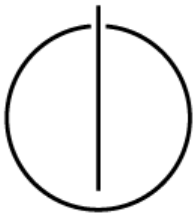
Technische Universität München

Fakultät für Informatik



Bachelor’s Thesis in Informatik

Implementierung einer Sensorik für virtuelle Fußgänger im Kontext der agentenbasierten Modellierung und Simulation

Christian Thieme

Technische Universität München

Fakultät für Informatik

Bachelor's Thesis in Informatik

Implementierung einer Sensorik für virtuelle Fußgänger im Kontext der agentenbasierten Modellierung und Simulation

Implementation of sensor technology for virtual pedestrians in the context of an agent-based modeling and simulation

Autor: Christian Thieme

Aufgabensteller: Univ.-Prof. Dr. Hans-Joachim Bungartz

Betreuer: M.Sc. M.Phil. Oliver Handel

Abgabedatum: 15.04.2016

Ich versichere, dass ich diese Bachelor's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

_____

Ort, Datum

_____

(Christian Thieme)

ABSTRACT

Pedestrian simulations are developed for several reasons. It's important though, that the simulation provides realistic results. At this, the modelling of human behavior is a big obstacle. Further, decisions made by humans are often influenced by the momentary perception. In course of this Bachelor's Thesis a sensor technology for virtual pedestrians is implemented. First, the general concept of agents and sensory information is explained. Then the development environment Anylogic and the case study are introduced. The implementation of a Java library for sensor functions was made in and for Anylogic using the pedestrian simulation and case study „Back to the Woods". The case study was also implemented in Anylogic and was provided for this work. The implementation of the sensor technology allows virtual persons to perceive their environment. The implemented sensor technology functions are limited to an appropriate extent for the case study. Agents or virtual pedestrians can then use the information gained by the sensor technology to bias their decision-making process.

KURZFASSUNG

Fußgängersimulationen werden aus unterschiedlichen Gründen entwickelt. Wichtig ist, dass die Simulation realitätsnahe Ergebnisse liefert. Hierbei ist die Modellierung von menschlichem Verhalten eine große Hürde. Des Weiteren sind Entscheidungen die Menschen treffen oftmals von der momentanen Wahrnehmung beeinflusst. Um die Wahrnehmung für virtuelle Fußgänger zu simulieren wird im Rahmen dieser Bachelorarbeit eine Sensorik für virtuelle Fußgänger implementiert. Zunächst werden allgemein das Agentenkonzept und sensorische Informationen erklärt. Dann werden die Entwicklungsumgebung Anylogic und die Fallstudie vorgestellt. Die Implementierung einer Java Bibliothek von sensorischen Funktionen in und für Anylogic erfolgte unter Verwendung der Fußgängersimulation und Fallstudie „Back to the Woods". Auch die Fallstudie wurde in Anylogic implementiert und für diese Arbeit zur Verfügung gestellt. Die Implementierung der Sensorik erlaubt es virtuellen Personen ihre Umgebung wahrzunehmen. Die implementierten sensorischen Funktionen beschränken sich auf einen für die Fallstudie angemessenen Umfang. Agenten bzw. virtuelle Fußgänger können die über die Sensorik gewonnenen Informationen dann in ihren Entscheidungsprozess einfließen lassen.

Inhalt

1 EINLEITUNG

Es besteht ein Bedarf für die Modellierung des Verhaltens von Fußgängern für einige Anwendungen. Darunter befinden sich beispielsweise Eventplanung, Ressourcenplanung oder Stadtplanung [13]. Das Verhalten von Fußgängern kann in Fußgängersimulationen modelliert werden. Oft spielt bei diesen Simulationen auch der Sicherheitsaspekt eine große Rolle. Eine hohe Menschendichte in bestimmten Bereichen, Institutionen oder Events können Gründe dafür sein, weshalb Fußgängersimulationen vielleicht sogar essenziell sind um die Sicherheit besagter Fußgänger zu gewährleisten und Vorfälle wie das Unglück der Love Parade 2010 [8] zu vermeiden. Mit Hilfe von Fußgängersimulationen können Engpässe bei Flucht oder Gefahrensituationen schon im Vorhinein ausgemacht und gegebenenfalls vermieden oder generell Fluchtwege und Gefahrenszenarien getestet werden. Allerdings spielen Fußgängersimulationen nicht nur für die Sicherheit eine bedeutende Rolle. Veranstalter oder Gewerbetreibende, welche ein Gewerbe in der simulierten Umgebung betreiben, können anhand einer Fußgängersimulation beispielsweise auch Prognosen über Verkaufszahlen erstellen. Mit diesen Daten können dann Bestellmengen oder Umsätze abgeschätzt werden.

Je besser die Umgebung und die Personen nachgebildet bzw. abgebildet werden können, umso realistischer sind folglich auch die Ergebnisse der Simulation. Während die Nachbildung von Grenzen, Eingängen, Ausgängen und anderen Objekten innerhalb einer Simulation verhältnismäßig einfach ist, ist die Simulation von menschlichem Verhalten hingegen deutlich vielschichtiger und komplexer [10]. Die zu simulierende Umgebung kann, im Gegensatz zum menschlichen Verhalten, in der Regel direkt der realen Welt entnommen und in die Simulationsumgebung kopiert werden. Um allerdings menschliches Verhalten realitätsnah zu simulieren, wird eine Verhaltenslogik für die Fußgänger benötigt. Bei einer solchen Verhaltenslogik spricht man auch von einer künstlichen Intelligenz.

In der Forschung der künstlichen Intelligenz können sogenannte Agenten dazu verwendet werden, um menschliches Verhalten zu simulieren. Damit die künstliche Intelligenz eines solchen Agenten Entscheidungen treffen kann, welche einer menschlichen nahe kommen, muss unter anderem dafür gesorgt werden, dass virtuelle Personen ihre Umwelt, wie auch die Menschen, über die Sinne wahrnehmen. Denn viele Entscheidungen die Menschen treffen, sind durch die momentane Wahrnehmung beeinflusst. Hierbei spricht man bei virtuellen Entitäten auch von einer sogenannten Sensorik. Folglich müssen auch virtuelle Fußgänger in der Lage sein ihre Umwelt wahrzunehmen, um mit Hilfe dieser gegebenenfalls Entscheidungen treffen zu können. Um diese Fähigkeiten bei virtuellen Fußgängern annähernd nachzuahmen, rüstet man sie sozusagen mit Sensoren aus, über die sie ihre Umgebung wahrnehmen können.

Ziel dieser Arbeit ist es, virtuelle Agenten in die Lage zu versetzen ihre Simulationsumwelt über Sensoren wahrzunehmen. Diese Sensoren sollen der virtuellen Person ermöglichen ihre Umwelt wahrzunehmen und dadurch ihre Entscheidungen beeinflussen. Die Sensoren werden durch Abfragen, die an sich selbst und an die Simulationsumgebung gestellt werden können, realisiert. Die im Rahmen dieser Arbeit entstandene Bibliothek wurde primär für die Fallstudie „Back to the Woods“ (siehe Kapitel 3) entwickelt.

In Kapitel 2 wird das Agentenkonzept und Multi-Agentensysteme kurz vorgestellt, um dann sensorische Informationen genauer zu erläutern. Kapitel 3 beschäftigt sich mit der vom Betreuer dieser Arbeit zur Verfügung gestellten Fallstudie „Back to the Woods“ und der Entwicklungsumgebung Anylogic, in welcher diese Fußgängersimulation entwickelt wurde. In Kapitel 4 wird dann die Bibliothek Senso vorgestellt, die im Rahmen dieser Arbeit entstanden ist. Kapitel 5 enthält eine Zusammenfassung der implementierten sensorischen Funktionen.

2 VIRTUELLE AGENTEN UND SENSORISCHE INFORMATIONEN

In diesem Kapitel wird zunächst das Agentenkonzept allgemein vorgestellt. Dann wird erläutert, welchen Nutzen ein Agent aus sensorischen Informationen gewinnt.

2.1 Agenten

Ein virtueller Agent ist ein populäres Konzept in der Entwicklung künstlicher Intelligenz. Eine vereinfachte Darstellung der Funktionsweise eines Agenten und seiner Umgebung findet sich in der Abbildung 1.

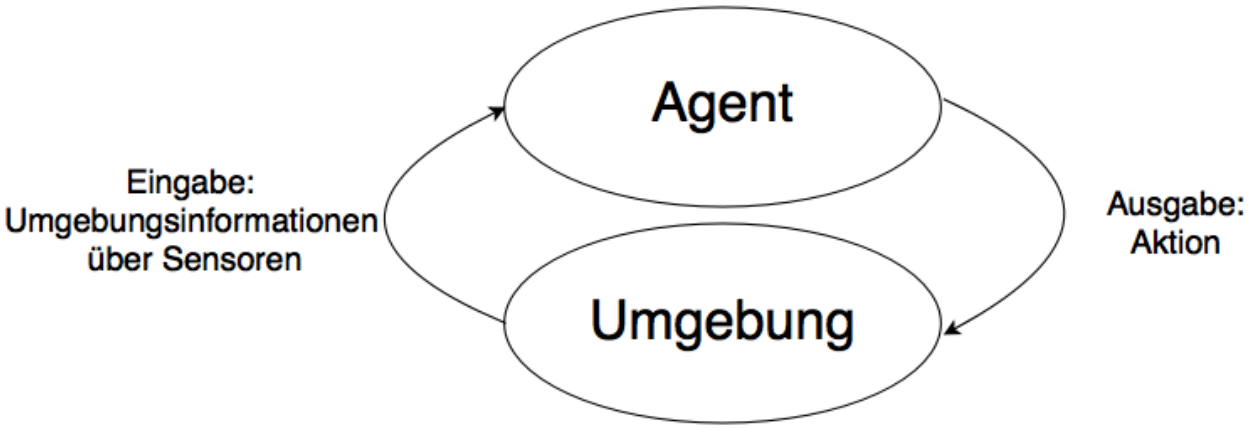


Abbildung 1: Ein Agent und seine Umgebung (nach [1]).

Ein Agent ist ein Zielsystem, in dem alle Komponenten einer künstlichen Intelligenz zusammenlaufen. Eine klare Definition für einen Agenten ist in der Literatur nicht auszumachen, allerdings gibt es eine, welche die wesentlichen Aspekte eines Agenten auf den Punkt bringt: „Ein Agent ist ein Computer-System in einer Umgebung, das in der Lage ist, in dieser Umgebung autonom zu agieren, um seine Ziele zu realisieren“[1]. Klar ist, dass Agenten in unterschiedlichen Quellen häufig bestimmte Eigenschaften zugeordnet werden: Intelligenz, Interaktivität, Autonomie und Zielorientiertheit. (vgl. [1])

- Zielorientiertheit: Der Agent erfüllt Aufgaben, die er selbst generiert, oder die von einem anderen Agenten vorgegeben werden.
- Autonomie: Der Agent kann seine Aufgaben selbstständig erledigen, wobei er eigenständig Entscheidungen trifft.
- Interaktivität: Ein Agent kann durch bestimmte Aktionen seine Umwelt kontinuierlich wahrnehmen, sie beeinflussen und auf sie reagieren. [1]

Diese drei Eigenschaften werden in der Literatur oft in nur einer Eigenschaft, der Autonomie, zusammengefasst [18]. Intelligenten Agenten schreibt man außerdem folgende Eigenschaften zu:

- Reaktivität: Ein intelligenter Agent kann Veränderungen in der Umgebungen wahrnehmen und darauf angemessen reagieren, wobei oft auch vorauszusetzen ist, dass der Agent innerhalb einer „angemessenen Zeitspanne“ reagiert. [18]
- Proaktivität: „Intelligente Agenten zeigen zielgerichtetes Verhalten und entwickeln Initiative, um ihre Ziele zu erreichen.“ [1, 18]
- Soziale Interaktivität: Der intelligente Agent kann mit anderen Agenten (ob menschlichen oder virtuellen) in Interaktion treten. [1, 18]

