensoren Lichtquellen wahrnehmen.
Über einen Infrarot-Sensor kann man dem Bot per Fernbedienung Befehle schicken.
Hiermit ist der Aufbau des Bot im groben beschrieben. Detaillierte Informationen zum Aufbau liefert die Projektseite im Internet. [ct0202]

2.2. Was ist der Sim

Der Sim ist ein frei programmierbarer Simulator. Er ist in Java geschrieben und verwendet zum einfachen Darstellen dreidimensionaler Welten die Erweiterung Java3D. Auf einfache weise kann man damit unterschiedliche Welten wie zum Beispiel Labyrinthe simulieren.
Die Programmierung des Bot kann nicht nur auf dem Mikrocontroller sondern auch auf dem Rechner ausgeführt werden. Damit man sieht wie sich der Bot verhält, ist er mit dem Sim verbunden. Über eine TCP/IP[2] Verbindung simuliert der Sim dem Bot Sensordaten. Umgekehrt bekommt der Sim von Bot simulierte Steuerimpulse für die Motoren.
Der Sim kann dazu verwendet werden um die Programmierung des Bot zu testen, so dass Programmierfehler nicht zu einem Hardwareschaden führen, wenn zum Beispiel eine Tischkante nicht richtig erkannt wird. Mit Hilfe des Sim ist es auch Möglich, den Bot zu programmieren ohne die Hardware zu besitzen oder mehrere Bots zu simulieren.
Neben der simulierten Welt werden die Sensordaten angezeigt, die der Bot empfängt. Dadurch lassen sich leichter Fehler in der Programmierung des Bot finden. Ein Zeitraffer erleichtert ebenfalls die Fehlersuche.

1liquid crystal display; Flüssigkristallbildschirm
2 Transmission Control Protocol/ Internet Protocol = Protokoll zum Übertragen von Daten

Die Funktionen des Sim lassen sich beliebig erweitern und verändern. Bekommt der Bot neue Sensoren oder zum Beispiel einen Greifarm, so muss der Sim entsprechend angepasst werden. [ct0203]

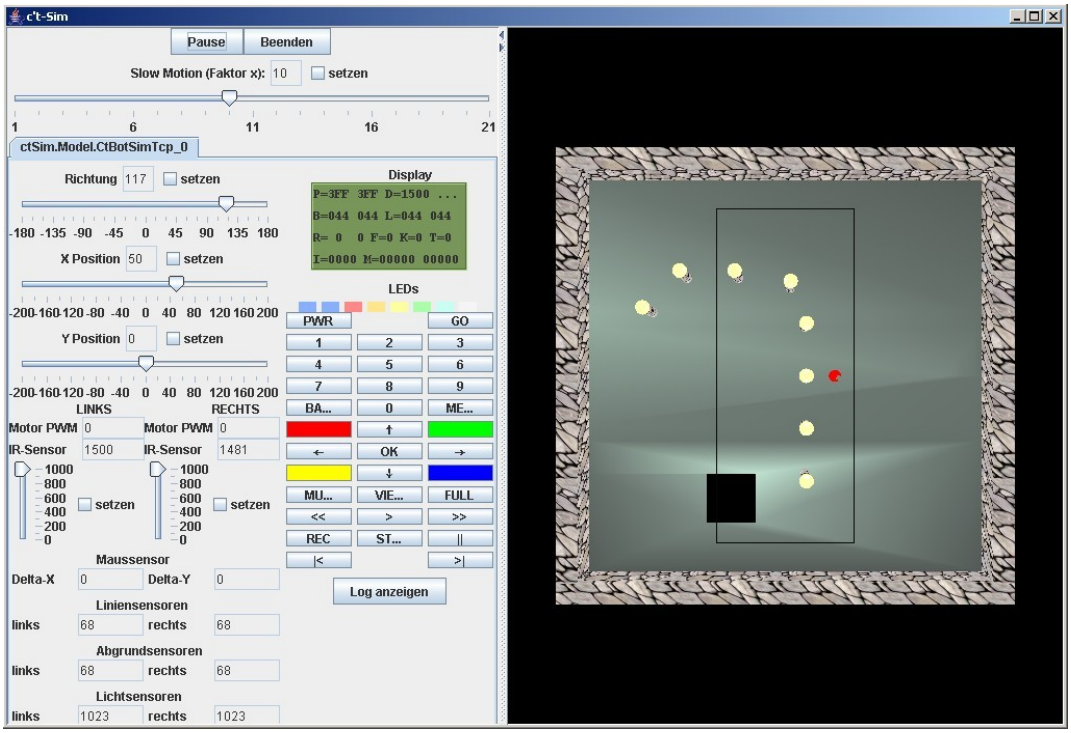

Abbildung 2: c't-Sim [Bild2]

2.3. Voraussetzungen für die Programmierung

Damit man den Bot und Sim programmieren kann, braucht man Informationen. Eine gute Grundlage sind die 13 Artikel in der Computerzeitschrift c't. Zusätzlich bietet die Projektseite viele Informationen zur Installation der Software, Fehlersuche, Hardware und Programmierung. Zudem kann man sich in einem Forum mit anderen Interessierte austauschen.

Auf der Projektseite stehen Quelltext für Bot und Sim bereit. Der Quelltext kann beliebig verändert und erweitert werden. Man muss also nicht alles selbst programmieren, sondern kann auf die Entwicklung von anderen aufsetzen. Das ist praktisch, wenn man sich zum Beispiel mehr für die Verhaltenslogik des Bot als für die Ansteuerung der Hardware interessiert.

