c't magazin für computer technik

Simulator für c't-Bots

Virtuelle Spielgefährten

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

Im letzten Heft fiel der Startschuss für den Selbstbau-Roboter c't-Bot. Mit dem maßgeschneiderten plattformübergreifenden Simulator c't-Sim können alle roboterbegeisterten Leser schon vor der Endmontage ihres eigenen Bot in die Programmierung der Steuerung einsteigen.

Manche Robotik-Fans möchten ihren selbst gelöteten c't-Bot furchtlos in die freie Wildbahn loslassen, obwohl seine Lötstellen noch nicht ganz kalt sind und der Compiler für den Mikrocontroller-Code mit Warnungen um sich wirft. Vorsichtigere Naturen dagegen wollen vielleicht erst einmal ausprobieren, ob die selbst gestrickte Steuerung auch wirklich so funktioniert, wie sie gedacht war - ein Vorzeichenfehler beim Auslesen des Bodensensors führt sonst schnell dazu, dass sich der mit viel Liebe aufgebaute maschinelle Spielgefährte entschlossen die Kellertreppe hinunterstürzt, anstatt vor dem Abgrund zurückzuschrecken.

Damit kleine Programmierfehler nicht solche großen Folgen haben, lohnt sich der ausführliche Test des Steuercodes, bevor dieser in den Roboter übertragen wird. Dazu bedarf es allerdings einer Umgebung, die den Routinen vorgaukelt, tatsächlich einen Bot zu steuern, der sich durch eine Welt voller Hindernisse bewegt. Eine solche Testumgebung bietet der Roboter-Simulator c't-Sim. Das Grundkonzept des Systems stellt dieser Artikel vor; die erste Version des c't-Sim steht auf der Projektseite zum Download bereit [1].

c't-Sim ist für Testläufe originalen Steuerungscodes für den c't-Bot gedacht: Der simulierte Bot ist mit Abstandssensoren ausgestattet, welche die Entfernung zum nächsten Hindernis messen, und seine virtuellen Motoren drehen unsichtbare Räder, die in guter Näherung an die physikalische Realität den Kurs bestimmen. Kollidiert er mit einem Hindernis, so bleibt der simulierte Bot stehen. Damit stehen alle Daten und Ansatzpunkte bereit, um beispielsweise eine clevere Hindernisvermeidung zu programmieren und ausführlich zu testen.

Eine spätere Ausbaustufe des c't-Sim soll auch als Steuerstand für echte c't-Bots dienen, die über Funk angebunden sind. Dann genügt ein Blick auf den Bildschirm, um einen Eindruck von den Zuständen der Sensoren und Aktuatoren zu bekommen, selbst wenn der kleine Geselle auf der Jagd nach Wollmäusen bereits unter dem Sofa verschwunden ist.

Wie der Roboter c't-Bot soll auch der c't-Sim als Plattform zu eigenen Experimenten und Weiterentwicklungen anregen. Die Simulation ist in Java geschrieben. Diese Sprache bringt viele vorgefertigte Hilfsklassen mit, die dem Programmierer manche mühsame Detail-Implementierung abnehmen. Auch komplexere Konzepte wie Threads oder sogar 3D-Grafik sind relativ leicht zugänglich und gut dokumentiert.

Ziel des Projekts c't-Sim ist, den einfachen Simulator Schritt für Schritt zu einer möglichst genauen Nachbildung des c't-Bots und seiner Umgebung auszubauen. Vertiefende Artikel in den folgenden c't-Ausgaben sollen zeigen, wie man Schritt für Schritt den eigenen Robotersimulator verfeinert oder andere Simulationen von Grund auf neu baut. Erweiterungsvorschläge und originelle Programme greifen wir gerne auf (s. Kasten „Erweiterungen“) und veröffentlichen sie auf der Projektseite [1]. Für Diskussionen rund um das Robotik-Projekt steht auf heise online ein Leserforum bereit. Eine ausführliche Dokumentation macht den Einstieg in den Code leicht, Details hierzu finden sich im Kasten „Dokumentation“ auf Seite 188.

Let’s sim!