Es wurden einige Agentenarchitekturen entwickelt, um das Agentenkonzept zu realisieren (z.B. BDI oder PECS [5, 14]). Eine bekannte Agentenarchitektur ist die BDI-Architektur [10, 11, 13]. Hierbei steht BDI für subjektives Wissen (belief), Wünsche (desires) und Absichten (intension). Unter dem subjektiven Wissen ist zu verstehen, dass der Agent zu einem bestimmten Zeitpunkt über ein bestimmtes Wissen über sich und seine Umgebung verfügt. Aus diesem Wissen kann eine Art Vorauswahl an Wünschen abgeleitet werden. Diese Wünsche können vorgegeben sein, oder sich dynamisch aus der Simulation ergeben. Aus diesen wählt der Agent nun das für ihn beste Ziel aus. Das weitere Handeln des Agenten wird nun durch die Absicht bestimmt, bis das Ziel erreicht, befriedigt oder unerreichbar ist. [1]

Sind mehrere Agenten innerhalb einer Umgebung untergebracht, wie bei einer Fußgängersimulation, spricht man auch von einem Multi-Agentensystem. Ein Multi-Agentensystem ist von Jennings et al. [9] wie folgt charakterisiert:

- In einem Multi-Agentensystem hat jeder Agent eine subjektive Sichtweise. Ein Agent hat nur unvollständige Informationen über das System, da seine Sichtweise beschränkt ist. [9]
- Keine globale Kontrolle [9]: Jeder Agent hat einen eigenen, vollständigen Zustand, auf den andere Teilnehmer des Systems keinen Zugriff haben. Dieser Zustand ändert sich individuell als Ergebnis interner Regeln. Die Regeln selbst werden durch Daten, die in der Umgebung wahrgenommen werden, beeinflusst. [12]
- Die Daten sind vollständig dezentralisiert und verteilt auf Agenten in der Umgebung und auf die Umgebung selbst. [9, 12]

Schlussendlich ist für die meisten Agentenarchitekturen und -konzepte eine Komponente immer von Nöten: die der Wahrnehmung der Umgebung. Diese Wahrnehmungskomponente ist unabhängig von den darüber liegenden Agenten-Architekturen, insofern, dass Abfragen an die Umgebung für alle Architekturen gleich sind. Die Architekturen jedoch sind in der Regel auf eine Sensorik angewiesen.

2.2 Sensorische Informationen

Der Bedarf nach flexibler und dynamischer Interaktion zwischen intelligenten Agenten, ihrer Umgebung und untereinander ist in der Forschung der künstlichen Intelligenz früh angestiegen [3]. Obwohl die Wahrnehmung in vielen Multi-Agentensystemen weit verbreitet ist, wurde dazu relativ wenig strukturierte Forschung betrieben, die Theorien und generische Modelle für die Wahrnehmung entwickeln [17]. Das ist besonders bei Software Multi-Agentensystemen der Fall, bei denen

die Wahrnehmungskomponente explizit und anwendungsspezifisch modelliert bzw. implementiert werden muss [17].

Eine virtuelle Umgebung ist normalerweise ein Ort, an dem dynamische und unvorhersehbare Veränderungen stattfinden [16]. Daher ist es für einen virtuellen Agent nötig seine Sinne bzw. Sensoren regelmäßig zu nutzen, um seine Umgebung so gut wie möglich wahrzunehmen [16]. Die so gewonnenen Informationen kann ein Agent in seinem Aktionsauswahlmechanismus berücksichtigen, um nicht-lineare, realistische Entscheidungen zu treffen. Der Auswahlmechanismus eines Agenten ist letztendlich die Entscheidungskomponente des Agenten, der künftige Aktionen auslöst. Ein Implementierungsvorschlag für einen Aktionsauswahlmechanismus präsentiert Handel in [6].

In dieser Arbeit wird hinsichtlich sensorischer Informationen zwischen externen und internen sensorischen Informationen unterschieden. Die externen sensorischen Informationen sind eine Momentaufnahme in der direkten Umgebung des Agenten. So kann ein Agent seine Umgebung Wahrnehmen und dieses Wissen in Entscheidungen einfließen lassen. Interne sensorische Informationen hingegen sind abgespeicherte externe sensorische Informationen eines Agenten. Interne Informationen werden also durch externe Informationen erzeugt. Daher sprechen wir in dieser Arbeit bei interner Sensorik auch von einem Wissensspeicher.

Interne und externe sensorische Informationen ermöglichen dem Agenten also eine konkrete Wahrnehmung seiner Umwelt und Veränderungen in dieser.

3 ENTWICKLUNGSUMGEBUNG UND FALLSTUDIE

Im Folgenden wird die Entwicklungsumgebung Anylogic und die Fallstudie “Back to the Woods” vorgestellt, welche zusammen die Voraussetzung für die Implementierung der Sensorik in dieser Arbeit schaffen.

3.1 Anylogic

Anylogic ist eine Entwicklungsumgebung für die Implementierung und Ausführung unterschiedlichster Simulationen, welche auf der Programmiersprache Java basiert. Sie wird erfolgreich eingesetzt in Bereichen wie Produktion, Logistik, Lieferketten, Gesundheitswesen, und - besonders für wichtig für diese Arbeit - Fußgängerverhalten, um nur einige zu nennen [2, 7]. „Anylogic ist ein dynamisches Simulationswerkzeug, welches alle heute etablierten und gebräuchlichen Simulationsmethodiken unterstützt: systemdynamische, prozesszentrierte (ereignisdiskrete) und agentenbasierte Modellierung“ [2]. Besonders letztere Methodik ist für diese Arbeit essenziell. Anylogic stellt bereits mehrere optionale Bibliotheken zur Verfügung, welche bei Bedarf verwendet werden können. Darunter befindet sich auch eine Bibliothek, welche Funktionalitäten für die Modellierung von Fußgängern bereitstellt, die ‚Pedestrian Library‘. Anylogic bietet des Weiteren eine einfache Agentenstruktur auf deren Grundlage der Anwender eine spezifische Agenten Architektur (weiter-) entwickeln kann.

Mithilfe der Entwicklungsumgebung Anylogic ist es möglich eine zwei-dimensionale oder dreidimensionale Umgebung zu konstruieren und mit unterschiedlichen Objekten zu füllen. Diese Objekte können simple Umgebungsobjekte sein, wie beispielsweise Wände, Gehwege, aber auch Servicepunkte, welche virtuelle Personen nutzen können, um Bedürfnisse zu befriedigen oder Aufgaben zu bewältigen. Eine Wegfindung ist auch bereits von Anylogic Bibliotheken abgedeckt.

Um eine konkrete Implementierung von sensorischen Funktionen vorzunehmen, ist es hilfreich bereits eine vorhandene Simulationsumgebung zu haben, in der die Funktionen auch getestet werden können. Die Fallstudie „Back to the Woods“, welche noch keine detaillierte Sensorik besitzt, wurde vom Betreuer dieser Arbeit zur Verfügung gestellt. Die Fallstudie selbst ist bereits in der Arbeit [6] zum Einsatz gekommen. Im Folgenden Kapitel wird auf die Fallstudie genauer eingegangen.

3.2 Fallstudie: „Back to the Woods“

Nachfolgend werden kurz einige Daten zur Fallstudie „Back to the Woods“ aufgeführt. Das Open Air Musikfestival fand am 27. Juli 2014 auf dem Campus der Technischen Universität München statt und hatte über 5000 Besucher, wobei sich das Gelände auf ungefähr 9000 m² erstreckte. Die Festival Wiese nahe der Isar ist außerdem im Norden, Osten und Westen von zwei Bächen umgeben mit einem Haupteingang auf der Westseite, um den Großteil der Besucher einzulassen, welche über die U-Bahn Station anreisten [4].

Das Festival bestand aus zwei unterschiedlichen öffentlichen Bereichen. Ein Bereich mit der Hauptbühne und zwei großen Bars und ein anderer Bereich mit einer kleineren DJ Bühne, Essensständen, sanitären Anlagen und einem Gebiet, welches zur Entspannung vorgesehen war [4].

3.3 "Back to the Woods" in Anylogic

In Abbildung 2 ist die Nachbildung des „Back to the Woods" Festivalgeländes in Anylogic zu sehen. Bei dieser Fallstudie handelt es sich um ein Multi-Agentensystem. Es ist anzumerken, dass es sich hier nur um eine zweidimensionale Rekonstruktion handelt. Die Nachbildung in Abbildung 2 ist aus zwei grundsätzlichen Teilen zusammengesetzt. Das erste Element ist ein Plan des Festivals, welches von Öhlhorn M. im Auftrag von VABEG (www.vabeg.com), einem deutschen Unternehmen für Event Sicherheit, angefertigt wurde. Dieser Teil erfüllt also nur einen visuellen Zweck.

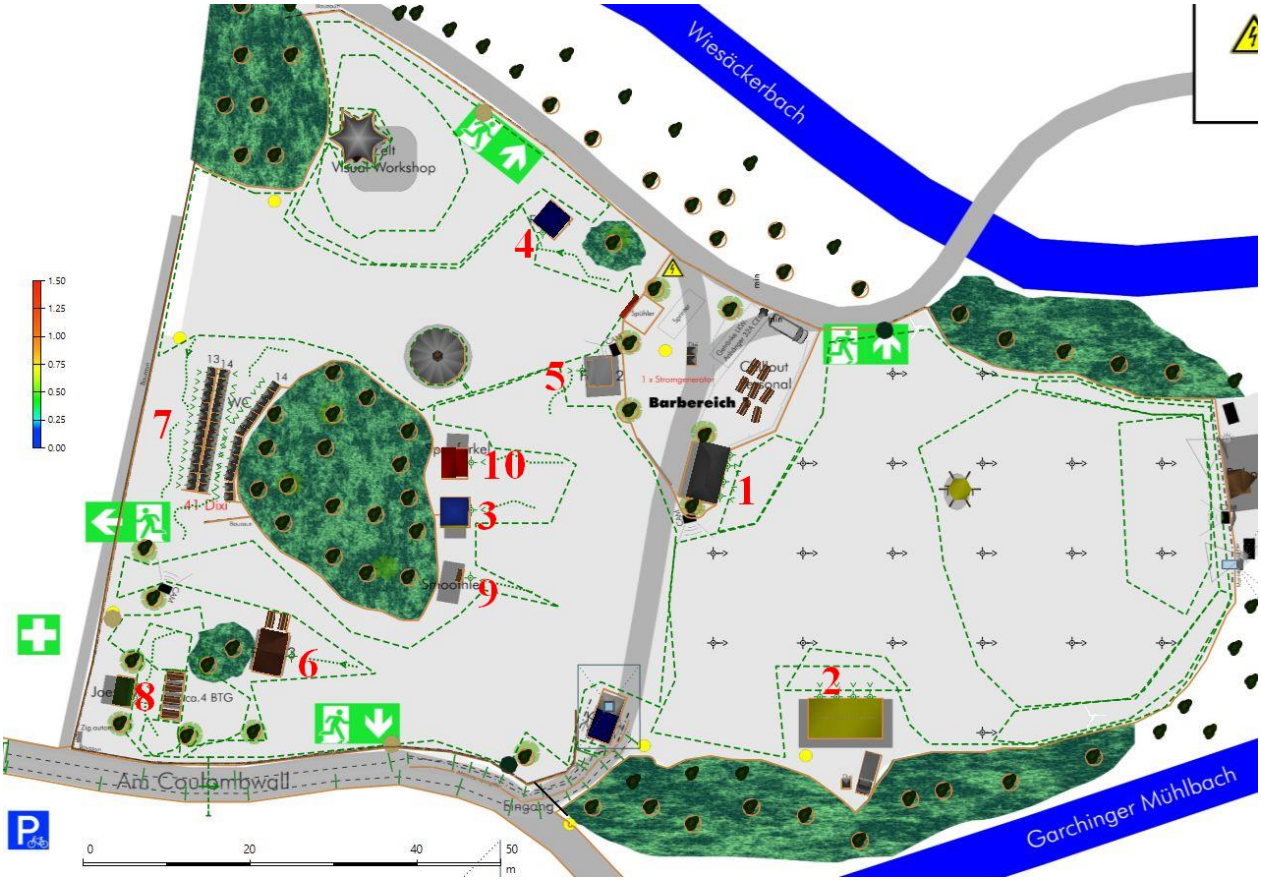


Abbildung 2: Nachbildung des "Back to the Woods" Festivals in Anylogic mit kleiner Legende.