Damit man Änderungen und Verbesserungen von anderen in die eigene Programmierung einfließen lassen kann, steht der Quelltext als CVS-Quelle[2] bereit. So gibt es eine einheitliche Quelltext Basis die von jedem genutzt werden kann.

Als Entwicklungsumgebung wird auf der Projektseite Eclipse vorgeschlagen. Man kann aber jede beliebige Entwicklungsumgebung verwenden. Eclipse bietet einige Vorteile gegenüber anderen Entwicklungsumgebungen:

- verfügbar unter Linux und Windows
- Entwicklung von C-Quelltext und Java-Quelltext
- automatisches generieren der Quelltextdokumentation

2Concurrent Versions System; Software-System zur Versionsverwaltung

- standardmäßiger Zugriff auf CVS-Quellen

Um den Sim zu verwenden, braucht man eine Java-Laufzeitumgebung. Möchte man den Sim verändern, braucht man einen Java-Compiler. Für den Bot braucht man einen C-Compiler, der den Quelltext für den Rechner und für den Mikrocontroller übersetzen kann. Welche Werkzeuge für die einzelnen Aufgaben nötig sind, beschreibt detailliert die Projektseite. [ct0202]

3 Programmierung des Bot

Im Folgenden wird die Programmierung des Bot beschrieben. Dabei wird zuerst die Ansteuerung der Motoren und Sensoren erläutert.
Anschließend wird gezeigt wie man die Verhaltenslogik implementieren kann. Zuletzt wird gezeigt, wie der Bot Aufgaben löst.

3.1. Hardware nahe Programmierung

Im Zentrum der Hardware nahen Programmierung steht der Mikrocontroller. Hier laufen alle Informationen der Sensoren zusammen. Manche Sensoren schicken ihre Daten bereits digitalisiert an den Mikrocontroller, andere schicken sie analog. Der Mikrocontroller kann die analogen Daten digitalisieren. Alle Sensordaten liegen nach dem Digitalisieren als Bit-Strings, also Zahlen vor. Um die Hardware auf unterster Ebene zu Programmieren, muss man wissen was die einzelnen Werte bedeuten. Auskunft darüber liefern die Datenblätter der Hersteller.
Um ein Programm für den Mikrocontroller zu schreiben muss man wissen wie die Aus- und Eingänge des Controllers mit der Übrigen Hardware verbunden ist. Wie man die Aus- und Eingänge des Mikrocontrollers ansteuert steht im Datenblatt und wie diese mit der Übrigen Hardware verbunden sind steht im Schaltplan.
Der Hersteller des Mikrocontrollers liefert die benötigten Header-Dateien (Dateien in denen vor allem Zahlen-Konstanten hinterlegt sind). In diesen Header-Dateien sind auch spezielle Datentypen für den Mikrocontroller hinterlegt, da der Controller nur 8-Bit lange Worte verarbeiten kann. Das folgende Beispiel zeigt wie eine LED direkt angesteuert wird.

```
#include <avr/io.h>
volatile uint8 led=0;
/* @param LED Bitmaske der anzuschaltenden LEDs */
void LED_on(uint8 LED){
    led |= LED;
    LED_set(led); }
void LED_off(uint8 LED){
    led &= ~LED;
    LED_set(led); }
```

Bei allen Variablen handelt es sich um Zahlen des Typs uint8. Die Variable LED gibt an, welche LED ein-/ausgeschaltet werden soll. Mit dem bitweisen-Oder setzt man led so, dass der Aufruf von LED_set(led) das einschalten der entsprechenden LED bewirkt (led = led | LED). Ausgeschaltet wird die LED, wenn led mit dem einer-Komplement von LED und dann das bitweise-Und ausgeführt wird. Bit-Operatoren sind das entscheidende Werkzeug bei der Hardware nahen Programmierung.
Auf diese Weise zu programmieren ist wenig intuitiv. Deshalb stehen alle Methoden für die Hardware nahe Programmierung auf der Projektseite bereit. Man kann kann sie sich anschauen und verändern oder man nutzt sie um den Bot zu einer Aktion zu veranlassen. So kann der Einsteiger sich beispielsweise gleich mit der Verhaltenslogik beschäftigen

ohne erste mühsam zu lernen wie man es schafft, das sich die Motoren in die gewünschte Richtung drehen. [ct0206]

3.2. Verhalten 1

An dieser Stelle wird beschrieben, wie der Bot einen Befehl ausführt. Ein abstrakter Befehl wie zum Beispiel „Fahre von A nach B“, ist zu komplex um ihn in einer einzigen Verhaltensregel zu programmieren. Deshalb setzt sich ein Befehl aus vielen kleinen Regeln zusammen, die geschickt hintereinander ausgeführt werden. Dabei darf die Reihenfolge nicht starr vorgegeben sein, denn der Bot muss schnell reagieren können wenn ein Hindernis von den Sensoren erkannt wird.
Die Verhaltenslogik und die Auswertung der Sensordaten findet in unterschiedlichen Modulen statt. Bei der Programmierung der Verhaltenslogik muss man sich nicht darum kümmern, dass aktuelle Sensordaten vorliegen. Über globale Variablen kann die Verhaltenslogik auf die aktuellen Sensordaten zugreifen. Die Header-Datei „sensor.h“ enthält alle Sensor-Variablen mit ihren Wertebereichen.
Zwei Verhaltensregeln sollen hier beispielhaft gezeigt werden, aus denen sich das komplexe Verhalten des Bot zusammensetzt.

