c't magazin für computer technik

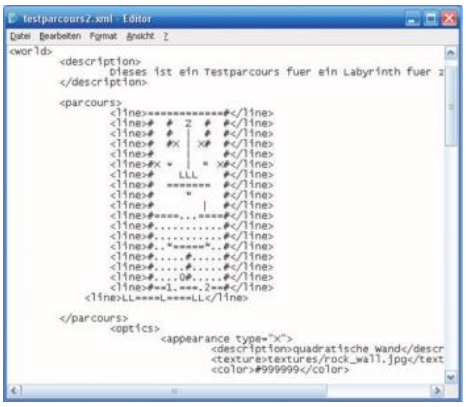
c't-Sim: Weltenbau und Netzwerkzuschauer

Genesis

Praxis & Tipps | Praxis .. Uhr
Benjamin Benz

Wenige ASCII-Zeichen reichen aus, um den Bots im c't-Sim eine eigene Welt zu bauen. Freunde können sich das Spektakel eines Bot-Rennens in der neu erschaffenen Umgebung nun auch aus der Ferne über das Internet anschauen.

Bisher bekamen die simulierten c't-Bots nur starre Welten zu Gesicht, die ihnen der Code fest vorschrieb. Wer die Welt verändern wollte, musste sich mit Interna von Java3D herumschlagen und Wände, Lampen sowie Abgründe von Hand eintragen. Genug damit! Der mit c't 10/06 still und heimlich in den c't-Sim eingeführte `ParcoursLoader` schafft Abhilfe. Er erzeugt die Welt für die Bots aus einer Konfigurationsdatei. Mit einem einfachen Texteditor kann nun jeder Bot-Gott spielen und Welten mit eigenen Labyrinthen, Rennstrecken oder virtuellen Wohnzimmern bauen.



Ein einfacher Texteditor reicht, um selbst eine Welt für die Bots zu bauen.

Das Erschaffen einer Welt ist denkbar einfach - sieben Tage wird dafür wohl niemand brauchen: Die Welt hat ein festes Raster (vierfacher Bot-Durchmesser) und setzt sich aus kleinen Quadraten zusammen. Jedes ASCII-Zeichen in der Weltdefinition legt fest, welches Objekt der ParcoursLoader auf das entsprechende Quadrat in der Bot-Welt stellt. So platziert man Wände, Fußböden, Lampen, Abgründe und Linien. Auf Bodenplatten vom Typ Startposition setzt der c't-Sim neu in die Welt eingefügte Bots. Die Tabelle unten zeigt, welche Grundmodule wir dem ParcoursLoader bereits mit auf den Weg gegeben haben. ASCII-Zeichen für eigene Erweiterungen sind noch reichlich frei.

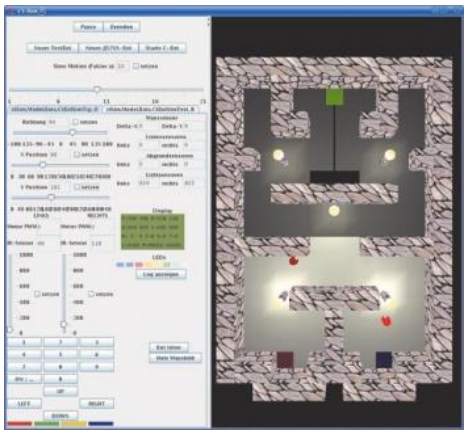
Ein solch schlichtes Modell bringt aber auch ein paar Einschränkungen mit sich: Alles spielt sich in einer Ebene mit starrem Raster ab, denn ASCII-Dateien sind zwar übersichtlich, aber nur zweidimensional. Den Szenegraphen, mit dessen Hilfe Java3D die Welt dreidimensional darstellt, baut dann der ParcoursLoader zusammen.

Es werde Licht!

Trist müssen die neuen Welten dennoch nicht sein. Zur Beschreibung einer Welt gehört nicht nur der ASCII-Parcours, sondern auch Angaben über Farben, Texturen und Lichtquellen. Damit all das nicht nur leicht zu bearbeiten, sondern auch erweiterbar und maschinenlesbar ist, haben wir XML als Dateiformat gewählt. Diese Dokumentenbeschreibungssprache, deren Dateien immer etwas an HTML-Code erinnern, eignet sich bestens für Konfigurationsdateien. Die Datei besteht - nach einem Header - aus einzelnen Blöcken, die von so genannten Tags eingerahmt werden. Ein Beispiel für eine solche Datei zeigt das Listing unten.