Ein c't-Sim im Betrieb besteht aus mehreren gleichzeitig laufenden Teilsystemen: der Simulator-Umgebung ct-Sim und mindestens einem simulierten ct-Bot (s. Kasten „Der c't-Bot“). Das Java-Projekt ct-Sim steckt den Rahmen der Welt ab, in der sich die simulieren Bots tummeln; der C-Code des ct-Bot kapselt Steuerungsroutinen, die auch auf echten c't-Bots laufen, und kommuniziert per TCP/IP mit dem Simulator.

Beide Komponenten stehen auf der Projektseite [1] als ausführbare Dateien (ct-Bot.exe und ct-Sim.jar) zur Verfügung. Außerdem ist der gesamte Programmtext als ZIP-Datei und in einem CVS-Archiv erhältlich. Wie man diesen Code in der freien Entwicklungsumgebung Eclipse modifiziert und selbst kompiliert, beschreibt eine ausführliche Online-Anleitung.

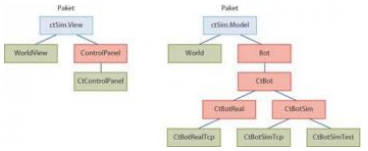
Die Simulationsumgebung ct-Sim.jar startet man auf der Kommandozeile über `java –jar ct–Sim.jar`. Sie öffnet ein Fenster mit der grafischen Darstellung der (noch leeren) Welt sowie ein zweites, das später für jeden Bot eine Kontrolltafel zeigt.

In einer zweiten Konsole startet man den ct-Bot. Dieser meldet sich über eine TCP/IP-Verbindung beim Simulator an, standardmäßig verwendet er die lokale IP (127.0.0.1) auf Port 10001. Sein roter Avatar taucht innerhalb der simulierten Umwelt auf, im anderen Fenster erscheint eine Kontrolltafel, auf der die Zustände der IR-Sensoren und der Motoren sowie Position und Blickrichtung des Bot als Zahlen angezeigt werden. Werte, die der Benutzer zur Laufzeit direkt beeinflussen kann, verfügen außerdem über einen Schieberegler. Wird die Checkbox „setzen“ aktiviert, überschreibt der eingestellte Wert den aktuellen Zustand des Bot. Möchte man solche Einstellungen nicht im laufenden Betrieb vornehmen, unterbricht die „Pause“-Taste die Simulation vorübergehend.

Einzeln verpackt

Unter seiner grafischen Bedienoberfläche teilt sich das Java-Projekt ct-Sim intern in drei Packages, was die in der Software-Entwicklung übliche konzeptionelle Trennung der Darstellung (View) von der eigentlichen Modellierung (Model) und von Kontrollelementen (Control) widerspiegelt.

Das Paket `ctSim.View` sorgt für die grafischen Darstellungen der Kontroll-Panele und der künstlichen Welt. Hierfür kommen Klassen aus dem API (Application Programming Interface) Java3D zum Einsatz. Die fertigen Klassenbibliotheken ermöglichen es, mit wenigen Zeilen Code eine eigene dreidimensionale Welt aufzuspannen und bringen viele nützliche Methoden mit, die beispielsweise Objekte mit Texturen überziehen und diffuses oder gerichtetes Licht ins künstliche Universum bringen.



Die Pakete des c't-Sim trennen die Darstellung der Welt von der Modellierung. Abstrakte Klassen sind rot eingefärbt, aus grünen Klassen dagegen lassen sich Objekte erzeugen.

Die konzeptionelle Trennung der grafischen Darstellung der Welt vom Rest des Systems bietet die Möglichkeit, die Simulation auch mit alternativen View-Klassen auszustatten. So kann neben der Aufsicht auf die Welt in einem zweiten Fenster der subjektive „Blick" eines Bot auf seine Umgebung treten, ohne den eigentlichen Simulationsablauf zu beeinflussen. Java3D macht solche effektvollen Erweiterungen mit wenig Aufwand möglich.

Java3D kommt aber nicht nur bei der Darstellung der Welt zum Einsatz, sondern auch bei deren Modellierung: Die Klasse `World` im Paket `ctSim.Model` benutzt zu einem so genannten Szenegraphen zusammengesetzte räumliche Objekte, um die Hindernisse in der Welt zu speichern; auch Änderungen innerhalb des Szenegraphen (durch die Bewegung von Objekten) finden innerhalb des `World`-Modells statt. Praktischerweise bringt Java3D Methoden mit, die überprüfen, ob sich zwei Körper überschneiden, was sich beispielsweise für Kollisionserkennung verwenden lässt. Eine ausführliche Einführung in die vielseitige 3D-Bibliothek folgt in einem späteren Teil dieser Artikelserie.