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

Die Variable sensBorderL enthält den Wert, des linken Sensors der den Boden vor dem Bot abtastet. Dieser Wert liegt zwischen 0 und 1023. Liefert der Sensor den Wert 1023 so bedeutet dies, das er keinen Boden in seiner Reichweite erkennt. Die Konstante BORDER_DANGEROUS ist mit dem Wert 0x3A0 (928) vorbelegt. Ab diesem Wert kann man sich sicher sein, dass eine Kante vor dem Bot ist. Mit der Anweisung speedWishLeft = - BOT_SPEED_NORMAL wird bewirkt, dass speedWishLeft auf normale Rückwärtsgeschwindigkeit (144 mm/s) gesetzt wird. Das selbe gilt ebenso für die Rechteseite.
Über die globalen Variablen speedWishLeft und speedWishRight kann jede Verhaltensregel die Motordrehzahl bestimmen. Geändert wird die Drehzahl, wenn das nächste mal die Motorsteuerung aufgerufen wird.
Die Methode bot_drive_square_behaviour(Behaviour_t *data) lässt den Bot ein Quadrat fahren. Bei dieser Methode wird der Übergabeparameter *data genutzt. *data ist ein Zeiger auf die Liste aller initialisierten Verhaltensroutinen mit deren Zustand. Dies ist ein einfaches Beispiel für eine Routine die einen Zustand besitzt. Weil sie auf andere Routinen zugreift, setzt sie nicht selbst speedWishRight und speedWishLeft voraus.

```
void bot_drive_square_behaviour(Behaviour_t *data){
#define STATE_TURN 1
#define STATE_FORWARD 0
#define STATE_INTERRUPTED 2
  static uint8 state = STATE_FORWARD;
  if (data->subResult == SUBFAIL) // letzter Auftrag schlug fehl?
    state= STATE_INTERRUPTED;
  switch (state) {
    case STATE_FORWARD: // Vorwaerts
      bot_goto(100,100,data); // beide Räder drehen sich um 100 Einheiten nach vorne
      state = STATE_TURN;
      break;
    case STATE_TURN: // Drehen
      bot_goto(22,-22,data);  // bewirkt eine 90°-Drehung des Bot, das eine Rad dreht eine
                              // halbe Umdrehung nach vorne, das andere eine halbe zurück
      state=STATE_FORWARD;
      break;
    case STATE_INTERRUPTED:
      return_from_behaviour(data);// Beleidigt sein und sich selbst deaktiviern
```

```
      break;
    default:              /* Sind wir fertig, dann Kontrolle zurueck an Aufrufer */
      return_from_behaviour(data);
      break;
  }
}
```

Um die Methode bot_drive_square_behaviour besser zu verstehen ist es Wichtig den Aufbau der Verwaltungsstruktur zu kennen.

```
/*! Verwaltungsstruktur fuer die Verhaltensroutinen */
typedef struct _Behaviour_t {
    void (*work) (struct _Behaviour_t *data); /* Zeiger auf die Funktion, die das Verhalten
bearbeitet */
   uint8 priority;                          /* Prioritaet */
   struct _Behaviour_t *caller ;            /* aufrufendes verhalten */
   uint8 active:1;                          /* Ist das Verhalten aktiv */
   uint8 subResult:2;              /* War das aufgerufene unterverhalten erfolgreich (==1)?*/
   struct _Behaviour_t *next;               /* Naechster Eintrag in der Liste */
} Behaviour_t;
```

Schaltet man den Bot ein, wir eine Liste mit mehreren Verhaltensroutinen geladen. Die Verhaltensroutinen werden gemäß ihrer Priorität in die Liste einsortiert. Die Liste kann während des Betriebs erweitert oder verkürzt werden. Überträgt man beispielsweise einen Befehl per Fernsteuerung an den Bot, so wird die entsprechende Routine in die Liste aufgenommen. [ct0205]

3.3. Regelungstechnik

Sendet der Mikrocontroller den beider Motoren das Signal mit gleicher Geschwindigkeit nach vorne zu fahren muss das nicht bedeuten, dass der Bot geradeaus fährt. Es kann auch sein, dass der Bot einen Bogen fährt. Dafür gibt es unterschiedliche Gründe: Fertigungstoleranzen der Motoren, sinkende Akkuspannung oder Beschaffenheit des Untergrunds um nur einige zu nennen.
Um dem entgegen zu wirken, muss der Mikrocontroller die eingestellte Geschwindigkeit der Motoren mit Hilfe der Lichtschranken an den Rädern messen und Korrekturen an der Motordrehzahl vornehmen. Diesen Kreislauf aus Messung und Nachstellen nennt man Regelung.
Die einfachste Art der Regelungstechnik ist die An-Aus-Schaltung, bei der Schwellenwerte definiert sind. Werden diese Werte Unter- oder Überschritten wird gegen gesteuert. Diese Technik ist für den Bot ungeeignet, da er sich dadurch auf eine Wellenlinie fortbewegen würde. Für den Bot bietet sich ein PID-Regler an, dabei handelt es sich um eine Kombination aus Proportional-, Integral- und Differentialregler. Eine solche Formel kann nicht mit hoher Geschwindigkeit auf dem Mikrocontroller berechnet werden, da für Integral- und Differentialrechnung viele Taktzyklen notwendig sind. Vereinfacht man die Formel für den PID-Regler, so das eine endliche Summe entsteht, kann die Methode sinnvoll angewendet werden. Der Verlust ans Genauigkeit kann vernachlässigt werden, es ist wichtiger die Formel in kurzer Zeit zu berechnen.
Es stellt sich nun die Frage an welcher stelle man die PID-Regel implementiert. Von ihrer Struktur her würde sie bei den Verhaltensregeln implementiert werden. Die anderen Verhaltensregeln steuern aber auch den Motor an und würden so die Einstellungen der PID-Regel rückgängig machen. Deshalb ist es sinnvoller die PID-Regel direkt bei der Motorsteuerung zu implementieren.
Über den PID-Regler hat man zudem die Möglichkeit die Geschwindigkeit des Bot zu bestimmen und nicht wie vorher nur über einen Stellwert. Dadurch kann man die Geschwindigkeit in der sich der Bot fortbewegt viel feiner steuern, denn der Bot achtet nun selbstständig darauf, dass er die geforderte Geschwindigkeit einhält. [ct0209]