Der andere Teil sind Anylogic Bausteine, welche teilweise bereits eine Logik für die Simulation darstellen. Die beige Umrandung des Geländes stellt Wände dar, die die Bewegungsfreiheit der Fußgänger innerhalb der Simulation beschränkt. Die gestrichelten grünen Linien sind Geländebereiche, welche virtuelle Personen beherbergen können. Man kann in der Simulation Fußgänger auch gezielt in diese Bereiche schicken bzw. kann ein Fußgänger sich auch selbst dazu entscheiden

Bereiche aufzusuchen. Weiterhin sind einige Servicepunkte (z.B. Toiletten oder Essensstände) mit dazugehörigen Warteschlangenelementen definiert, an denen virtuelle Fußgänger bestimmte innere Zustände befriedigen können (z.B. Harndrang oder Hunger). Diese Servicepunkte spiegeln natürlich die tatsächlichen Stände auf dem Festivalgelände wieder.

Alle logischen Anylogic Bausteine besitzen x, y und z Koordinaten, wobei die z Koordinate bei dieser zweidimensionalen Simulation keine Rolle spielt. Dies ermöglicht eine realistische Rekonstruktion bezüglich der Distanzen zwischen Objekten. In Tabelle 1 werden alle Service Punkte aufgezählt. Abbildung 2 (zusammen mit Tabelle 1) bilden die Grundlage für alle Beispiele und Szenarien der Implementation in Kapitel 4.

Tabelle 1: Die Namen aller Servicepunkte verknüpft mit der Nummerierung in Abbildung 2.

Nummer in Abbildung 2	Servicepunktname innerhalb der Simulation	Beispiel für Namenskonvention
1	bar1	trinken_bar1
2	bar2	trinken_bar2
3	eis	essen_eis
4	essensstand1	essen_stand1
5	essensstand2	essen_stand2
6	essensstand3	essen_stand3
7	geheAufToilette	sonstige_toiletten
8	joesPub	essen_trinken_joesPub
9	smoothies	trinken_smoothies
10	spanferkel	essen_spanferkel

Eine einheitliche Namenskonvention bei der Benennung der Servicepunkte in diesem Projekt kann, wie später im Kapitel 4 zu sehen, nützlich sein. Man könnte Beispielsweise drei Namenskonventionen in Form von Präfixen für drei Kategorien von Servicepunkten einführen: eine für Essensstände (Präfix „essen_“), Getränkestände (Präfix „trinken_“) und andere Stände (Präfix „sonstige_“). Es können so auch einem Stand mehrere Funktionen zugeordnet werden, wie bei „joesPub“ in Tabelle 1 zu sehen. Dadurch können später leichter Listen besagter Servicepunkte erstellt werden.

4 IMPLEMENTATION

Dieses Kapitel beschäftigt sich mit der Implementierung von sensorischen Abfragen im Kontext der agentenbasierten Modellierung und Simulation. Die Implementierung erfolgte in der Anylogic Version 7.0.2. Um die Implementierung generisch zu halten und um sie auch für andere Projekte nutzbar zu machen, wurde im Rahmen dieser Arbeit eine Java Bibliothek mit Hilfe von Anylogic erstellt. Unsere Bibliothek heißt *Senso* und enthält sowohl mehrere Klassen, als auch einen Agententyp *SAgent*, welcher sensorische Funktionalitäten bereitstellt. Um die Funktionalitäten von *SAgent* nutzen zu können, muss in einer Simulation ein Anylogic Agent bzw. ein Agententyp erstellt werden, der von *SAgent* erbt. Instruktionen zum Einbinden der Bibliothek, der Implementierung und zu Anwendungsbeispielen werden in diesem Kapitel vorgestellt. Beispiele, Beispielcode und Abbildungen beziehen sich auf die Fallstudie, die in Kapitel 3 vorgestellt wurde. Außerdem besitzt *Senso* noch die drei weiteren Klassen *Core*, *Item* und *Weather*, die im Verlauf des Implementations Kapitels vorgestellt werden.

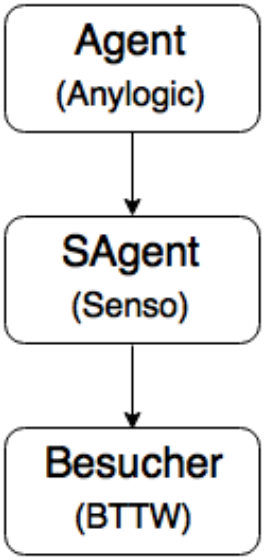


Abbildung 3: Die Vererbungshierarchie, die der Arbeit zugrunde liegt. Das oberste Glied in der Abbildung ist die Agentenklasse, die Anylogic zur Verfügung stellt. Auf der nächsten Stufe befindet sich der SAgent von Senso. Die Klasse Besucher ist der Agententyp, der im Back to the Woods Festival eingesetzt wird.

Um alle Funktionalitäten von *Senso* nutzen zu können, ist es notwendig, dass die Fußgängerbibliothek von Anylogic im Modell eingebunden ist. In Abbildung 3 ist die Vererbungshierarchie der in diesem Kapitel verwendeten Agenten dargestellt.

Zunächst wird in Kapitel 4.1 erklärt, wie Bibliotheken in Anylogic erstellt und wieder eingebunden werden können. Kapitel 4.2 beschäftigt sich mit der *Agent* Klasse von Anylogic, die bereits einige für diese Arbeit nützliche Funktionen bereitstellt, welche später auch in der Implementierung verwendet werden. Kapitel 4.3 zeigt, wie Servicepunkte in Anylogic gefiltert werden können. Kapitel 4.4 und 4.5 befassen sich dann mit den externen bzw. internen Sensorischen Funktionen von *Sensos SAgent*.

4.1 Bibliotheken in Anylogic

Dieses Unterkapitel soll die zwei folgenden Fragen beantworten: „Wie erstelle ich mit Hilfe von Anylogic eine Bibliothek, welche ich in anderen Anylogic Projekten wieder nutzen kann?“ und „Wie binde ich meine eigene Bibliothek richtig in ein anderes Projekt ein?“. Um der nachstehenden Anleitung folgen zu können, sollte sichergestellt sein, dass in der Menüleiste unter „*Ansicht“* die Schaltflächen „*Projekte“*, „*Eigenschaften“* und „*Palette“* markiert bzw. aktiviert sind.

4.1.1 Erstellen und exportieren einer Anylogic Bibliothek

Anylogic bietet eine einfache Möglichkeit eine Bibliothek zu erstellen. Um dieses Feature zu nutzen, muss man in einem Modell, welches man als Klassenbibliothek exportieren möchte, ein Bibliothekelement hinzufügen wie in Abbildung 4 gezeigt. Nachdem der Dialog zum Erstellen des Elements bestätigt wurde, öffnet sich eine Registerkarte. In dieser Registerkarte können einige Eigenschaften und Einstellungen für die zu exportierende Bibliothek vorgenommen werden.

Es gibt nun mehrere Möglichkeiten die Bibliothek zu exportieren.

- Innerhalb der Registerkarte, unter der Überschrift *„Wird exportiert“*, den Hyperlink *„Diese Bibliothek exportieren“* auswählen.
- Nachdem das Bibliothek Element hinzugefügt wurde, ist es nun möglich per Rechtsklick auf das Modell selbst oder *„Datei“* ein Kontextmenü aufzurufen, welches unter *„Exportieren…“* nun *„Bibliothek exportieren…“* anbietet.
- Rechtsklick auf das Bibliothekelement bietet die Option *„Bibliothek exportieren…“* an, welche auch einen Dialog öffnet.

Alle Optionen führen dazu, dass sich ein Dialog öffnet, in dem ein Pfad angegeben werden kann, der angibt, wohin die Bibliothek exportiert werden soll. Außerdem bietet Anylogic an, die exportierte Bibliothek direkt zur Palette hinzuzufügen. Die Bibliothek kann dadurch sofort für andere Modelle und Projekte zur Verfügung gestellt werden. Wählt man diese Option aus und generiert die Bibliothek, schließt Anylogic das soeben exportierte Modell automatisch.

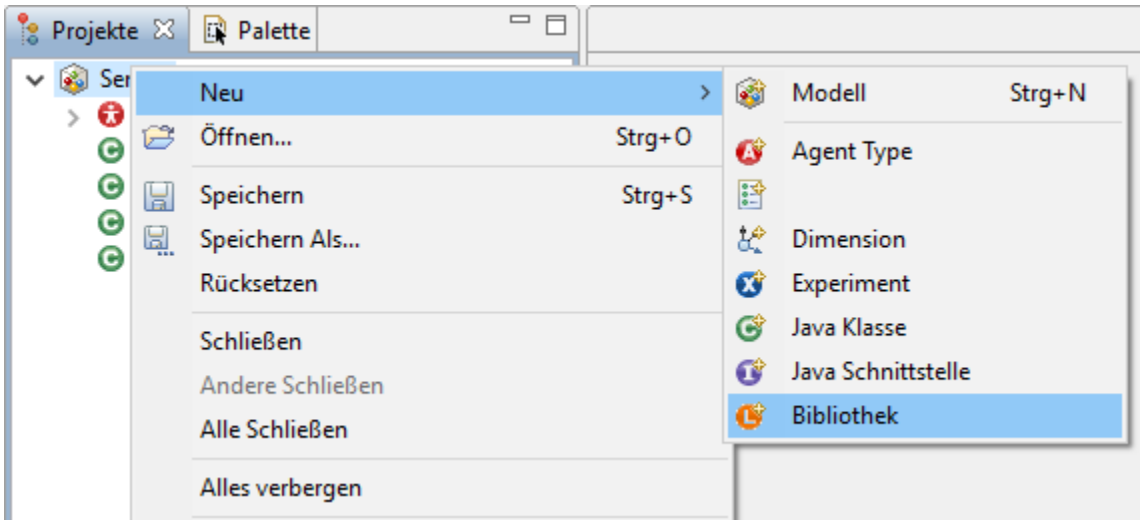


Abbildung 4: Bei einem Rechtsklick auf das Modell öffnet sich ein Kontextmenü. Unter „Neu“ kann dem Modell ein Bibliothekselement hinzugefügt werden.

Bei der Implementierung sollte darauf geachtet werden, welche *access modifier* bei Funktionen, Klassen, Agententypen oder ähnlichem, angegeben werden, da Anylogic beim Erstellen dieser automatisch den Java *default modifier* verwendet. Um Funktionalitäten nach außen bereitzustellen,

sollten die *access modifier „public“* oder „*protected“* verwendet werden. Der Name des Modells dient als *package* Name, während der Name des Bibliothekelements nur den Namen der Bibliothek im Dateisystem bestimmt.

4.1.2 Einbinden einer Bibliothek in ein Modell in Anylogic

Möchte man eine Bibliothek in ein Modell einbinden, muss man zunächst sicherstellen, dass die Bibliothek in die Palette geladen ist. Um eine Bibliothek der Palette hinzuzufügen wählt man das weiße Plussymbol im Paletten Fenster aus und wählt *„Bibliotheken verwalten…“*. Im Bibliotheken Dialog hat man die Möglichkeit Bibliotheken hinzuzufügen oder zu entfernen. Ist die gewünschte Bibliothek geladen, muss das Dialogfenster mit der OK Schaltfläche beendet werden, um die Änderungen wirksam zu machen. Nun ist die Bibliothek dem Modell zur Verfügung gestellt.

Der Eigenschaften-Tab für ein Modell sollte sich mit einem einfachen Klick auf das Modell öffnen lassen. Unter „*Dependencies“* im Eigenschaftenfenster kann man nun über die grüne Plus Schaltfläche Bibliotheken hinzufügen, welche benötigt werden um das Modell zu bauen. Wichtig ist, dass die Bibliothek bei den Anylogic Bibliotheken aufgenommen wird. Wird dies nicht beachtet, funktioniert die Vererbungshierarchie, welche in Senso aufgebaut wird, nicht.