Am eigenen Faden

Neben der Welt-Klasse enthält das Paket `ctSim.Model` noch eine ganze Hierarchie diverser Robotertypen. Sie erben alle von der Oberklasse Bot, diese ihrerseits erweitert den Java-Typ Thread. Threads besitzen eine `run()`–Methode, die über `start()` von außen aufgerufen wird und die das Objekt wie einen eigenen Prozess laufen lässt - jeder einzelne Bot kann also ungestört seine Routinen abarbeiten, ohne erst darauf warten zu müssen, dass zuvor alle seine Kollegen ihre Arbeit beendet haben.

`Bot` überschreibt die geerbte `run()`-Methode:

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

Diese Methode ist als `final` deklariert, deshalb kann sie in keiner Unterklasse überschrieben werden. Alle Bots müssen ihre Steuerungslogik daher auf die drei aufgerufenen Routinen verteilen:

Wird der Bot gestartet, sorgt `init()` zunächst für die letzten Vorbereitungen, bevor der Bot in einer Schleife immer wieder seine eigene Methode `work()` aufruft - solange das Flag `run` den Wert true aufweist. `work()` enthält die Statements, die das Verhalten des Bots bestimmen. Die Methode `Bot.die()` setzt `run` auf `false`, die Schleife wird unterbrochen, `cleanup()` schließlich beendet den Bot-Prozess.

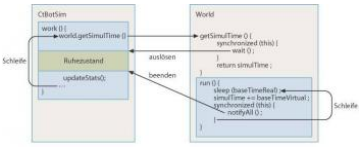
Für diese Grundstruktur sorgt bereits der oberste `Bot` der Hierarchie. Die direkte Unterklasse `CtBot` konkretisiert den abstrakten `Bot` zu einem Abbild des c't-Bot, indem sie Felder für die Zustände der speziellen Sensoren und Aktuatoren hinzufügt. Will man später auch Roboter anderer Architektur in unserem Simulator nachbilden, kann man einen weiteren Vererbungszweig von Bot ableiten. Veränderungen beim Controller oder in der Welt sind dabei nicht fällig - die Klassen arbeiten intern mit dem generischen `Bot`, nicht konkret mit einer seiner Spezialisierungen.

Startet man für die Simulation einen in C programmierten ct-Bot wie oben beschrieben, so stellt ihm ct-Sim ein Gegenstück in Form eines `CtBotSimTcp`-Objekts zur Verfügung, über das die TCP/IP-Kommunikation abgewickelt wird. Setzt man dagegen im Controller das Flag `test` auf `true`, so wird ein Roboter vom Typ `CtBotSimTest` zum Leben erweckt. Mit Bots dieser Klasse ist ein Betrieb des Simulators auch komplett ohne die C-Komponente ct-Bot des c't-Sim möglich. Eigenen Steuerungscode fügt man einfach in die Methode `CtBotSimTest.work()` ein. Allerdings darf der Code keine Schleife enthalten, da `work()` selbst immer wieder innerhalb einer Schleife aufs Neue aufgerufen wird. Weiterhin ist darauf zu achten, dass mit `super.work()` zusätzlich die gleichnamige Methode der Oberklasse CtBotSim aufgerufen wird.

Fragen des Takts

Wie alle Computersimulationen arbeitet auch c't-Sim mit diskreten Zeitpunkten. Dazu wird die Systemzeit im Rechner in Abschnitte der Länge `baseTimeReal` (derzeit 10 ms) unterteilt. Die World-Klasse ist ein Thread, der in seiner `run`-Methode die Länge der `baseTimeReal` abwartet und dann den Zähler für die Simulationszeit `simulTime` um die festgelegte Taktzeit des Simulators `baseTimeVirtual` erhöht (aktuell ebenfalls auf 10 ms gesetzt). Verändert man diese virtuelle Zeiteinheit, so kann der Simulator auch im Zeitraffer oder in Zeitlupe betrieben werden.

Über `World.getSimulTime()` findet die Synchronisation der verschiedenen `Bot`-Threads mit der World-Zeit statt (s. Abbildung links). In dieser Methode ist die Welt selbst als Synchronisationsobjekt deklariert (`synchronized (this)`). Fragen Bots aus der Methode `CtBotSim.work()` heraus die Zeit an, so werden sie durch `wait()` vorläufig in den Ruhezustand versetzt.