3.4. Verhalten 2

Um dem Bot ein komplexeres Verhalten zu geben, soll hier der Ansatz der Subsumptions-Architektur beschrieben werden. Dabei werden einfache Verhaltensmuster, wie zum Beispiel „nicht mit Objekten zusammenstoßen“ in eine abstrakte Handlungsweise wie zum Beispiel „erforsche“ eingefasst oder anders ausgedrückt subsumiert. Dabei ist es wichtig, dass die einfachen Verhaltensmuster allgemein gehalten sind. Dadurch kann der Bot flexibel auf seine Umwelt reagieren.
Der Bot verfügt über eine Reihe von Zuständen. Es ist immer eindeutig bestimmt, in welchem Zustand er sich befindet. Treten äußere Umstände auf die einen Zustandswechsel hervorrufen, ändert sich auch das Verhalten des Bot. Befindet sich der Bot zum Beispiel auf einer Erkundungsfahrt und stellt er fest, das die Akkuspannung nachlässt, so bricht er die Erkundung ab und kehrt an seine Ausgangspunkt zurück.
Die unterschiedlichen Verhaltensmuster befinden sich auf unterschiedlichen Ebenen. Verhaltensmuster die sich tiefer innerhalb der Subsumptions-Architektur befinden, knüpfen direkter an Sensordaten und Steuerimpulse an, als solche auf höheren Ebenen. Auf der höchsten Ebene befinden sich Aufwendige Tätigkeiten, werden sie nicht aktiviert greifen die Verhaltensmuster wie sie im Kapitel „Verhalten 1“ beschrieben wurden.
Möchte man zum Beispiel dass der Bot eine Slalom-Kurs abfährt, kann man in anweisen einfache Verhalten zu kombinieren. Geradeaus fahren, um 90° drehen, wieder gerade aus, um -90° drehen usw. Man kann aber auch eine Mischung aus mehreren Einzelbewegungen verwenden. Soll der Bot einen Bogen fahren, ist das eine Mischung aus geradeaus fahren und Kurve fahren. Je nachdem wie das Verhältnis zwischen geradeaus und Kurve ist fährt der Bot einen weiten Bogen oder dreht sich auf der Stelle. Kombiniert man auf diese Weise einfache Verhaltensmuster kann man aus wenigen Elementen viele Verhaltensmuster erzeugen.
Ein Gutes Beispiel für die unterste Ebene des Subsumptions-Prinzip ist das Kurven fahren, so wie es die Methode bot_drive() umsetzt. Dabei wird nur eine Geschwindigkeit und ein Kurven-Wert übergeben. Der Kurven-Wert bewegt sich zwischen -127 (scharf links) über 0 (geradeaus) bis 127 (scharf rechts). Dabei wird auch die Methode bot_avoid_harm() verwendet, die den Bot davor schützt gegen Gegenstände und über Abgründe zu fahren. Sie liefert „Wahr“ zurück, wenn sich der Bot in Gefahr befinden und „Falsch“ wenn keine Gefahr droht.

```
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;
  }
}
```

In den höheren Ebenen der Subsumptions-Architektur befinden sich Regeln die nicht die Hardware direkt ansprechen sonder andere Regeln aufrufen. Damit entsteht eine Verkettung von Regeln deren Ausführungszeit steigt. Damit der Bot aber immer noch schnell genug reagieren kann, wird in einem Abstand von zehn und 20 Millisekunden die Methode bot_behave() aufgerufen. Diese Methode wurde bereits in „Verhalten 1“ beschrieben.
Das folgende Beispiel soll das Subsumptions-Prinzip auf einer höheren Ebene erklären. Aufgabe des Bot ist es, eine Lichtquelle zu finden. Hat er sie gefunden, soll er damit

beginnen eine Slalom-Kurs zwischen den weiteren Lichtquellen zu fahren. Zuerst muss der Bot eine Lichtquelle finden, dazu fährt er in eine Richtung los bis er auf eine Wand trifft. Er fährt ein Stück an der Wand entlang, dreht sich um 90° und fährt in einem Kreisbogen von der Wand weg. Fährt er auf eine Wand zu, fährt er wieder ein Stück an der Wand entlang, diesmal in die andere Richtung, dreht sich um 90° und beginnt eine Kreisbogen zu fahren, in die andere Richtung. Dieser Vorgang wird so lange wiederholt bis der Bot eine Lichtquelle gefunden hat, dann wechselt er in den Modus Slalom. In diesem Modus fährt der Bot auf die Lichtquelle zu, bis er kurz vor der Lichtquelle steht. Dann startet ein neuer Modus in dem der Bot um die Lichtquelle herum fährt und nach entfernten Lichtquellen sucht. Hat er eine gefunden wechselt er wieder in den Slalom-Modus und fährt auf die neue Lichtquelle zu. Bei all den unterschiedlichen Modi verwendet der Bot die oben beschriebene Methode bot_drive(). [ct0207]