4.2 Anylogics Agent Klasse

Die Klasse Agent in Anylogic bietet bereits einige nützliche Funktionen, auf die Senso aufbaut. In Tabelle 2 befindet sich eine Auflistung der Signaturen der genutzten Funktionen der Agent Klasse inklusive einer Kurzbeschreibung.

Tabelle 2: Eine Auflistung der verwendeten Funktionen aus der Agent Klasse von Anylogic. Die Informationen zu den Funktionen sind der Anylogic API entnommen.

Signatur	Kurzbeschreibung
distanceTo(...)	Die Routine gibt den Abstand zwischen dem Agenten und einem anderen Agenten oder einem Punkt zurück.
getEmbeddedObjects(...)	Die Methode gibt eine Liste von Objekten zurück, welche in einem Agenten eingebettet sein können.
agentsInRange(...)	Es wird eine Liste zurückgegeben, die alle Agenten zurückgibt, die sich in einer bestimmten Reichweite befinden. Zusätzlich zur Reichweite kann auch eine Liste von Agenten als weiterer Parameter übergeben werden.
getName()	Ein Getter-Aufruf für den Namen des Agenten. Der Rückgabewert ist ein String.
getPosition()	Ein Getter-Aufruf für die Position des Agenten. Der Rückgabewert ist ein Point Objekt, welches eine x, y und z Koordinate besitzt.
getRotation()	Der Getter-Aufruf gibt die momentane Ausrichtung des Agenten zurück. Die Richtung wird in Radianten angegeben.

Die Agent Klasse dient in Anylogic Simulationen auch als Container. Jede Simulationsumgebung ist ein Agent und kann andere Agenten (Populationsagenten) in sich aufnehmen. Diese können dynamisch zur Laufzeit oder am Anfang der Simulation hinzugefügt werden. Der Agent der als Umgebung dient, wird in Anylogic standardmäßig mit „*Main*“ benannt.

4.3 Filtern von Servicepunkten

In diesem Unterkapitel wird eine Grundlage für die Kapitel der externen und internen sensorischen Informationen geschaffen.

In Anylogic lassen sich in einer Simulation Servicepunkte (*PedServices*) platzieren. Servicepunkte werden verwendet, um Agenten der Simulation verschiedene Dienste bereitzustellen. Ein solcher Dienst könnte beispielsweise ein Essensverkaufsstand sein. In Anylogic gibt es bislang noch keine Möglichkeit, einen Fußgänger wahrnehmen zu lassen, dass er beispielsweise zehn Meter von einem solchen Servicepunkt entfernt steht. Dadurch werden seine Entscheidungen nicht von seiner Wahrnehmung dieser Servicepunkte beeinflusst. Senso stellt mit SAgent im nächsten Unterkapitel Funktionen vor, die diese Funktionalität gewährleisten. Allerdings benötigen diese Funktionen Listen von Servicepunkten, die in Anylogic zunächst so noch nicht konkret vorliegen. Hierfür bietet die *Core* Klasse von *Senso* die folgenden Routinen zum Erstellen der benötigten Listen.

```
public static List<PedService> instanciatePedServiceList(Agent
      environment){
      List<PedService> result = new ArrayList<PedService>();
      PedService ref = new PedService();
      List<Object> embedded = environment.getEmbeddedObjects();
      if(!embedded.isEmpty())
            for (Object obj : embedded)
                if (ref.getClass().isAssignableFrom(obj.getClass()))
                    result.add((PedService) obj);

      return result;
}
```

Codeausschnitt 1: Anylogic speichert alle Objekte der Umwelt in einer internen Liste von Objekten, welche in einen Agenten eingebettet sein können. Diese Liste lässt sich durch getEmbeddedObjects() abrufen. In der Liste befinden sich auch alle PedService Objekte. Objekte der Liste können dann sicher der Ergebnisliste hinzufügt werden, wenn die Methode isAssignableFrom(...) den Wert true zurückliefert. Gibt die isAssignableFrom(...) Methode den Wert true zurück, bedeutet das, dass das Objekt ein PedService oder eine Unterklasse davon repräsentiert. Der Agent-Parameter dieser Funktion sollte eine Referenz auf einen Agenten sein, welcher als Umgebung für andere Agenten dient und PedServices beherbergt.

Es bietet sich an die in Codeausschnitt 1 vorgestellte Routine in einem Vorgang (*Event*) zu Beginn der Simulation aufzurufen und in einer Variablen zu speichern. So kann die Referenz jedem *SAgent*, bei seinem Eintritt in die Simulation, übergeben bzw. gesetzt werden. Hierfür besitzt jeder

SAgent die Listenreferenz *pedServiceList*. Dadurch kann jeder *SAgent* innerhalb seines Lebenszyklus in seiner Simulation die zusätzlichen Funktionalitäten von *Senso* nutzen, die in den folgenden Kapiteln vorgestellt werden.

Eine weitere Funktionalität der *Core* Klasse von *Senso* ist die Möglichkeit zur Erstellung weiterer (Sub-) Listen von Servicepunkten. *PedService*-Objekte in Anylogic sind abgesehen von ihrem einzigartigen Namen, welcher durch einen String repräsentiert ist, in ihrer Funktion nicht unterscheidbar. Daher implementiert *Senso.Core* eine Möglichkeit, Listen von Servicepunkten nach Namen zu filtern. Auf diese Weise können Servicepunkte selektiert werden.

```
public static <T extends Agent> List<T> listOfAgentsByName(String name,
      Iterable<T> agentList){

      List<T> result = new ArrayList<T>();
      for(T a : agentList)
            if(a.getName().contains(name))
                  result.add(a);

      return result;
}
```

Codeausschnitt 2: Über die getName() Funktion kann der Name (siehe auch Abbildung 5) eines PedServices abgerufen werden. Die Klasse PedService erbt von der Klasse Agent, daher kann diese Methode für diesen Zweck genutzt werden. Die Senso Implementation nutzt die case-sensitive contains-Methode. Daher sollte ein Modellierer auf seine Namenskonvention achten. Alternativ zur getName() Routine ist es auch möglich die getFullName() Methode zu verwenden. Diese liefert zusätzlich den kompletten Pfad von der Wurzel bis zu diesem Objekt, gefolgt von seinem eigenen Namen.

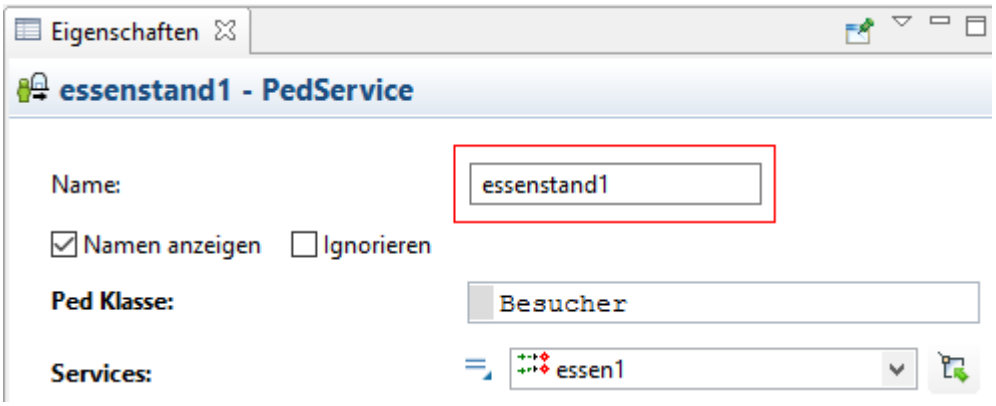


Abbildung 5: Ein Screenshot vom Eigenschaftenfenster eines PedService Objekts aus der Fallstudie. Im Bild ist der Name des Servicepunktes rot umrandet.

Der Modellierer kann also durch Namenskonventionen es Agenten ermöglichen Servicepunkte hinsichtlich ihrer Funktion in der Simulation zu unterscheiden. Mit Hilfe dieser Funktion können

Listen von Servicepunkten erstellt werden, die beispielsweise eine ähnliche Funktion in der Simulation übernehmen. Außerdem ist es natürlich möglich spezielle einzelne Servicepunkte zu filtern, indem der exakte Name des Servicepunktes als Parameter übergeben wird.

Beispiel

Im Codebeispiel 1 befindet sich ein Vorschlag, wie die eben vorgestellte Funktion aus Codeausschnitt 2 verwendet werden kann. Jedoch müssen für das Beispiel noch Vorbereitungen getroffen werden. Im Agententyp, welcher die Hauptsimulation bzw. die Umgebung repräsentiert (z.B. „*Main*“ in Anylogic), muss eine Liste erzeugt werden, welche *PedServices* speichert. In unserer Fallstudie erzeugen wir eine *List<PedService>* Variable, die *pedServiceList* und instanziieren sie zu Beginn der Simulation, wie in Abbildung 6 zu sehen ist. In Abbildung 7 ist dargestellt, wie die erzeugte und initialisierte Listenreferenz einem Besucher bei seinem Eintritt in die Simulation übergeben wird.

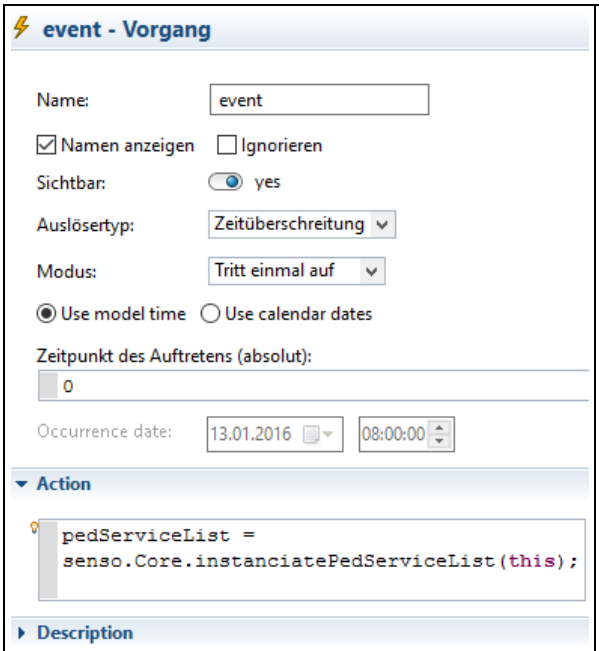


Abbildung 6: In einem Vorgang zu Beginn der Simulation wird die pedServiceList im Agenten der als Umgebung dient, instanziiert.

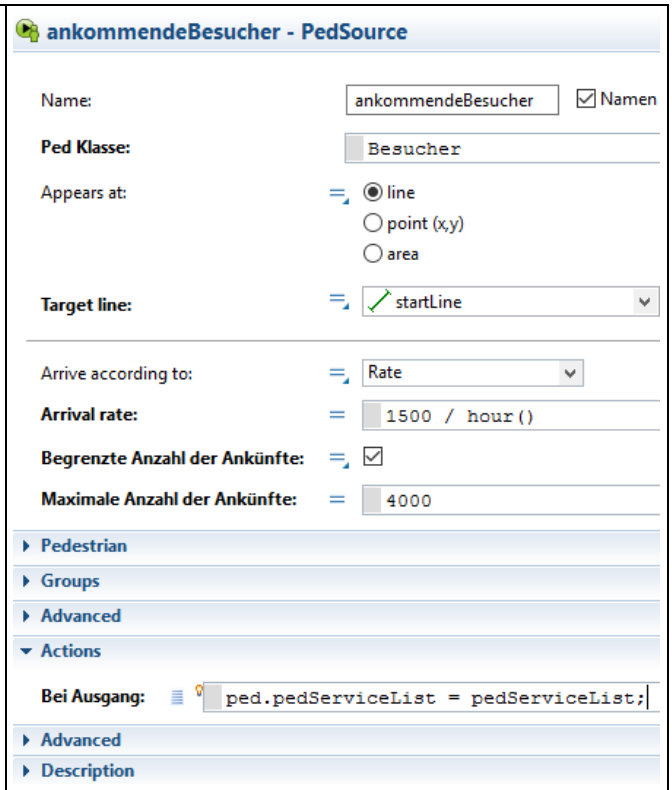


Abbildung 7: Die PedSource Klasse ermöglicht es zur Laufzeit der Simulation Fußgänger hinzuzufügen. In diesem Beispiel wird direkt nach Erzeugung des Fußgängers die pedServiceList Referenz für den Besucher gesetzt. Die Variable ped wird intern von PedSource verwaltet und repräsentiert den aktuell zu einem Zeitschritt erzeugten Fußgänger (ped ist die Abkürzung von Pedestrian in Anylogic).