Alles eine Frage der Synchronisation: Die simulierte Welt achtet darauf, dass kein Bot der virtuellen Zeit davonläuft.

Eimal pro Durchlauf ihrer `run()`-Methode erlöst die Welt alle ruhenden Threads aus ihrer Starre (über `notifyAll()`). Dadurch wird sichergestellt, dass jeder Bot pro Simulatortakt seine `work()`-Routine maximal einmal durchführen kann.

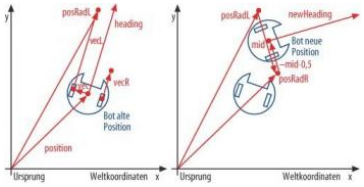
Action!

Der fleißige Gebrauch von `BotSim.work()` ist aber noch aus einem anderen Grund wichtig. Intern wird hier unter anderem die Methode `updateStats()` aufgerufen, die einen großen Teil der eigentlichen Simulation von Roboter-Aktionen modelliert. Sie sorgt unter anderem für die Berechnung der neuen Position des Bot, die im Folgenden genauer beschrieben wird. Dabei kommt etwas Vektormathematik ins Spiel (s. Abbildungen auf S. 191); glücklicherweise bietet die Bibliothek `javax.vecmath` hierfür die passenden Klassen und Methoden wie Vektoraddition.

Die Abrollstrecke des linken Rades im aktuellen Zeitabschnitt der Simulation errechnet sich aus dem PWM-Signal (Pulsweitenmodulation, s. [2]), das aktuell am linken Motor anliegt:

```
double turnsL = calculateWheelSpeed(getAktMotL()); turnsL= turnsL * deltaT / 1000.0f ;
```

Die Hilfsmethode `calculateWheelSpeed` berechnet die Anzahl der Radumdrehungen pro Sekunde. Momentan arbeitet die Umrechnung noch linear, zukünftige Verfeinerungen können hier bessere Näherungen an die Kennlinie des echten Motors bieten. `deltaT` ist die seit dem letzten Simulationsschritt verstrichene Zeit in Millisekunden, somit enthält `turnsL` die Zahl der noch nicht verarbeiteten Umdrehungen des linken Rades.



Ein paar Vektor-Operationen berechnen die neue Position des Bot in Abhängigkeit von der Geschwindigkeit seiner beiden Motoren.

`vecL` ist der Vektor, der die alte Position des Rads mit seiner neuen verbindet (s. Abbildung oben):

```
Vector3f vecL = new Vector3f(getHeading()); vecL.scale((turnsL * RAD_UMFANG),vecL);
```

Anschließend wird die relative Position des Rads zur aktuellen Position des Bot mit Hilfe eines Vektors berechnet, der senkrecht auf der Blickrichtung `heading` steht:

```
Vector3f vec = new Vector3f(getHeading().y,–getHeading().x,0f); vec.scale(RAD_ABSTAND,vec);
```

Der Vektor vom Ursprung des Koordinatensystems zur neuen Position des linken Rads ergibt sich aus der Addition dreier Vektoren:

```
Vector3f posRadL = new Vector3f(getPos()); posRadL.add(vec); posRadL.add(vecL);
```

Die neue Position des rechten Rads wird analog berechnet; aus der Mitte der beiden Radpositionen ergibt sich die neue `Bot`-Position:

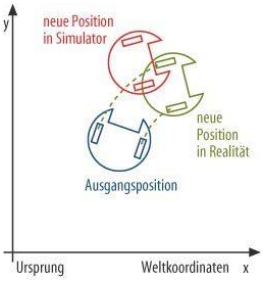
```
Vector3f mid = new Vector3f(posRadR); mid.sub(posRadL); mid.scale(0.5f,mid); Vector3f newPos = new Vector3f(po
```

Abschließend wird die neue Blickrichtung berechnet:

```
Vector3f newHeading = new Vector3f(–mid.y,mid.x,0); newHeading.normalize();
```

Der Einfachheit halber

Laufen die Motoren unterschiedlich schnell, fährt ein echter c't-Bot nicht geradeaus, sondern bewegt sich auf einem Kreisbogen. Daher bildet die Berechnung der neuen Position im Programm die Verhältnisse der realen Welt nicht ganz exakt ab, wie die Abbildung zeigt. Die Abweichung der Simulation fällt dabei umso größer aus, je länger der Abstand zwischen zwei Zeitpunkten ist, die das Näherungsverfahren überbrücken soll.