3.5.Positionsbestimmung

Der Bot hat bisher nur auf seine Umwelt reagiert, dabei konnte er nicht seinen Blickwinkel und seine Position ermitteln. Mit Hilfe des Maus-Sensors und der Rad-Encoder kann man den Bot so programmieren, dass er seine relative Position bestimmen kann. Dazu sind nur ein paar Zeilen Quelltext und etwas Mathematik nötig. Der Maus-Sensor hat eine geringen Abtastzeit (alle 661 Mikrosekunden ermittelt der Sensor die Position). Der Mikrocontroller kann jedoch nicht alle 661 Mikrosekunden die Daten des Maus-Sensors abfragen, deshalb besitzt der Maus-Sensor eine eigene Logik, die die Positionsveränderung auf kumuliert. So kann der Mikrocontroller in größeren Intervallen die Positionsveränderung in Form einer x- und einer y-Koordinate abfragen. Nach jedem Abfragen durch den Mikrocontroller werden die internen Koordinaten des Maus-Sensors auf 0 gesetzt. So erhält man immer die Veränderung zwischen zwei Abfragezyklen.
Das der Bot sich nicht seitlich bewegen kann, ist ein Vorteil, so kann der exzentrisch eingebaute Maus-Sensor jede Richtungsänderung als Drehbewegungen wahrnehmen. Aufgrund der geringen Abtastzeit kann jede Bewegung als Fortbewegung auf einer Geraden angenommen werden, selbst wenn sich der Bot mit maximaler Geschwindigkeit im Kreis dreht.
Um die Veränderte Blickrichtung des Bot zu bestimmen wird die Differenz zwischen dem aktuellen Maus-Sensor-Wert und den zuletzt gemessenen berechnet. Für die Blickrichtung ist nur die X-Koordinate entscheidend.

```
dX=sensMouseX - lastMouseX;
dHead=(float)dX*360.0/MOUSE_FULL_TURN; // MOUSE_FULL_TURN: Mausaenderung in X-Richtung
                                       // fuer einen vollen Kreis
heading_mou+=dHead;
```

In der Variable heading_mou steht die aktuelle Blickrichtung in Grad. So wie die Berechnung ausgeführt wird, muss der Wert nicht zwischen 0° und 360° liegen. Deshalb muss noch eine Korrektur vorgenommen werden, damit der Wert von heading_mou im gewünschten Wertebereich liegt.

```
while (heading_mou>=359)
  heading_mou-= 360.0;
while (heading_mou<0)
  heading_mou+=360.0
```

An dieser Stelle ist er erforderlich mit Gleitpunktzahlen zu rechnen. Für den Mikrocontroller ist dies zwar sehr aufwendig, aber an dieser Stelle kann aus Gründen der Genauigkeit nicht mit ganzzahligen Werten gerechnet werden.
Mit Hilfe des aktuellen Winkels und der gefahrenen Strecke, kann man die Position des Bot bestimmen. Natürlich kennt der Bot nicht seinen absoluten Standpunkt, sondern nur den relativen, bezogen auf den Punkt, an dem er eingeschaltet wurde. In den Variablen

x_mou und x_mou speichert der Bot seine Positionsänderung in Millimeter. Da der Maus-Sensor intern in Zoll arbeitet, muss den Faktor 25.4 zur Umrechnung verwenden.

```
dY=sensMousY-lastMouseY;
lastDistance+=dY*25.4/MOUSE_DPI;
dPos=(float)dY*cos(heading*PI/180)*25.4/MOUSE_CPI; // CPI-Wert aus Kalibrierung
y_mou+=dPos;
dPos=(float)dY*sin(heading*PI/180)*25.4/MOUSE_CPI;
x_mou+=dPos;
```

Die so gewonnenen Informationen über die Position des Bot kann man zum einen dafür verwenden um den Bot systematisch ein Gebiet erkunden zu lassen. Selbst dann wenn die Akkus des Bot zwischenzeitlich nachgeladen werden muss und der Bot deshalb die Suche Abbricht. Mit den Verhaltensregeln, die im vorangegangenen Kapitel beschrieben wurden, kann dies leicht realisiert werden. Des weiteren kann man über die Positionsänderung auch die Geschwindigkeit des Bot ermitteln, da die Methode zur Positionsbestimmung alle 500 Millisekunden aufgerufen wird.
Da der Maus-Sensor je nach Bodenbeschaffenheit unterschiedlich genaue Werte liefert, kann man sich nicht alleine darauf verlassen. Um die Geschwindigkeit und Postion zu verifizieren, muss man auch die Werte in die Berechnung mit einbeziehen, die die Rad-Encoder liefern. Hier wird man experimentieren müssen, wie man die berechneten Werte der unterschiedlichen Sensoren gewichtet um eine gute Positionsbestimmung zu erreichen. [ct0213]

4 Programmierung des Sim

In diesem Kapitel wird der Aufbau des Sim beschrieben. Es wird gezeigt wie der Sim erweitert werden kann um die Programmierung des Bot zu testen.