Wie in Codebeispiel 1 gezeigt, können außerdem benutzerdefinierte Service Listen erstellt werden. Der Code des Codebeispiels könnte sich auch im Action-Feld von Abbildung 6 befinden. Diese so erzeugte Liste kann dann wie in Abbildung 7 jedem Besucher übergeben oder in ihm erstellt werden. Wie in Abbildung 6 zu sehen ist, kann man in der *PedSource* Klasse nur auswählen, welcher Agententyp erstellt werden soll. Normalerweise würde man in Java die Listenreferenz in einem Konstruktor übergeben. Dies ist jedoch in der *PedSource* Klasse nicht möglich. Daher wird die Referenz so gesetzt, wie in Abbildung 7 gezeigt.

```
foodServiceList =
      senso.Core.listOfPedServicesByName("essen", pedServiceList);
```

Codebeispiel 1: Man könnte eine weitere List<PedService> namens foodServiceList erstellen und dort alle Essensstände erfassen. Diese kann man dann, ähnlich wie in Abbildung 7, jedem Besucher übergeben. Die Ergebnisliste würde die Referenz der drei Essensstände mit den Nummern 5-7 der Tabelle 1 (in Kapitel 3) enthalten.

Würde der *listOfPedServicesByName(...)* Funktion ein existierender vollständiger Name (z.B. „essenstand1“) als Parameter übergeben, enthielte die Ergebnisliste nur ein Element.

4.4 Externe sensorische Informationen

Die in diesem Kapitel vorgestellten Funktionen sollen einem Agenten ermöglichen, Abfragen über seine Umwelt zu stellen. Die so gewonnenen Informationen können dann vom Agenten intern verarbeitet werden, was jedoch nicht Teil dieser Arbeit ist. Jedoch wird in manchen Beispielen - der Anschaulichkeit halber - ein möglicher sinnvoller Nutzen vorgestellt.

4.4.1 Reichweitenabfragen

Im Kontext einer Fußgängersimulation, kann es für einen Agenten innerhalb der Simulation wichtig sein, was sich in seiner unmittelbaren Umgebung, innerhalb eines bestimmten Radius oder seinem Sichtfeld befindet. Durch die Anylogic Agenten Klasse sind bereits die Funktionen *distanceTo(...)* und *agentsInRange(...)* gegeben und stellen eine Grundlage für sensorische Abfragemöglichkeiten dar.

Agenten in Reichweite

Die *agentsInRange(double distance)* Methode gibt eine Liste aller Agenten zurück, die sich innerhalb der übergebenen Distanz um den aufrufenden Agenten herum befinden. Wobei die erhaltene Liste alle Agenten der Umgebung enthalten kann, unabhängig davon, welchen Typ sie haben, solange sie als Bevölkerung der Simulation registriert sind. Die *agentsInRange(Iterable<T> agents, double distance)* Methode hingegen ermöglicht es, allgemein Listen von Agenten bzw. Klassen T, welche von Agent erben, zu prüfen, ob diese in Reichweite sind. Folglich erhält man auch eine Liste vom Typ T als Rückgabe. Ihre Funktionalität spielt in der Implementierung von *Senso* eine

wichtige Rolle. *Senso* verpackt diese Funktionalität in eigenen Methoden. Dadurch werden sensorische Abfragen gebündelt und vereinfachen Änderungen an *Senso*, wie in Codeausschnitt 3 gezeigt. Außerdem ist teilweise eine alternative, auskommentierte Implementierung zu den gegebenen Funktionen angegeben. Der nicht auskommentierte Code in allen Codeausschnitten ist der Code, welcher auch in *Senso* tatsächlich verwendet wird.

```
protected <T extends Agent> List<T> listOfAgentsInRange(double range,
      Iterable<T> agents){

      return agentsInRange(agents, range); // Anylogic API

      // Alternativ
      /*List<T> result = new ArrayList<T>();
      for(T agent : agents)
            if(distanceTo(agent) <= range)
                  result.add(agent);

      return result;*/
}
```

Codeausschnitt 3: Eine Wrapperfunktion für die agentsInRange(Iterable<T> agents, double distance) Routine mit einer auskommentierten Implementierungsalternative. Die Methode iteriert über die übergebenen Agenten und gibt diejenigen in einer Ergebnisliste zurück, welche sich innerhalb der übergebenen Distanz befinden. Durch das Iterable-Interface ist es möglich der Funktion auch andere Sammlungen außer Listen, wie z.B. Sets, zu übergeben.

Zusätzlich zur Funktion in Codeausschnitt 3 überlädt Anylogic selbst die *agentsInRange(...)* Funktion um einen naheliegenden Schritt: Man übergibt als Parameter nur noch eine Reichweite und die Funktion iteriert automatisch über alle Agenten die in der Umgebung registriert sind. Senso übernimmt auch diese Funktionalität. Die einzige Änderung gegenüber der Funktion in Codeausschnitt 3 ist, dass die Liste über die iteriert wird, vorgegeben ist. In diesem Fall ist die vorgegebene Liste die der Population der Simulationsumgebung.

```
protected List<? extends Agent> listOfAgentsInRange(double range){

      return agentsInRange(range); // Anylogic API
}

// Alternativ
/* protected <T extends Agent> listOfAgentsInRange(double range){

      return listOfAgentsInRange(getPopulation(), range);
}*/
```

Codeausschnitt 4: Eine Wrapperfunktion für Anylogics agentsInRange(double distance) Funktion und ein alternativer, auskommentierter Implementierungsvorschlag. In der Alternative ist zu se-

hen, wie durch den getPopulation() Aufruf eine Liste aller Agenten innerhalb der Simulation abgerufen werden kann. Die Funktion gibt eine Liste von Agenten oder Unterklassen von Agenten als Ergebnis zurück.

Der *range*-Parameter der den *listOfAgentsInRange(...)* Routinen übergeben muss, hängt von der Skalierung bzw. dem verwendeten Maßstab der Simulationsumgebung ab.

Beispiel

Ein Beispiel für die Verwendung aus dieser Funktionalität könnte folgendes Beispiel zu unserer Fallstudie sein: Ein Besucher des Festivals befindet sich gerade auf einer Tanzfläche. Nun nimmt er wahr, dass sich in seiner Nähe (z.B. zwei Meter) zehn andere Besucher befinden. Der Besucher könnte sich dazu entscheiden, den Bereich zu verlassen, da es ihm zu voll ist. Der Code dafür könnte so ähnlich aussehen wie in Codebeispiel 2.

```
if(listOfAgentsInRange(5).size() >= 10){
      goTo(...);
      // Verlasse den Bereich...
}
```

Codebeispiel 2: Die listOfAgentsInRange(double range) Funktion, die innerhalb von einem SAgenten aufgerufen werden kann, gibt an, wie viele andere Agenten sich innerhalb von – in diesem Beispiel – fünf Metern Reichweite des Agenten befinden. Trifft die Bedingung der if-Abfrage zu, kann der Agent als Reaktion eine Aktion ausführen. In diesem Beispiel verlässt der Agent den Bereich, falls sich mehr als zehn andere Agenten in seiner Reichweite befinden.

Services in Reichweite

Bei Servicepunkten in Reichweite handelt es sich um einen Spezialfall der Abfragen für Agenten in Reichweite. Da die Klasse *PedService* von der *Agent* Klasse erbt, können die bereits vorgestellten Funktionen wiederverwendet werden. Da Servicepunkte in unserer Fallstudie eine wichtige Rolle spielen, ist in *Senso* eine ähnliche Ausprägung wie in Codeausschnitt 4 für *PedServices* implementiert, bei der die Liste für Servicepunkte vorgegeben ist.

Jeder *SAgent* besitzt eine nicht-initialisierte Listenreferenz von *PedServices*, die *pedServiceList*. Im vorangegangenen Unterkapitel wurde bereits erläutert, wie diese Referenz zu setzen ist. Mit Hilfe dieser Referenz ist jeder *SAgent* in der Lage, zu jedem Zeitpunkt und Ort, wahrzunehmen, welche Servicepunkte sich in seiner Nähe befinden. Dazu kann die im Folgenden vorgestellte Funktion genutzt werden.

```
public List<PedService> pedServiceList; // Diese Referenz kann über die
InstanciatePedServiceList Methode gesetzt werden.

protected List<PedService> listOfPedServicesInRange(double range){

      return listOfAgentsInRange(range, pedServiceList);
}
```

Codeausschnitt 5: Die öffentliche pedServiceList, die von jedem SAgent verwaltet wird, ist zunächst nicht initialisiert. In Abbildung 6 oder Abbildung 7 ist ein Beispiel gezeigt, wie diese Referenz initialisiert werden kann. Die Funktion listOfPedServicesInRange(double range) benutzt die von SAgent intern verwaltete pedServiceList, um alle Services in Reichweite des SAgent zu ermitteln. Hierzu ruft sie die Funktion aus Codeausschnitt 3 auf.

Zusätzlich kann es für den Agenten nützlich sein, benutzerdefinierte Listen von Servicepunkte auf Reichweite zu überprüfen. So kann der Agent nach speziellen Servicepunkten oder Servicepunktarten in seiner Umgebung suchen. Daher wird außerdem die Funktion in Codeausschnitt 6 implementiert.

```
protected List<PedService> listOfPedServicesInRange(double range,
      Iterable<PedService> pedServiceList){

      return listOfAgentsInRange(range, pedServiceList);
}
```

Codeausschnitt 6: Im Gegensatz zur Funktion in Codeausschnitt 5 ist es möglich benutzerdefinierte Servicelisten, statt die intern verwaltete pedServiceList zu verwenden. Um benutzerdefinierte Listen zu erzeugen kann beispielsweise die listOfAgentsByName(…) Funktion aus dem Unterkapitel 4.3 genutzt werden (Codeausschnitt 2).

Anhand der Liste, die von der Funktion zurückgegeben wird, kann der Agent unterschiedliche Entscheidungen treffen. Beispielsweise kann die übergebene Reichweite die Sichtweite oder Laufbereitschaft des Agenten repräsentieren und ihn dazu motivieren, einen der Servicepunkte der erhaltenen Liste aufzusuchen.

Durch die Funktionen in Codeausschnitt 5 bzw. Codeausschnitt 6 kann der Agent auch die wahrgenommenen Servicepunkte seinem Wissensspeicher hinzuzufügen. Mehr Informationen zu Wissensspeichern in *Senso* sind im Kapitel zur Implementierung der internen sensorischen Funktionen zu finden.

Beispiel

Durch die in diesem Unterkapitel vorgestellten Funktionen, ist es einem *SAgent* nun möglich abzufragen, ob, wie viele und welche Agenten oder Servicepunkte sich in einer bestimmten Reichweite befinden.

Im Codebeispiel 3 wird das Codebeispiel 1 ein wenig erweitert. Die neue Funktionalität liefert dem Agenten nun konkretere Informationen über sein Umfeld. Aufgrund dieser kann der Agent gezielte Entscheidungen treffen.

```
foodServiceList =
      senso.Core.listOfPedServicesByName("essen", pedServiceList);
      //Code aus Codebeispiel 1

if(hungry){
      List foodStands = listOfPedServicesInRange(100, foodServiceList);

      if(!foodStands.isEmpty()){
            PedService target = getNearestAgent(foodStands);
            goTo(target);
            //Der Agent sucht den Essensstand auf, der ihm am nächsten ist.
      }
}
```

Codebeispiel 3: Dieser Code wird in einem Besucher ausgeführt. Zunächst wird wie in Codebeispiel 1 eine Liste von Essensständen erzeugt. Falls der Agent als internes Ziel gesetzt hat, dass er Hunger verspürt, erstellt er eine Liste von Essensständen, die sich innerhalb eines Radius von 100 befinden. Ist dies der Fall, wählt der Agent denjenigen Stand aus, der ihm am nächsten ist über die getNearestAgent(...) Funktion aus. Danach sucht er den ermittelten Stand auf.