So fassen `world`-Tags die ganze Beschreibung ein und parcours-Tags die ASCII-Definition der Welt. Da sich jede Zeile zwischen line-Tags befindet, kann ein Standard-XML-Parser die Datei einlesen. Ob sie nun mit Windows- oder Unix-Zeilenumbrüchen arbeitet, spielt keine Rolle, der Zeichensatz (in diesem Beispiel UTF-8) ebenso wenig. Damit der Parser es einfach hat, eine effiziente Welt - die aus möglichst wenigen Elementen besteht - zu bauen, gibt es neben dem X für quadratische Wände noch die Zeichen = und #. Sie geben dem Parser die Richtung vor, in der er Wandelemente zusammenfassen darf.

Wer eine eigene Welt erschaffen will, kopiert einfach die Musterdatei und bearbeitet erst einmal nur die



Das komplette Labyrinth inklusive der Startpositionen für die Bots lädt der c't-Sim aus einer XML-Datei.

Einträge zwischen den line-Tags.

Auf gleicher Hierarchie-Ebene mit description und parcours steht der optics-Block (Zeile 20), der festlegt, wie der ParcoursLoader die einzelnen ASCII-Zeichen umsetzt. Für jedes im parcours-Block verwendete Symbol gibt es hier die Definition des Erscheinungsbildes (Appearance, Zeile 21).

Die Datei für die Textur (Zeile 23) sollte sich im Ordner „textures“ des Projektverzeichnisses befinden, sonst meckert später der Parser. Da man immer wieder verschiedene Objekte braucht, die aber gleich aussehen sollen, kann man mit dem Tag clone eine andere Appearance kopieren (Zeile 27).

Elemente wie Fußbodenplatten könnte man zwar auch mit Texturen belegen, oft reicht aber schon die Angabe der gewünschten Farben als hexadezimale RGB-Werte (Zeile 31).

Java3D unterscheidet vier Typen von Farben: ambient beschreibt die Farbe eines indirekt beleuchteten Objekts. Die diffuse-Farbe hingegen verwendet Java3D für gerichtete Lichtquellen. Mit specular kann man die Farbe von glänzenden Reflexionen beeinflussen. Bei einer Lichtquelle kommt emmissive zum Zuge (Zeile 36). Setzt man nicht alle Farben per Hand, so verwendet der ParcoursLoader die Java3D-Standards. Für bestimmte Objekte - wie die mattschwarzen Linien auf den Bodenplatten - taugen die Standards (grau für ambient, weiß für diffuse und specular sowie schwarz für emmisive) jedoch nicht, da sie sonst immer noch leicht glänzen.

Setup

Nicht nur die Welt, sondern auch alle anderen Parameter lädt der c't-Sim beim Start aus einer Konfigurationsdatei. Im Ordner „config“ befindet sich eine weitere XML-Datei namens ct-sim.xml. Sie enthält eine ganze Reihe von parameter -Definitionen:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE collection SYSTEM "config.dtd">
<ct-sim>
 <description>c't-Sim-cinfig  </description>
 <parameter name="botport" value="10001"/>
 <parameter name="parcours"
            value="parcours/testparcours2.xml"/>
 ...
</ct-sim>
```

Sämtliche Einträge stehen später im Controller in dem Objekt config vom Typ HashMap zur Verfügung. Außerhalb des Controllers sollte ein Zugriff auf globale Parameter nicht nötig sein [3]. HashMap-Objekte speichern ein Tupel, bestehend aus Schlüssel und Wert. Als Schlüssel dient das name-Feld, als Wert der Eintrag unter value. Möchte man im Code beispielsweise auslesen, auf welchem Port der c't-Sim nach Bots lauschen soll, so schreibt man:

String portString = (String)config.get("botport"); Eine einfache untypisierte HashMap verwaltet nur Objekte vom abstrakten Java-Typ Object, daher ist der Type-Cast in ein String-Objekt nötig. Nun muss man aber noch die Zeichenkette aus der Konfigurationsdatei in eine Zahl verwandeln:

```
int port=0;
if (portString != null){
 try {
   port = Integer.valueOf(portString).intValue();
 } catch (NumberFormatException ex ){
   ErrorHandler.error("Kann "+portString+" nicht konvertieren. "+ex);
 }
} else
 ErrorHandler.error("Kein Eintrag viewport"+" in der Konfiguration");
```

ErrorHandler.error("Kein Eintrag viewport"+" in der Konfiguration"); Obiger Code prüft, ob es überhaupt einen passenden Eintrag in der Konfigurationsdatei gibt und ob sich dieser in eine Zahl umwandeln lässt.

Möchte man auf eine Datei zugreifen, deren Name in der Konfiguration steht, ist ein kleiner Trick nötig:

```
String parcours = ClassLoader.getSystemResource(´(String)config.get("parcours")).toString();
```

Der Umweg über die Methode Classloader.getSystemResource stellt sicher, dass Java die Datei auch dann findet, wenn sie sich innerhalb eines Jar-Archives befindet.

Wer es leid ist, C-Bots per Kommandozeile zu starten oder sich jedes Mal mit einem Öffnen-Dialog herumzuschlagen, verrät dem c't-Sim in der Config-Datei einfach den Ort, an dem das Binary liegt:

```
<parameter name="botbinary" value="../ct-Bot/Debug-Linux/ct-Bot.elf"/>
```

Ein Druck auf den Knopf „Starte C-Bot“, und der c't-Sim ruft den Bot automatisch auf.

Bot-Styling

Setzt man mehr als einen Bot in ein Labyrinth, so möchte man die kleinen Racker gerne unterscheiden können. Dafür bietet ct-Sim.xml eine eigene Rubrik

```
<bots>
```

, in der jedem Bot ein eigenes Styling für verschiedene Zustände zugeordnet wird:

```
<bots>
 <bot name="ctSim.Model.Bots.CtBotSimTest_0">
   <appearance type="normal">
     <description>Standard für TestBot 0</description>
     <color>#FF0000</color>
   </appearance>
   <appearance type="collision">
     <description>TestBot 1 bei Kollision</description>
     <color>#800000</color>
   </appearance>
   <appearance type="falling">
   ..
   </appearance>
 </bot>
 <bot name="ctSim.Model.Bots.CtBotSimTcp_0">
   ...
 </bot>
 ...
</bots>
```

Die Namen der Bots setzen sich - wie mittlerweile überall im Sim - aus dem Klassennamen und einer angehängten Nummer zusammen. Fährt der Bot normal, so zeigt er sein Standardaussehen (`normal`). Er kann jedoch auch vor Wut beispielsweise dunkelrot anlaufen, wenn er gegen eine Wand fährt (`collision`), oder vor Angst erblassen, wenn er in einen Abgrund fällt (`falling`).

Zu Gast bei Freunden

Zwei Bots stehen auf den Startfeldern, zucken nervös nach nach rechts und links und warten darauf, sich im soeben gebauten Labyrinth zu beweisen. Aber wo bleibt die Spannung, wenn man das Bot-Rennen allein anschauen muss? Klar, man könnte einen Bot-Abend organisieren, mit Beamer, Bier und reichlich Chips. Wer aber die Krümel auf dem Sofa scheut oder keinen Platz für eine solche Party hat, lädt die Bot-Gemeinde virtuell ein. In der aktuellen Code-Version lauscht der c't-Sim nicht nur auf einem TCP-Port (10001) nach Bots, er nimmt auf einem zweiten Port (10002) auch Anmeldungen von ClientViews entgegen:

```
ServerSocket server = new ServerSocket(port);
TcpConnection connection = new TcpConnection();
tcp.connect(server.accept());
```

`ClientView`-Objekte aus dem c't-Sim-Code besitzen eine `main`-Routine und lassen sich daher selbstständig starten. Sie bauen eine Verbindung zum c't-Sim-Server (`Controller`) auf:

```
TcpConnection connection = new TcpConnection();
connection.connect("ct-sim.freund.de",10002);
```

Dieser überträgt ihnen dann zuerst den vollständigen Java3D-Szenegraphen und sendet danach regelmäßig die Änderungen. Ein Szenegraph enthält alle Objekte, Lichtquellen, Texturen, Farbdefinitionen und Kamerapositionen, die Java3D benötigt, um die Roboterwelt darzustellen [5]. Er allein reicht aus, damit die ClientView ein (statisches) Bild der Welt rendern kann. Dieses enthält zwar bereits alle Bots, bewegen können sie sich aber erst, wenn der c't-Sim Updates für deren `TransformGroups` liefert. Für ClientViews sind Bots nichts weiter als geometrische Objekte, die auf Anforderung des c't-Sim verschoben oder gedreht werden. Daher simulieren ClientViews auch nichts selbst, sondern überlassen das dem zentralen c't-Sim.

In der Theorie ist der Transfer eines Szenegraphen über eine bereits etablierte TCP-Verbindung ganz einfach. Die Klasse `SceneGraphStreamWriter` überträgt einen Ausschnitt des Szenegraphen. Dieser hängt als Baum unterhalb der `BranchGroup scene`:

```
ServerSocket server = new ServerSocket(port);
TcpConnection connection = new TcpConnection();
tcp.connect(server.accept());
```

Und auf der Gegenseite nimmt ein `SceneGraphStreamReader` die Szene wieder entgegen:

```
SceneGraphStreamReader reader = new SceneGraphStreamReaderFixed(connection.getSocket().getInput
scene=reader.readBranchGraph(map);
```

In obigem Beispiel taucht eine Referenz auf `map` auf. Dabei handelt es sich um eine indexierte Liste (`HashMap`) mit Einsprungpunkten in den Szenegraphen. Die ClientViews benötigen diese Referenzen, um beispielsweise die `TransformGroup` wiederzufinden, die die Position eines Bots kontrolliert. Ohne sie würde der Bot immer an der Stelle stehen bleiben, an der er sich beim Start der ClientView befand. Die ganze Szene für jede Bewegung neu zu übertragen wäre pure (Zeit-)Verschwendung. Die Liste der Einsprungpunkte ermöglicht eine elegante Konstruktion: Der c't-Sim versorgt die ClientView bei jedem Update nur mit Änderungswünschen. Jeder Wunsch besteht aus dem Schlüssel, unter dem die ClientView das zu aktualisierende Objekt in der `map` findet, und einer Transformationsmatrix. Ob es sich dabei um einen Bot, ein bewegliches Hindernis oder einen anderen Gegenstand handelt, spielt keine Rolle. Neben Transformationen kann der c't-Sim auch neu hinzugekommene Bots übertragen oder bestehende aus der Welt entfernen lassen. Viel mehr Aktionen müssen ClientViews aber nicht beherrschen. Übrigens verwaltet auch der eigentliche c't-Sim mittlerweile alle Einsprungpunkte in den Szenegraphen in solchen HashMaps.

Der Teufel steckt jedoch auch hier im Detail. Die Klasse `SceneGraphStreamReader` aus dem Java3D-API und ihr Gegenpart haben leider einen Bug und verstümmeln die `map`, sodass wir uns gezwungen sahen, sie zu überschreiben. Daher instantiieren wir in obigem Beispiel auch von `SceneGraphStreamReaderFixed`. Des Weiteren darf man keine Szenegraphen übertragen, die bereits `alive` sind. Daher muss der c't-Sim immer einen Klon des angezeigten Graphen vorhalten. Wir haben deshalb eine neue Klasse `SceneLight` eingeführt. Sie enthält nur die für die Anzeige wesentlichen Daten: den Szenegraphen und die bereits erwähnte `HashMap` mit den Einsprungpunkten. Mit Methoden ist sie hingegen reichlich bestückt, da sie sowohl auf Server- als auch auf Client-Seite verwendet wird. So kümmert sie sich um das Eintragen von Updates, die wir in einem eigenen Typ `SceneUpdate` über das Netzwerk schicken.

Teilnehmen

An dieser Stelle bitten wir Sie nun um rege Mithilfe: Das Grundgerüst für die Netzwerk-Views steht; alle Bereiche, die eingehende Java3D-Kenntnisse erfordern, sind abgeschlossen. Damit kann man bereits aus der Ferne einem Bot-Rennen in 3D zuschauen. Jeder Zuschauer kann dabei seinen Blickpunkt und -winkel frei wählen. Es fehlen jedoch noch sämtliche Statusinformationen, die der c't-Sim sonst lokal in den Panels darstellt, von Interaktionsmöglichkeiten der Zuschauer ganz abgesehen. Auch ist das Netzwerkprotokoll bislang weder optimiert noch beherrscht es eine Benutzerauthentifizierung.

Auf der Mailingliste [1] hat bereits die erste Diskussion darüber begonnen, was für Fähigkeiten ClientViews haben sollten. Von einem Chat zwischen den Zuschauern, der Annahme von Wetten oder Boxeninformationen à la Formel 1 war bereits die Rede. Programmierer mit einem Händchen für grafische Oberflächen sind nun gefragt, die noch recht schlichten ClientViews in spannende Bot-Arenen zu verwandeln. Wer sich lieber mit Netzwerkschnittstellen beschäftigt, kann sich an einem erweiterbaren Protokoll zwischen c't-Sim und ClientView versuchen. Kenntnisse von Java 3D sind für keines von beidem nötig.

Wen nun die Lust gepackt hat, eigene Labyrinthe zu bauen oder sein Wohnzimmer für die Bots

ClientViews verbinden sich per Netzwerk mit dem c't-Sim und zeigen das Geschehen in der Bot-Welt an. Den Blickwinkel kann man dabei frei wählen.

nachzumodellieren, kann uns seine Welt-Definitionen gerne als Patch einschicken. Entweder nehmen wir sie direkt in den Code auf oder stellen sie zum Download bereit. Wie man einen Patch einreicht, beschreibt die Projektseite [1] ausführlich. Antworten auf viele häufig gestellte Fragen haben wir dort bereits in den FAQ gesammelt. Neben dem Link zum c't-Bot-Diskussionsforum findet sich auf der Projektseite auch die Anmeldung für die Mailingliste. Noch unmittelbarer kommt man per Chat (IRC) [1] mit anderen Entwicklern in Kontakt. (**bbe[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]**

[3] Benjamin Benz, Peter König; **Virtuelle Spielgefährten, Simulator für c't-Bots[4]**

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

[5] Benjamin Benz, Peter König, Lasse Schwarten; **Drängelnde Spielgefährten, Kollisionen und Sensoren für den c't-Sim, neues Verhalten für den c't-Bot[6]**

[6] Benjamin Benz, Thorsten Thiele; **An der Leine, Debuggen des c't-Bot über USB[7]**

[7] Christoph Grimmer; **Hohe Schule, c't-Bots bewältigen komplexe Aufgaben[8]**

Listing

Um eine eigene Welt für die Roboter zu bauen, kopiert man einfach die Muster-XML-Datei und bearbeitet die Abschnitte zwischen den line-Tags.

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <!DOCTYPE collection SYSTEM "parcours.dtd">
 3
 4 <world>
 5   <description>
 6     Labyrinth für einen einzelnen Bot
 7   </description>
 8
 9   <parcours>
10     <line>#============</line>
11     <line>#===        Z</line>
12     <line>#===      ==#</line>
13     ...
14     <line>#      ===  #</line>
15     <line>#..#        #</line>
16     <line>#..#   X    #</line>
17     <line>#1.=========#</line>
18   </parcours>
19
20   <optics>
21     <appearance type="X">
22       <description>quad. Wand</description>
23       <texture>
           textures/rock_wall.jpg
         </texture>
24     </appearance>
```

```
25     <appearance type="#">
26       <description>vert. Wand</description>
27       <clone>X</clone>
28     </appearance>
29     <appearance type=".">
30       <description>Boden weiß</description>
31       <color type="ambient">#FFFFFF</color>
32       <color type="diffuse">#FFFFFF</color>
33     </appearance>
34     <appearance type="*">
35       <description>Lichtkugel</description>
36       <color type="emmissive">
           #FFFF90
         </color>
37     </appearance>
38     <appearance type="-">
39       <description>Linie</description>
40       <color type="ambient">#000000</color>
41       <color type="diffuse">#000000</color>
42       <color type="specular">
           #000000
         </color>
43     </appearance>
44     ...
45   </optics>
46 </world>
```