Je mehr Zeit die Simulation ihren Bots am Stück lässt, umso ungenauer gerät die Positionsberechnung - dank dem schnellen Takt des c't-Sim treten in der Praxis keine Probleme auf.

Das Herz des c't-Sim schlägt im 10-Millisekunden-Takt - in dieser Zeit rollen die Räder des Bot bei Höchstgeschwindigkeit nur rund 4,5 mm ab. Selbst wenn sich der Bot um die eigene Achse dreht, ist auf dieser Länge der vom Rad beschriebene Kreisbogen so wenig gekrümmt, dass die Abweichung des c't-Sim von der realen Welt an diesem Punkt als vernachlässigbar angesehen werden kann.

Der Rand der Welt

Jede Simulation kann nur einen Ausschnitt der realen Welt nachbilden, reduziert auf die wichtigsten Objekte und deren Eigenschaften. Wo die Grenzen dieses Weltausschnitts liegen, welche Gegenstände und Ereignisse als wichtig einzustufen sind und bis zu welchem Grad Prozesse vereinfacht werden können, hängt vom Zweck der Simulation ab. Die erste Version des c't-Sim testet bereits zuverlässig Steuerungsprogramme für den c't-Bot,

welche die beiden Abstandssensoren auslesen und daraus Befehle für die Motorensteuerung ableiten.

Der nächste Artikel der Serie zum c't-Bot wird sich wieder mit dem echten Roboter befassen - auf dem Programm stehen dann die Montage der Mechanik, der Aufbau der Platine und die ersten Gehversuche mit dem Mikrocontroller. Wem das Schrauben und Löten nicht liegt, dem bietet der c't-Sim viele Möglichkeiten, an der Tastatur aktiv zu werden. Das Gesamtsystem ist für weitere Anwendungen offen: Vielleicht begegnen sich eines Ta-ges auf einem simuliertem Spielfeld irgendwo im Netz ein Team aus Software-Robotern und eines aus virtuellen Stellvertretern echter c't-Bots zum Fußball-Match. (**pek[1]**)

Literatur

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

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

Soft-Link[4]

Der c't-Bot

Der Rahmen der Robotersimulation c't-Sim ist in Java geschrieben, was unter anderem die grafische Darstellung erleichtert. Die Nachbildungen der Roboter selbst laufen dagegen auf Basis von C-Code. Vorteil: Da der Mikrocontroller der realen c't-Bots in der gleichen Sprache programmiert wird, kommt später auch die Hardware des echten Roboters mit dem gleichen Steuerungscode klar.

Der Ordner ct-Bot aus dem Code-Archiv auf der Projektseite [1] sollte als komplettes Projekt in die Entwicklungsumgebung Eclipse importiert werden (Details siehe Online-Anleitung). Die Datei ct-Bot.c enthält das Hauptprogramm. Der Aufruf der Hauptmethode main() startet eine Endlosschleife, die jeweils nach zehn Millisekunden die Routine bot-behave() (zu finden in bot-logik.c) aufruft; in dieser Funktion findet eigener Steuerungscode seinen Platz. Der Zustand von Sensoren und Aktuatoren kann aus globalen Variablen (siehe bot_sens.h) ausgelesen werden, beispielsweise enthält sensDistL den Messwert des linken IR-Distanz-Sensors. Eine weiter gehende Beschreibung des Teilprojekts ct-Bot bleibt einer späteren Folge dieser Artikelserie vorbehalten.

Dokumentation

Der Quelltext des vorgestellten Robotersimulators c't-Sim ist ausführlich dokumentiert. Aus den Kommentaren im Source kann man bequem durchsuchbare HTML-Seiten generieren, die auch auf der Projektseite [1] zur Verfügung stehen.

Für den Java-Code benutzt man dazu das Tool Javadoc, das zum Lieferumfang des kostenlosen Java Development Kit (JDK) gehört. Der Eclipse-Menüpunkt „Project/Generate Javadoc“ erzeugt die HTML-Seiten im Unterverzeichnis `doc` des ct-Sim-Projekts.