4.4.2 Sichtfeldabfragen

Bisher hat der Agent die Möglichkeit seine Umgebung in einem 360 Grad Radius durch die Reichweitenfunktionen in Kapitel 4.4.1 wahrzunehmen. Jedoch ist es realistischer anzunehmen, dass Menschen hauptsächlich nur das Wahrnehmen, was sich in ihrem Sichtfeld befindet. *Senso* implementiert hierfür eine generische Möglichkeit, um Objekte im Sichtfeld eines Agenten wahrzunehmen.

```
protected boolean pointInSight(Point other, double visionDegree,
      double viewDistance){

      if(visionDegree < 0)
            return false;

      if(visionDegree >= 360.0)
            return distanceTo(other) <= viewDistance;

      double faceDirection = Math.toDegrees(getRotation());
      double lowBound = faceDirection - (visionDegree / 2.0);
      double upBound = faceDirection + (visionDegree / 2.0);

      return internInSight(other, lowBound, upBound, viewDistance);
}
```

Codeausschnitt 7: Die Funktion erhält als Parameter eine Referenz der Anylogic Point Klasse, welche eine x, y und z Koordinate bündelt. Der Parameter viewDistance ist ähnlich dem range

Parameter aus den bisherigen Funktionen. Er wird dazu benötigt, um den maximalen Abstand zwischen dem Agenten und dem übergebenen Punkt festzulegen. Der visionDegree Parameter soll das Sichtfeld des Agenten repräsentieren und muss vom Entwickler festgelegt werden.

Ziel der Funktion ist es, zu überprüfen, ob sich der übergebene Punkt innerhalb der Sichtweite und dem Sichtradius befindet. Zunächst werden die Randfälle für das Sichtfeld überprüft, bei denen bereits bekannt ist, was sie für den Agenten bedeuten. Unterschreitet der übergebene Grad den Wert 0, ist klar, dass der Agent nicht einmal in einer Linie geradeaus sehen kann, daher wird automatisch *false* zurückgegeben. Ist der Grad größer oder gleich 360, bedeutet das, dass der Agent alles in seiner Reichweite befindliche sehen kann. Daher ist nun nur noch zu überprüfen, ob sich der Punkt innerhalb der Sichtreichweite befindet. Danach kann die Blickrichtung des Agenten über die *getRotation(...)* Methode ermittelt und in Grad umgerechnet werden. Dann wird das Sichtfeld ermittelt, indem die Hälfte des übergebenen Sichtfeldwertes einmal auf die Blickrichtung addiert und einmal von der Blickrichtung subtrahiert wird. Als Ergebnis wird nun das Ergebnis der internen Methode in Codeausschnitt 8 zurückgegeben.

```
private boolean internInSight(Point other, double lowBound, double upBound,
      double viewDistance){

      if(distanceTo(other) <= viewDistance){

            double angle = Math.toDegrees(Math.atan2(
                  other.x - getPosition().x,
                  other.y - getPosition().y));

            if(lowBound <= -180.0){
                  lowBound += 360.0;

                  return angle >= lowBound || angle <= upBound;
            }

            if(upBound > 180.0){
                  upBound -= 360.0;

                  return angle >= lowBound || angle <= upBound;
            }

            return angle >= lowBound && angle <= upBound;
      }

      return false;
}
```

Codeausschnitt 8: Diese Routine führt die tatsächliche Überprüfung durch, ob der übergebene Punkt innerhalb des Sichtfeldes und Reichweite des Agenten befindet.

Zu Beginn der Routine wird überprüft, ob der Punkt innerhalb der angegebenen Sichtweite befindet. Ist dies der Fall, wird mit Hilfe der *Math.atan2(...)* Funktion der Winkel von der y-Achse ausgehend zum Punkt *other* berechnet. Um dies zu tun, muss erreicht werden, dass die Position des Agenten den Punkt (0, 0) in einem kartesisches Koordinatensystem annimmt. Dafür wird nur der übergebene Punkt um die Position des Agenten verschoben. Indem die x- bzw. y-Koordinate des Punktes um die x- bzw. y-Koordinate des Agenten reduziert wird, wird diese Verschiebung erreicht.

Normalerweise wird die atan2 Funktion genutzt um den Winkel zwischen der Abszisse und einer Schnittgeraden durch den Ursprung zu berechnen. Jedoch ist es in Anylogic üblich, dass die positive y-Achse als Richtungsangabe durch den Wert 0° beschrieben wird. Wobei -90° der negativen x-Achse, 90° der positiven x-Achse und 180° der negativen y-Achse entspricht. Folglich kann eine Richtung die Werte zwischen -180° und einschließlich 180° Grad annehmen. Daher ist eine Spiegelung der Abszisse und Ordinate nötig. Diese wird erreicht, indem statt *Math.atan2(y, x)* dieselbe Funktion nur mit vertauschen Argumenten, nämlich *Math.atan2(x, y)*, aufgerufen wird.

An diesem Punkt sind alle Werte (*upBound*, *lowBound*, *angle*) bestimmt, die benötigt werden, um festzustellen, ob der übergebene Punkt innerhalb des Sichtfeldes liegt. Für die Überprüfung müssen drei Fälle unterschieden werden. Dies ist nötig, da die untere oder obere Schranke im Code (*lowBound* bzw. *upBound*) unzulässige Winkelwerte annehmen kann, die nicht sinnvoll mit der Blickrichtung des Agenten verglichen werden kann.

<u>Fall 1: lowBound <= -180.0</u>

Der Grundgedanke der unteren bzw. oberen Schranke ist der, dass die Blickrichtung des Agenten innerhalb der unteren und oberen Sichtschranken liegt. Dieser Gedanke ist allerdings in dieser Fallunterscheidung falsch, da in manchen Fällen die untere Schranke einen Wert gleich oder unter -180 Grad annehmen kann. Die Blickrichtung kann aber nur einen Wert zwischen -180 und einschließlich 180 Grad annehmen. Durch eine Addition um den Wert 360 befindet sich die untere Grenze wieder im Bereich von]-180,180]. Nun ist es ausreichend, wenn die Blickrichtung des Agenten eine der zwei folgenden Bedingungen erfüllt:

- Der Winkel des Punktes ist größer oder gleich der unteren Grenze, oder
- er ist kleiner oder gleich der oberen Grenze.

Ist einer der beiden Bedingungen erfüllt, wird als Ergebnis *true* zurückgegeben. Abbildung 8 veranschaulicht den eben geschilderten ersten Fall.

<u>Fall 2: upBound > 180.0</u>

Ähnlich wie in Fall 1, kann durch die Addition des halben Sichtgrades ein Wert größer als 180 Grad entstehen. In diesem Fall wird von der oberen Grenze 360 abgezogen, um die obere Grenze wieder vergleichbar zu machen. Es entsteht im Prinzip dasselbe Szenario wie in Fall 1. Daher veranschaulicht Abbildung 8 auch diesen Fall. Es wird dann der Wert true zurückgegeben wenn eine der beiden folgenden Bedingungen erfüllt ist:

- Der Winkel des Punktes ist größer oder gleich der unteren Grenze, oder
- er ist kleiner oder gleich der oberen Grenze.

Fall 3: upBound <= 180.0 && lowBound > -180.0

Dieser Fall sollte der intuitivste Fall sein, da es als Bedingung ausreichend ist, dass sich der Winkel des Punktes größer oder gleich der unteren Grenze und gleichzeitig kleiner oder gleich der oberen Grenze sein muss. In Abbildung 9 ist dieser Fall veranschaulicht.

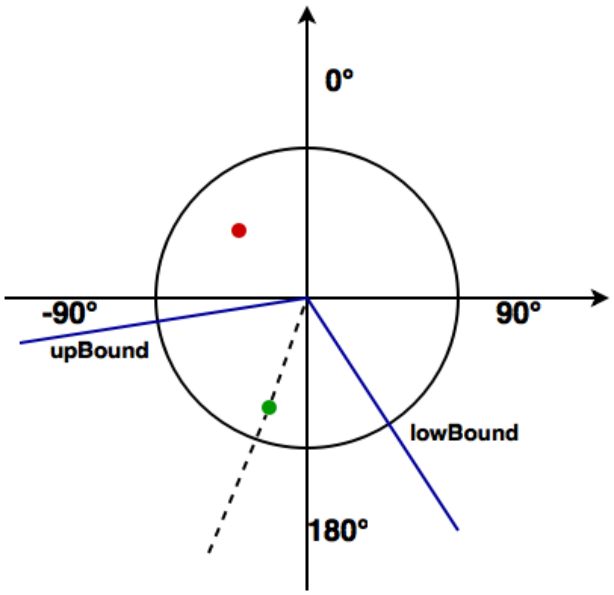


Abbildung 8: Der Agent befindet sich im Punkt (0,0). Die blauen Linien stellen die obere und untere Grenze für den Winkel der Blickrichtung des Agenten dar. Die gestrichelte Linie bildet die Richtung ab, in welche der Agent ausgerichtet ist. Zu sehen ist, dass der grüne Punkt in der Abbildung auf der Sichtlinie des Agenten liegt. Zusätzlich befindet er sich innerhalb des schwarzen Kreises, welcher die Sichtreichweite darstellt. Daraus ist zu folgern, dass der grüne Punkt sich innerhalb der Sicht des Agenten befindet. Der rote Punkt hingegen befindet sich zwar innerhalb der Reichweite des Agenten, jedoch außerhalb seines Sichtfeldes.

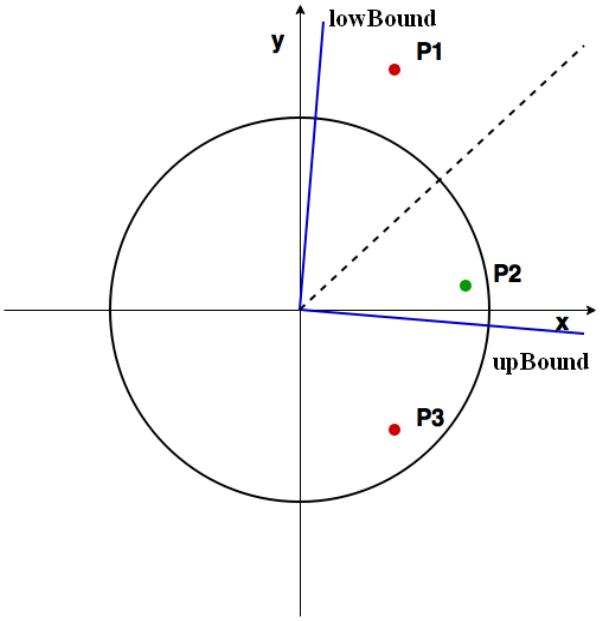


Abbildung 9: Die blauen Linien repräsentieren wie in Abbildung 8 die obere und untere Sichtschranke aus Sicht des Agenten. Für den grünen Punkt in der Abbildung wird der Wert true zurückgegeben, da er sich innerhalb der Sichtreichweite und den Sichtschranken befindet. Die beiden roten Punkte erfüllen je eine der beiden benötigten Sichtkriterien nicht, und liefern deshalb den Wert false zurück.