Kammerjäger

Selbst beim ausgebufftesten Programmierer versteckt sich irgendwann der eine oder andere hartnäckige Bug im Code. So genau man auch auf den Quelltext starrt, gerade kleine Vorzeichen- oder Pointer-Fehler tarnen sich effektiv zwischen hunderten Zeilen korrekten Codes. Für PC-Software gibt es eine ganze Reihe an wirkungsvollen Schädlingsbekämpfern. Entwicklungsumgebungen wie Eclipse bieten komfortable Debugger, die das Programm Schritt für Schritt ausführen und jederzeit die Werte aller Variablen anzeigen können.

Anders sieht das Ganze aus, wenn der Code auf einem Mikrocontroller läuft. Zwar gibt es Emulatoren, die den ganzen Prozessor zyklusgenau simulieren, aber die realen Messwerte der Sensoren fehlen dann immer noch. Für den c't-Bot haben wir daher bereits in [6] den USB-2-Bot-Adapter vorgestellt, mit dem er Ausgaben an den c't-Sim weiterleiten kann. Dieser zeigt Sensorwerte und einige andere Parameter genau so an, wie man das auch vom simulierten Bot her gewohnt ist. Mit ein paar einfachen Befehlen - die der Leser Andreas Merkle implementiert hat - kann man aber noch viel mehr Informationen darstellen.