Die Kommentare für den C-Source bereitet Doxygen auf - unter Linux gehört dieses Tool zur Standardinstallation, Windows-Nutzer müssen es nachrüsten. Als Frontend dient das Eclox-Plug-in für Eclipse (Details siehe Online-Installationsanleitung).

Erweiterungen

Der Roboter-Simulator c't-Sim ist „Work in Progress“. In der aktuellen Version verfügt er mit Motorsteuerung und Abstandssensoren bereits über die notwendigen Ansatzpunkte für Routinen, die aktiv verhindern, dass der Bot gegen die Wand fährt. Aktiven Programmierern bieten sich aber noch viele Möglichkeiten, das System zu erweitern und raffinierter arbeiten zu lassen.

Im Inneren des simulierten c't-Bot drehen sich bei der Positionsberechnung bereits virtuelle Räder, grafisch dargestellt werden sie allerdings noch nicht. Linien- und Abgrundsensoren harren ebenso einer cleveren Implementierung wie unterschiedliche Bodenfarben oder eine differenzierte Beleuchtung der Kunstwelt, auf die Lichtsensoren reagieren können.

Erweiterungen und Verfeinerungen können über die Projektseite [1] ausgetauscht und im Forum diskutiert werden. Damit eigene Entwicklungen für den c't-Sim zur aktuellen Codebasis passen, steht diese in Form eines CVS-Archivs zur Verfügung. Die Entwicklungsumgebung Eclipse bietet die Möglichkeit, bequem ein Repository des Versionierungssystems CVS auf einem entfernten Rechner einzubinden. Wie das genau geht, steht in der Anleitung auf der Projektseite.

Alle c't-Leser können das CVS-Repositiory lesen, die Schreibrechte sind allerdings nicht allgemein freigegeben. Eigener Erweiterungscode kann als Patch verpackt auch anderen Roboter-Enthusiasten zur Verfügung gestellt werden. Eine Anleitung hierfür liefert die Eclipse-Hilfe unter dem Schlagwort „Patch“.

Überzeugende Erweiterungen des Simulators, die uns als Patch erreichen, finden zeit-nah Aufnahme in die offizielle Codebasis. Wer also selbst Patches schreibt, sollte darauf achten, seinen c't-Sim vorher auf den aktuellen Stand des Repository zu bringen, damit keine Versionskonflikte entstehen.

Werkzeuge

Will man den Simulator möglichst schnell in Gang setzen, braucht man dazu nur eine Java-Laufzeitumgebung (s. Soft-Link) und die beiden ausführbaren Dateien `ct-Bot.exe` und ct-Sim.jar von der Projektseite [1]. Für Experimente am Code des Simulators und des Bots bietet sich die Entwicklungsumgebung Eclipse an, da sie kostenlos erhältlich ist. Eclipse unterstützt Java für den Simulator, C-Code für den simulierten Bot und später auch die Software-Entwicklung für Mikrocontroller. Eclipse ist selbst in Java geschrieben und läuft daher unabhängig von der Rechnerplattform.

Der C-Compiler gcc steht ebenfalls für beide Plattformen kostenlos zur Verfügung. MinGW und MSYS

versehen ihn unter Windows mit einer Unix-kompatiblen Umgebung; für Threads nach dem Posix-Standard sorgt die Bibliothek Pthreads. Daher kann unter Windows wie unter Linux der gleiche C-Code verwendet werden.

Der Soft-Link zu diesem Artikel führt Bezugsquellen zu allen benötigten Tools auf, eine genaue Installationsanleitung findet sich auf der Projektseite [1]. Manche Linux-Distribution bringt von Haus aus bereits das eine oder andere Werkzeug mit, Windows-Nutzer müssen dagegen mit Downloads im Gesamtumfang von rund 200 MByte rechnen.

Sind alle Werkzeuge installiert, erzeugt Eclipse über Project/Clean aus dem Code des ct-Bot eine ausführbare Datei (ct-Bot.exe unter Windows bzw. ct-Bot.elf unter Linux) im Unterverzeichnis Debug-Linux beziehungsweise Debug-Win32. Der ct-Sim läuft auch direkt innerhalb von Eclipse: Dazu startet man einfach die Klasse Controller aus dem Paket ctSim.Controller über Run as/Java Application.

URL dieses Artikels:
`http://www.heise.de/-290294`

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/Spielgefaehrten-290274.html`
[4] `http://www.heise.de/ct/06/03/links/186.shtml`