Durch die *pointInSight(...)* Methode ist es dem Agenten nun möglich, Punkte in seinem Sichtfeld wahrzunehmen. Der Nutzen aus dieser Funktionalität ist im Prinzip der Selbe wie bei reinen Reichweitenabfragen. Analog zu diesen ist mit der privaten Methode *internInSight(...)* ein Grundstein gelegt, um Methoden zu schreiben, die auch über Listen von Agenten iterieren können. Als nächstes wird die Implementation der Funktion *listOfAgentsInSight(…)* gezeigt, die genau das macht.

```
protected <T extends Agent> List<T> listOfAgentsInSight
      (double visionDegree, Iterable<T> list, double viewDistance){

      List<T> result = new ArrayList<T>();

      if(visionDegree < 0)
            return result;

      if(visionDegree >= 360.0)
            return listOfAgentsInRange(viewDistance, list);

      double faceDirection = Math.toDegrees(getRotation());
      double lowBound = faceDirection - (visionDegree / 2.0);
      double upBound = faceDirection + (visionDegree / 2.0);

      for(T agent : list){
            if(internInSight(agent.getPosition(), lowBound,
                  upBound, viewDistance))

                  result.add(agent);
      }

      return result;
}
```

Codeausschnitt 9: Zunächst wird eine leere Ergebnisliste erstellt. Wie in Codeausschnitt 7 werden dann die Randfälle für visionDegree behandelt. Bei einem Sichtfeld von über 360 Grad wird die in Sichtweite-Funktion auf nur noch eine Reichweitenfunktion reduziert, welche bereits in Senso implementiert ist. Analog zur pointInSight(...) Routine werden nun noch die nötigen Variablen kreiert um dann über alle Elemente der Liste zu iterieren. In der if-Abfrage kann nun die interne Funktion internInSight(...) aus Codeausschnitt 8 verwendet werden, welche praktischer Weise einen Boolean-Wert zurückliefert, der angibt, ob sich die Position des aktuellen Agenten Elements in Sicht und Reichweite befindet. Ist dies der Fall, kann der aktuelle Agent der Ergebnisliste hinzugefügt werden. Am Ende der Routine wird die Ergebnisliste zurückgegeben.

Zur Funktion in Codeausschnitt 9 ist anzumerken, dass sie keine Sichtschatten berücksichtigt. Das bedeutet, dass Agenten die Sicht auf andere Agenten nicht verdecken können.

4.4.3 Warteschlangen von Servicepunkten

Nachdem mehrere Möglichkeiten aufgezeigt wurden, wie benutzerdefinierte Listen von Servicepunkten erzeugt oder einzelne Servicepunkte selektiert werden können, geht es in diesem Unterkapitel um die Warteschlangen die diese Servicepunkte besitzen können. Bis jetzt kann der Besucher Servicepunkte und ihre Funktion wahrnehmen. Allerdings ist noch nicht berücksichtigt, dass sich ein Besucher lieber in einer kürzeren Warteschlange bei einem alternativen Geschäft anstellt, statt in einer vollen Warteschlange. *Senso.Core* implementiert eine Funktion, die es ermöglicht abzufragen, welche Agenten sich in den Warteschlangen eines Servicepunkts befinden. Wie dies implementiert ist, ist in im Codeausschnitt 10 dargestellt.

```
public static List<Agent> listOfPedsInQueue(PedService pedService){

      List<QueueUnit> queues;
      List<Agent> result = new ArrayList<Agent>();

      if(pedService.services != null){
            queues = pedService.services.getQueues();

            for(QueueUnit qu : queues)
                  result.addAll(qu.getPeds());
      }

      return result;
}
```

Codeausschnitt 10: Die listOfPedsInQueue(...) nimmt als Parameter einen PedService entgegen und gibt die Agenten aller Warteschlangen zurück. Die Methode iteriert über alle Warteschlangen des PedService, falls der übergebene Service solche besitzt und fügt die Agenten in den Warteschlangen der Ergebnisliste hinzu und gibt diese als Ergebnis zurück.

Beispiel

Im Folgenden Codebeispiel wird gezeigt, wie die neue Funktionalität sinnvoll angewendet werden kann. Dazu wird das Codebeispiel 3 aus dem Unterkapitel zu Reichweitenabfragen ein wenig abgeändert. Unser Besucher versucht wieder sein Hungerbedürfnis zu stillen. Dafür sieht er sich um und sucht nach Essensständen in seiner Reichweite. Findet er mehr als einen Stand, entscheidet er sich für den, bei dem die Warteschlange am kürzesten ist und sucht ihn auf.

```
foodServiceList =
      senso.Core.listOfPedServicesByName("essen", pedServiceList);
      // Code aus Codebeispiel 1

if(hungry){
  List foodStands = listOfPedServicesInRange(100, foodServiceList);

  if(!foodStands.isEmpty()){
    PedService target = foodStands.get(0);

    for (PedService service : foodStands)
      if(listOfPedsInQueue(service).size() <
                              listOfPedsInQueue(target).size())
            target = service;

    goTo(target);
    // --> Agent sucht den Essensstand mit der kürzesten Warteschlange auf
  }
}
```

Codebeispiel 4: Im Gegensatz zu Codebeispiel 3 wird in diesem Beispiel eine Minimum Suche durchgeführt, welche den Essensstand mit der kürzesten Warteschlange heraussucht. Befinden

sich Essensstände in Reichweite des Besuchers, sucht er denjenigen mit der kürzesten Warteschlange auf.

4.4.4 Wetter

Anylogic bietet zunächst keine Möglichkeit Wetter zu simulieren. Da das Wetter aber Einfluss auf Entscheidungen des Menschen haben kann, ist es durchaus sinnvoll, dass auch virtuelle Personen von Wetterkonditionen hinsichtlich ihrer Entscheidungen beeinflusst werden können. *Senso* implementiert eine einfache Wetterklasse, welche minimale Wetteranforderungen stellt. Durch diese Implementation ist es einem Entwickler auch möglich, die Wetterklasse beliebig zu erweitern.

```
public class Weather {

	public Weather(double temperatureInCelsius, double humidity,
			double airPressure, double windStrength,
			CellDirection windDirection){

		this.temperatureInCelsius = temperatureInCelsius;
		this.humidity = humidity;
		this.airPressure = airPressure;
		this.windStrength = windStrength;
		this.windDirection = windDirection;
	}

	// ...

}
```

Codeausschnitt 11: Die Klasse Weather mit ihrem Konstruktor ist in diesem Codeabschnitt abgebildet, wobei Weather zusätzlich noch einen leeren Konstruktor besitzt. Die Klasse verwaltet fünf private Variablen, zu denen jeweils zusätzlich Getter und Setter implementiert ist. Weather repräsentiert eine Temperatur, die in Grad Celsius angegeben wird, eine Luftfeuchtigkeit, einen Luftdruck, eine Windstärke und Windrichtung. Für die Windrichtung wird die Anylogic Klasse CellDirection genutzt, die acht unterschiedliche Werte annehmen kann. Sie bietet zusätzlich zu den vier Windrichtungen Nord, Ost, Süd und West auch noch die Richtungen Nord-Ost, Süd-Ost, Süd-West und Nord-West.

Es bietet sich an, für das Wetter eine globale Referenz zu erzeugen, welche das Wetter für die ganze Simulation repräsentiert. Diese Wetterreferenz kann dann jedem Agenten bei seiner Erzeugung übergeben werden (analog zur Erklärung zum setzen der *pedServiceList* in Kapitel 4.3). Hierfür besitzt jeder *SAgent* eine nicht-instanziierte Referenz zum aktuellen Wetter, sie heißt *weather*.

Eine Logik, wie sich das Wetter entwickelt, ist in *Senso* nicht enthalten. Ein Anwendungsbeispiel für den Nutzen der Wetter Klasse ist der Zusammenhang von Konsum warmer Getränke und Speisen bei kaltem bzw. schlechtem Wetter. Es ist natürlich auch möglich dem Agenten eine andere Wetterreferenz zu übergeben, wenn er beispielsweise einen anderen Bereich der Simulation betritt.

4.5 Interne Sensorische Informationen (Wissensspeicher)

In diesem Kapitel werden Funktionalitäten von *Senso* vorgestellt, die sich damit beschäftigen, eine interne Sensorik zu erstellen und Abfragen an diese zu richten. Letztendlich handelt es sich bei der internen Sensorik - in dieser Arbeit - um einen Wissensspeicher von externen sensorischen Informationen. Mit Hilfe der internen Sensorik soll der Agent in der Lage sein, eine Art Erfahrung aufzubauen, welche dynamisch erweitert und abgefragt werden kann.

Die durch *Senso* erstellten Wissensspeicher besitzen keine Implementation oder Logik für einen Wissensverfalls- oder Wahrnehmungsprozess. Diese muss, falls gewünscht, vom Modellierer der Agenten entwickelt werden.

4.5.1 Wahrnehmungswissensspeicher für Agenten und Services

Wie ein Fußgänger andere Fußgänger bzw. Servicepunkte zu einem beliebigen Zeitpunkt wahrnehmen kann, wurde bereits im vorherigen Kapitel erläutert. Nun geht es darum, diese gewonnenen Informationen zu erhalten. Um den Entwickler nicht in eine bestimmte Richtung zu drängen, verwaltet SAgent keine eigenen Kollektionen von Agenten oder Servicepunkten. Stattdessen ist die Erstellung dieser Kollektionen dem Entwickler überlassen.

Die *Senso.Core* Klasse bietet jedoch eine Möglichkeit um Abfragen an diese Listen zu richten. Wie bereits im Kapitel zu den Reichweitenabfragen erwähnt, ist es möglich Agenten bzw. Servicepunkte durch ihren Namen zu unterscheiden. Um also vom Entwickler erstellte Listen zu Filtern, kann in Senso die Funktion *listOfAgentsByName(...)* verwendet werden. Die *listOfAgentsByName(...)* Methode wurde bereits im Kapitel in 4.3 vorgestellt und ermöglicht das gezielte filtern von bestimmten Agenten oder auch Servicepunkten aus einer Liste.

Beispiel

Im Folgenden wird ein Beispiel zu einer benutzerdefinierten Liste eines Wahrnehmungsspeichers vorgestellt. Außerdem wird gezeigt, wie diese Liste gefüllt und abgefragt werden kann. Es kann zudem von Vorteil sein, statt Listen sogenannte *Sets* zu verwenden. Das *Set* Interface von Java bietet die für uns nützliche Eigenschaft, dass keine identischen Einträge abgelegt werden können. Ansonsten verhalten sich Klassen, die das *Set* Interface implementieren, ähnlich wie die Implementationen die das *List* Interface implementieren: Sie speichern Referenzen hinzugefügter Elemente.

```
Set<PedService> perceivedPedServices = new HashSet<PedService>();
/* Diese Variable sollte intern in einem Agenten als Variable oder Parameter
erstellte werden werden! */

for(PedService ps : listOfAgentsInSight(90, pedServiceList, 30))
      if(Math.random() < 0.5)
            perceivedPedServices.add(ps);
```

Codebeispiel 5: Im ersten Teil des Codes ist zu sehen, wie ein HashSet - eine Implementation des Set Interfaces - genutzt wird, um ein Set zu erstellen, welches wahrgenommene PedServices speichern soll. Im unteren Teil des Beispielcodes befindet sich ein Codeabschnitt, der zum Beispiel in einem Vorgang, welcher in bestimmten Abständen der Simulation innerhalb eines Agenten, aufgerufen wird um das Set zu füllen. Durch die if-Abfrage in der for-Schleife ist simuliert, dass eine eine 50% Wahrscheinlichkeit besteht, dass sich der Agent an die Servicepunkte erinnert, die er gesehen hat. Der Aufruf der listOfAgentsInSight(...) gibt hier eine Liste von PedServices zurück, die sich in einem Sichtfeld von 90 Grad und einer Entfernung von 30 befinden.

Wie in Codebeispiel 5 gezeigt, können beispielsweise über Reichweiten oder Sichtfeldabfragen Wissensspeicher befüllt werden. Nun soll unser Besucher auch noch einen Nutzen aus dem gewonnen Wissen ziehen.

```
if(hungry){
      PedService target = getNearestAgent(
      senso.Core.listOfObjectsByName("essen", perceivedPedServices));

      if(target != null)
            goTo(target);
}
```

Codebeispiel 6: Dieser Code könnte Bestand eines Aktionsauswahlmechanismus sein. Der Agent hat in diesem Beispiel das Ziel, seinen Hunger zu stillen. Er versucht sich zu erinnern, wo sich der nächste Essensstand befindet. Falls er sich an mindestens einen Essensstand erinnert, wird er diesen aufsuchen, um sein Bedürfnis zu befriedigen.