4.1. Aufbau

Der Sim ist eine Testumgebung um den Steuercode des Bot auszuführen. Mit Hilfe des Sim kann man testen ob die Programmierung des Bot das gewünschte Verhalten zeigt. Beim Sim handelt es sich aber nicht um ein reines Testwerkzeug, sondern um einen Simulator, der das Verhalten des Bot graphisch veranschaulicht. Man sieht wie der Bot sich fortbewegt und ob er es zum Beispiel schafft um eine Ecke herum zu fahren oder an der Kante hängen bleibt.
Die graphische Oberfläche des Sim besteht aus zwei Fenstern. Das eine Fenster zeigt die virtuelle Welt in der sich der Bot bewegt. Das andere Fenster zeigt die Sensorwerte und die Motoreinstellung sowie ein Zahlenfeld, über das man dem Bot Befehle senden kann, um sein Verhalten zu ändern. Hier hat man auch die Möglichkeit die Simulation anzuhalten, um sich die Sensorwerte genau anzusehen, und fortzusetzen. Im Gegenzug kann ein Zeitraffer aktiviert werden, der Aufschluss darüber gibt, ob der Bot irgendwann ans Ziel kommt. All diese Möglichkeiten sind wichtige Hilfsmittel um Fehler in der Programmierung des Bot zu finden.
Um eine Stimulationssitzung zu starten, muss man zuerst den Sim starten. Danach startet man den Steuercode des Bot, der für den Rechner übersetzt wurde. Der virtuelle Bot meldet sich über eine TCP/IP-Verbindung am Sim an. Ab diesem Zeitpunkt erscheint das Symbol für den Bot im Fenster mit der virtuellen Welt und die Sensordaten die der Bot empfängt werden angezeigt. Damit der Sim dem Bot Sensordaten schicken kann, muss im Sim hinterlegt sein wie der Bot aufgebaut ist, des weiteren muss der Sim die Steuerimpulse des Bot interpretieren können und diese in Bewegungen in der virtuellen

Welt umrechnen. Im Gegenzug muss der Bot-Quelltext für die Steuerung nicht an den Sim angepasst werden. Zwar muss der Quelltext des Bot unterschiedlich übersetzt werden, je nachdem ob er auf dem Rechner oder auf dem Mikrocontroller ausgeführt werden soll, aber auf die Programmierung der Verhaltenslogik hat dies keine Auswirkung.
Mit dem Sim können mehrere Bots gleichzeitig verbunden sein. Auf diese Weise kann man Bots miteinander interagieren lassen und unterschiedliche Programmierungen im direkten Vergleich testen. Der Sim und jeder Bot läuft in einem eigenen Thread (Aktivitätsträger). Die Threads werden alle 10 Millisekunden synchronisiert um eine realistische Simulation zu gewährleisten.
Die simulierte Welt in der sich die Bots bewegen kann verändert werden. Dazu muss man verstehen wie der Sim aufgebaut ist und wie er programmiert ist. Der Sim wird in Java programmiert, es wird die Erweiterung Java3D verwendet, weil hier bereits mehrere Klassen vorliegen um virtuelle Welten leicht aufzubauen. Standardmäßig besteht in Java3D der Sinn zu testen ob sich Objekte überschneiden. Mit Hilfe dieser Methoden lässt sich leicht testen ob der Bot eine Wand berührt. [ct0205]

4.2. Datenmodell des Sim

Das Datenmodell des Sim teilt sich in zwei Pakete auf, ctSim.View und ctSim.Model. Im Paket ctSim.View wird die graphische Darstellung der Kontrolltafel und der virtuellen Welt implementiert. Das Paket ctSim.Model modelliert in der Klasse „World“ die Welt. Durch diese Trennung kann man eine eigene Sicht der virtuellen Welt programmieren, zum Beispiel aus dem Blickwinkel der vorderen Abstandssensoren des Bot, ohne dabei die Logik verändern zu müssen, die zum Beispiel Kollisionserkennung steuert. In beiden Paketen wird Java3D verwendet. Denn mit Java3D lässt sich die virtuelle Welt optisch darstellen und es lässt sich ein so genannter Szenegraph erzeugen, der alle räumlichen Objekte in der simulierten Welt verwaltet. Innerhalb des Szenegraphen werden die Bewegungen von Objekten ermittelt und Kollisionen erkannt. Das Paket ctSim.View greift nur auf den Szenegraph zu, um Veränderungen in der virtuellen Welt darzustellen.

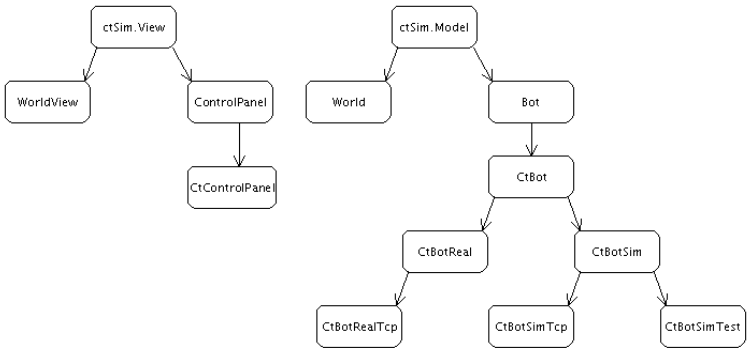

Abbildung 3: Datenmodell des c't-Sim

Neben der Klasse „World“ enthält das Paket ctSim.Model die abstrakte Klasse „Bot“ aus der die einzelnen Bot-Objekte abgeleitet werden. Hier werden die einzelnen Threads für die zu simulierenden Bots verwaltet. Jeder Thread besitzt eine run()-Methode, die über die Methode start() von außen aufgerufen wird. So sind die einzelnen Bot-Instanzen voneinander getrennt und können „parallel“ ausgeführt werden.
Die Klasse „Bot“ überschreibt die geerbte run()-Methode:

```
final public void run() {
```

```
  init();
  while(run == true) {
    work();
}
  cleanup();
}
```