Schreibt man im c't-Bot-C-Code folgende Zeile:

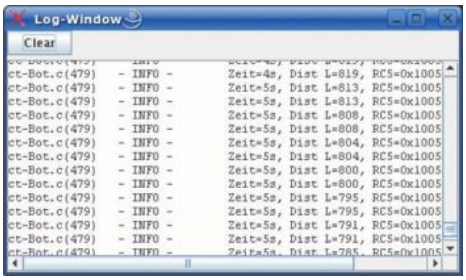
```
LOG_DEBUG(("Hallo Welt!"));
```

nimmt das Framework die Meldung zwischen den doppelten Klammern entgegen und leitet sie an eines von drei Log-Zielen weiter. Je nach den Optionen in der Datei ct-Bot.h erscheint die Ausgabe auf dem Display des Bots (`LOG_DISPLAY_AVAILABLE`), im Log-Fenster des c't-Sim (`LOG_CTSIM_AVAILABLE`) oder im Klartext auf der seriellen Schnittstelle (`LOG_UART_AVAILABLE`). Da es auf dem Display mit nur 4 x 20 Zeichen etwas beengt zugeht, ist hier die Ausgabe recht spartanisch: In jeder Zeile steht eine Meldung, ist das Display voll, wird wieder von oben nach unten überschrieben. Ein Cursor vor der Meldung markiert die zuletzt eingegangene.

Auf der seriellen Schnittstelle und im c't-Sim gibt es dagegen keine Platzbeschränkung. Stünde der obige Log-Befehl in der Datei bot-logic.c in Zeile 666, so erscheint:

```
bot-logic.c(666)     - DEBUG -     Hallo Welt!
```

Ohne dass man Zeilennummer und Datei per Hand angeben muss, erfährt man sofort, woher die Meldung stammt. Zur Unterscheidung verschiedener Typen gibt es nicht nur `LOG_DEBUG`, sondern `LOG_INFO`, `LOG_WARN`, `LOG_ERROR` und `LOG_FATAL`. Die Log-Kommandos können nicht nur feste Strings verarbeiten, sondern beherrschen dieselbe Syntax wie `printf`. Einziger Unterschied: Die doppelten Klammern sind



Der c't-Sim zeigt die Log-Ausgaben von realen und simulierten Bots inklusive Dateinamen und Zeilennummern an.

nötig, da es sich nicht um einen Funktionsaufruf, sondern ein Preprozessor-Makro handelt.

```
LOG_INFO(("Zeit=%ds, Dist L=%d, RC5=0x%04x",timer_get_s(),sensDistL,RC5_Code));
```

liefert nicht nur die aktuelle Zeit in Sekunden, sondern auch den Wert des linken Abstandssensors und den zuletzt empfangenen RC5-Code (siehe Screenshot).

Ausführliche Beschreibungen zur Syntax von `printf` haben wir in der Linkliste auf der Projektseite [1] gesammelt.

Alle genannten Log-Befehle funktionieren übrigens nicht nur auf dem realen Roboter, sondern gelten genauso für den PC. Die Angabe von `LOG_UART_AVAILABLE` ergibt dort natürlich keinen Sinn, stattdessen lenkt `LOG_STDOUT_AVAILABLE` die Ausgaben auf die Konsole, die den Bot gestartet hat.

Zeichensatz

Zeichen	Bedeutung	Bild
X	Wand, 1 Quadrat	
#	senkrechte Wand	
=	waagerechte Wand	
1	Startpunkt für Bot 1	
2	Startpunkt für Bot 2	
0	Default-Startpunkt für Bots	
Z	Zielpunkt für die Bots	
*	Säule mit Lichtquelle	
|	senkrechte Linie	
-	waagerechte Linie	
\	Linie Ecke NE	
/	Linie Ecke NW	
~	Linie Ecke SE	
+	Linie Ecke SW	
	Fußboden normal	
.	Fußboden weiß	
L	Abgrund	

URL dieses Artikels:

http://www.heise.de/-290480

Links in diesem Artikel:

[1] mailto:bbe@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] https://www.heise.de/ct/artikel/Virtuelle-Spielgefaehrten-290294.html

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

[6] https://www.heise.de/ct/artikel/Draengelnde-Spielgefaehrten-290334.html

[7] https://www.heise.de/ct/artikel/Hohe-Schule-290392.html

[8] https://www.heise.de/ct/artikel/Hohe-Schule-290392.html