Damit die Methode run() nicht von einer Unterklasse überschrieben werden kann, ist sie als final deklariert. Alle Bots müssen ihre Steuerungslogik auf die drei Methoden init(), work() und cleanup() verteilen. init() bereitet die Ausführung eines Bot vor. In der Methode work() wird die in C geschriebene Verhaltenslogik des Bot ausgeführt. Damit ein Bot beendet werden kann, gibt es die Methode „Bot.die()“, die die Variable run auf „false“ setzt und somit die Ausführung der Schleife beendet. Die Methode cleanup() sorgt dafür, dass der Thread korrekt beendet wird.
Die abstrakte Klasse „CtBot“ dient als Vorlage für den Bot. Hier befinden sich Informationen über Sensoren und (setzen Steuerimpulse in mechanische Bewegung um). Möchte man im Sim eine andere Art von Roboter simulieren, muss man auf dieser Ebene eine neue Klasse einfügen, die von „Bot“ erbt und den Aufbau des neuen Roboters beschreibt.
Um letztendlich eine Verbindung zwischen Sim und Bot herzustellen, wir ein Objekt der Klasse „CtBotSimTcp“ erzeugt. Diese Objekt kommuniziert über eine TCP/IP-Verbindung mit dem C-Programm des Bot. Die Methode work() ist dabei entscheidend, weil sie für die Synchronisation zwischen der Welt und eventuell anderen Bots sorgt. [ct0203]

4.3. Java3D

Wie bereits erwähnt, lässt sich Java3D sowohl für die graphische Darstellung der virtuellen Welt verwenden, wie auch für die Modellierung der Objekte. Alle Objekte der virtuellen Welt werden in einen hierarchischen Baum einsortiert, in dem jedes Element einen Zweig hat, zum Beispiel für Körper (Type Shape3D), Texturen, Lichtquellen und Blickpunkte. Dieser Baum wird Szenegraph genannt. Die Wurzel des Baums ist das virtuelle Universum, die Blätter sind alle sichtbaren Objekte, wie Wände, Boden und Bots. Die Knoten die zwischen Wurzel und Blättern liegen, sorgen für das Aussehen und Verhalten der Objekte.
Java3D hat einen Standardtyp „SimpleUniverse“ aus dem Paket com.sun.j3d.utils.universe. Hier muss nur noch der Zweig mit den sichtbaren Objekten eingefügt werden. Der Konstruktor der Klasse World() erzeugt ein „SimpleUniverse“ indem er World.createSzeneGraph() aufruft. Um leichter auf die unterschiedlichen Elemente zuzugreifen, werden für Boden, Lichtquellen und Hindernisse getrennte Zweige erzeugt.
Für die Bots wird mit CtBot.createBotShape() ein eigener Zweig erzeugt, welcher mit „obstBG“ bezeichnet wird. Hier gibt es eine Knoten für die Position (tg) und einen für Rotationen (rg). Der Bot Zweig ist Teil der Hindernisse, genauso wie Wände. Entscheidend für die Zuordnung ist die Eigenschaft, dass Wände und Bots einem Bot im Weg sein können, daher werden sie gleich behandelt.
Für jeden Bot gibt es einen eindeutigen Weg durch den Szenegraph. Alle Knoten die auf dem Weg durchlaufen werden wirken sich auf die Darstellung des Bots und seiner Position aus.
Möchte man den Abstand zwischen Objekten ermitteln, so muss man ein Objekt vom Typ „PickInof“ erzeugen. Dabei wird zwischen einem bestimmten Objekt und allen Objekten aus einem bestimmten Zweig des Szenegraphen die Schnittmenge bestimmte. Daraus lässt sich die Entfernung zwischen den einzelnen Objekten ermitteln. Der Sim generiert hieraus

die Daten für die Sensoren, die an das Bot-Programm geschickt werden. Aus diesen Daten lassen sich auch Kollisionen mit Wänden erkennen. Kollisionen zwischen Bots lassen sich so aber nicht erkennen. Damit den Abstand zwischen Objekten Bestimmen kann, müssen sie „greifbar“ sein. Um nun Kollisionen zwischen Bots zu erkennen, darf der Bot für den überprüft werden soll ob er mit anderen kollidiert nicht „greifbar“ sein. Der folgende Auszug aus dem Quelltext zeigt diesen Vorgang.

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

Man sieht wie der Test immer nur für ein Bot-Objekt durchgeführt werden kann. Deshalb müssen die Anweisungen in einem synchronized-Block stehen. Danach wird das Objekt von dem die Abstandsprüfung ausgehen soll als „nicht greifbar“ markiert. Danach wird dann der Abstand zu allen anderen ausgewählten Objekten ermittelt. Ist der Vorgang abgeschlossen wird das Objekt wieder als „greifbar“ markiert, damit es beim nächsten Test, der von einem anderen Bot ausgehen kann, erkannt wird. [ct0205]

4.4. Einfache Möglichkeit eine Welt zu erzeugen

Um für den Bot im Sim eine neue Welt zu bauen, gibt es zwei Möglichkeiten. Man kann den Aufbau der Welt fest in den Quelltext schreiben, dafür muss man sich mit Java3D auskennen. Die andere Möglichkeit ist, den im Sim integrierten Parcours-Loader zu verwenden. Im weiteren wird erklärt, wie man mit dem Parcours-Loader umgeht.
Der Parcours-Loader liest ASCII-Zeichen aus einer Textdatei. Jedes Zeichen steht für einen Gegenstand, der im Simulator dargestellt werden soll. Dabei ist die Fläche auf der die Welt aufgebaut werden soll mit einem Raster überzogen, dass erleichtert die Positionierung der einzelnen Elemente.
Um Wände und Lichtquellen zu positionieren reichen die ASCII-Zeichen aus. Möchte man aber auf die Beleuchtung oder die verwendeten Farben der Welt Einfluss nehmen, braucht man zusätzliche Beschreibungsmöglichkeiten. Mit Hilfe eines XML-Dokuments (Extensible Markup Language = erweiterbare Auszeichnungs-Sprache) können all diese Informationen in eine Textdatei geschrieben werden und vom Parcours-Loader gelesen und ausgewertet werden. In den unterschiedlichen Bereichen des XML-Dokuments werden werden die Elemente beschrieben und vom Parcours-Loader zu einer Welt zusammengefügt. Der Aufbau der Welt könnte zum Beispiel so aussehen:

```
<parcours>
        <line>===== Z ====#</line>
        <line>#     X   #O#</line>
        <line>#   X   * # #</line>
        <line># *   === # #</line>
        <line># =====     #</line>
        <line>#     *  ===#</line>
        <line>#=== ==     #</line>
        <line>#     X X   #</line>
        <line>==1=======2=#</line>
</parcours>
```

Wie die verwendeten ASCII-Zeichen interpretiert werden sollen, wird zum Beispiel so dargestellt:

```
<appearance type="X">
        <description>quadratische Wand</description>
        <texture>textures/rock_wall.jpg</texture>
        <color>#999999</color>
</appearance>
<appearance type="#">
        <description>senkrechte Wand</description>
        <clone>X</clone>
</appearance>
```

```
<appearance type="=">
        <description>wagrechte Wand</description>
        <clone>X</clone>
</appearance>
```

Hier ist zu sehen, das die verwendeten Wände gleich dargestellt werden, egal ob wie horizontal oder vertikal angeordnet sind. Zum besseren Unterscheiden beim gestalten der Welt ist es aber besser verschiedene Zeichen zu verwenden. Möchte man zum Beispiel das Aussehen der Wand ändern, so muss man nur beim <appearance type="X"> ein anderes Bild hinterlegen und schon sehen die Wände beim nächsten mal generieren anders aus.
Auf diese Weise lassen sich einfach individuelle Welten gestalten, ohne das man sich mit Java3D auseinander setzten muss. Erst wenn man den Parcours-Loader erweitern möchte, damit er weitere Zeichen interpretieren kann, muss man in Java3D programmieren und Anpassungen am Parcours-Loader vornehmen. [ct0211]

5 Fazit

Das „c't-Projekt - c't-Bot und c't-Sim" bietet viele unterschiedliche Möglichkeiten sich mit der Programmierung des Bot und Sim zu befassen. Durch den Quelltext, der zu Verfügung gestellt wird, hat man als Einsteiger die freie Wahl mit was man sich zuerst befassen möchte. Sowohl Bot als auch Sim lassen sich beliebig erweitern, so das dem Einsteiger erstmal keine Grenzen gesetzt sind.
Die Programmierung des Bot ist auf unterschiedliche Module verteilt. So findet sich auch der Einsteiger schnell zurecht. Auch der Sim ist strukturiert aufgebaut und kann vom Einsteiger schnell verstanden werden.
Der Bot ist kein ausgereiftes Produkt, das nach dem ersten Überspielen der Programmierung funktioniert. Zum Beispiel muss die Positionsbestimmung aufgrund der gefahrenen Strecke verbessert werden, damit der Bot eine Karte seiner Umgebung erstellen kann.
Der c't-Sim ist von Anfang an ein funktionierendes Programm. Um optimal damit arbeiten zu können, muss man den Sim erweitern, damit man Situationen Simulieren kann, auf die der Bot auch in der realen Welt treffen kann. Des weiteren muss man den Sim so erweitern, dass er dem Programmierer alle nötigen Daten liefert um Fehler in der Programmierung des Bot zu finden.
Bis Roboter so selbstständig werden wie die beiden berühmten Roboter aus „Star Wars" wird noch viel Zeit vergehen. Aber wenn man sieht, wie ein paar Zeilen Quelltext bewirken, glaubt man fast, dass gar nicht mehr so viel fehlt bis wir Roboter haben wie wir sie aus „Star Wars" kennen.

Literatur

[ct0202] Benz, B., Thiede, C., Thiele, T. (2006). Spielgefährten - Roboter für Löter, Simulator für Soft-Werker. c't, 2, 130-135.

[ct0203] Benz, B., König, P. (2006). Virtuelle Spielgefährten – Simulator für c't-Bots. c't, 3, 186-191.

[ct0205] Benz, B., König, P., Schwarten, L. (2006). Drängelnde Spielgefährten – Kollisionen und Sensoren für den c't-Sim, neues Verhalten für den c't-Bot. c't, 5, 224-230.

[ct0206] Benz, B. (2006). Nervensystem – Programmierung des c't-Bot von der Pike auf. c't, 6, 264-269.

[ct0207] Grimmer, C. (2006). Hohe Schule – c't-Bots bewältigen komplexe Aufgaben. c't, 7, 218-223.

[ct0209] Bachfeld, D. (2006). Steuermann – Mit einer Drehzahlregelung fährt der c't-Bot geradeaus und mit konstanter Geschwindigkeit. c't, 9, 222-226.

[ct0211] Benz, B. (2006). Genesis – c't-Sim: Weltenbau und Netzwerkzuschauer. c't, 11, 214-217.

[ct0213] Evers, T. (2006). Wo bin ich? - Positionsbestimmung für den c't-Bot. c't, 13, 226-229.

[ct0219] Birk, A. (2006). An der nächsten Ecke links... - Karten bauen (nicht nur) mit dem c't-Bot. c't, 19, 198-205.

[Bild1] http://www.heise.de/ct/ftp/projekte/ct-bot/bilder/bot3.jpg

[Bild2] http://www.ctbot.de/modules/mx_pafiledb/pafiledb/images/screenshots/CT-SIM.jpg