Im Codebeispiel 5 wurde ein Vorschlag dargelegt, wie ein Wissensspeicher erstellt und gefüllt werden kann. Weitere Vorschläge zu Wissensspeichern wären beispielsweise Listen bzw. Sets, die festhalten, welche Servicepunkte bereits besucht wurden oder mit welchen anderen Agenten der Agent bereits interagiert hat. Im Codebeispiel 6 ist dann zusätzlich ein möglicher Anwendungsfall demonstriert, der auf dem Codebeispiel 5 beruht.

4.5.2 Gegenstände

In diesem Unterkapitel wird eine neue Klasse vorgestellt, die es dem Entwickler ermöglicht, Gegenstände zu simulieren, die ein Agent tragen kann. Passend dazu wird dem Agenten eine Methode von *Senso* zur Verfügung gestellt, die Abfragen an die Listen der Gegenstände ermöglicht. Zunächst wird jedoch die *Item* Klasse von *Senso* in Codeausschnitt 12 vorgestellt.

```
public abstract class Item {

    public Item() {
    }
}
```

Codeausschnitt 12: Die Item Klasse besitzt einen leeren Konstruktor. Wichtig, aber nicht zwangsläufig notwendig ist, dass alle Klassen, die von Item erben, auch einen leeren Konstruktor zur Verfügung stellen. Da die Klasse als abstract deklariert ist, ist es nicht möglich eine Instanz von ihr zu initialisieren. Ansonsten besitzt die Item Klasse keine weiteren Methoden oder Variablen.

Mit Hilfe der *Item* Klasse kann eine Vererbungshierarchie von Gegenständen erstellt werden. Diese Gegenstände kann dann ein Agent bei sich tragen. Um dies zu simulieren können diese Gegenstände einem Wissensspeicher hinzugefügt werden. Dafür verwaltet der SAgent eine Liste, die Objekte der Klasse *Item* beinhaltet, die *carriedItems* Liste (siehe auch Codeausschnitt 13). Um Abfragen an diese Liste zu stellen implementiert *Sensos SAgent* drei Funktionen, die vom Aufbau fast identisch sind.

```
public List<Item> carriedItems = new ArrayList<Item>();

protected <T extends Item> List<T> listOfItemsAgentCarries(T item){

      List<T> result = new ArrayList<T>();
      for(Item i : carriedItems)
            if(item.getClass().isAssignableFrom(i.getClass()))
                  result.add((T) i);

      return result;
}
```

Codeausschnitt 13: Die carriedItems Liste wird als leere ArrayList initialisiert. Die Funktion erstellt zunächst eine leere Ergebnisliste und iteriert über jedes Element der carriedItems Liste. Dabei wird überprüft, welche der Elemente dieselbe Klasse haben, wie der übergebene Parameter item. *Auch Klassen, die Unterklassen vom Typ T sind, werden der Ergebnisliste hinzugefügt.*

Beispiel

Um die eben vorgestellten Funktionalitäten nutzen zu können, muss der Entwickler zunächst eine Vererbungshierarchie erstellen. Um den Nutzen der Funktionalität zu demonstrieren gehen wir von einer Vererbungshierarchie wie in Abbildung 10 aus.

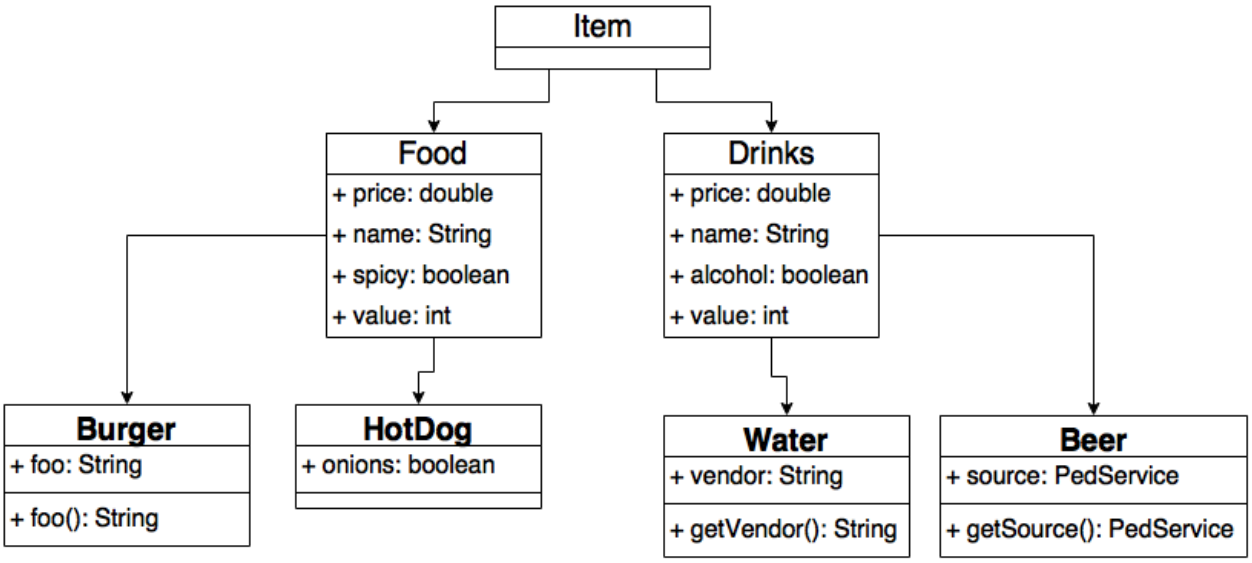


Abbildung 10: UML-Diagramm einer Beispielhierarchie für die Klasse Item. Jede der abgebildeten Klassen besitzt einen leeren Konstruktor. Außerdem implementieren die jeweils untersten Klassen im Diagramm (Burger, HotDog, Water und Beer) je einen weiteren Konstruktor, der die klasseneigenen und die Variablen der Oberklasse(n) initialisiert.

Um Abfragen an die *carriedItems* Liste zu stellen, muss diese zunächst mit Einträgen bestückt werden. Beispielsweise könnte der Agent einen Gegenstand erhalten (und seiner Liste hinzufügen), wenn er einen Servicepunkt verlässt, wie in Abbildung 11 demonstriert.

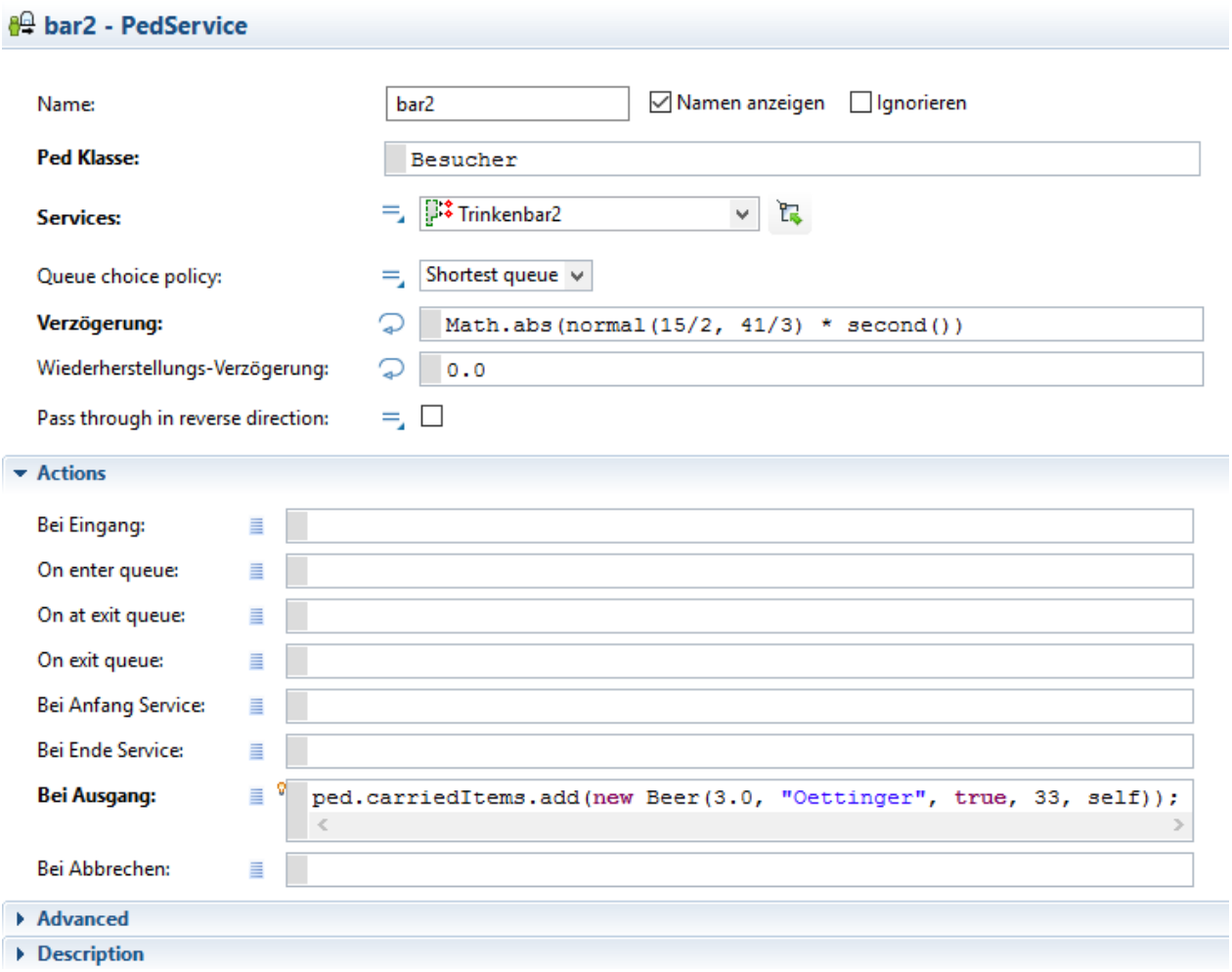


Abbildung 11: In dieser Abbildung ist zu sehen, wie die carriedItems Liste gefüllt werden könnte. Beim Verlassen des Servicepunktes ist der Besucher in Besitz eines Biers bzw. hat es bereits getrunken (je nach Auslegung). Das Schlüsselwort self (statt this) ist hier die Referenz auf den PedService "bar2".

Welche Informationen aus dem Wissensspeicher für Gegenstände gezogen werden können ist dem Entwickler überlassen. Im Codebeispiel 7 ist erklärt, wie Abfragen über *Senso* Funktionen an die *carriedItems* Liste verwendet werden können. Im Beispiel wird überprüft, ob der Besucher mehr als zwei Getränke bereits gekauft oder getrunken hat. Falls dies der Fall ist, könnte der Agent als Reaktion weniger Geld ausgeben wollen, öfter auf Toilette müssen oder sein Geld lieber für Essen aufheben.

```
/* Man gehe davon aus, dass die carriedItemsList ist gefüllt
mit einem "Burger", zwei "Water" und einem "Beer" */

if(numberOfItemsAgentCarries(new Drink()) > 2) // ausgewertet zu: true
      decreaseProbabilityOfBuyingMoreDrinks();


System.out.println(agentCarriesItem(new Burger()); // prints: true

System.out.println(listOfItemsAgentCarries(new HotDog()).size(); // prints: 0
```

Codebeispiel 7: Im Beispiel wird davon ausgegangen, dass sich insgesamt bereits 4 Elemente in der carriedItems Liste befinden, die zum Beispiel wie in Abbildung 11 demonstriert hinzugefügt wurden. Im ersten Teil der if-Abfrage wird von der Funktion der Wert 3 zurückgegeben, da die Klassen Water und Beer von Drink erben und sich in der Liste zwei Instanzen von Water und eine Instanz von Beer befindet. Auf diese Weise kann also eine Kategorie abgefragt werden. Die erste Ausgabe auf der Konsole wird zu true ausgewertet, da sich mindestens ein Element der Klasse Burger in der Liste befindet. Die zweite Ausgabe auf der Konsole wird zu 0 ausgewertet, da sich kein einziges Element der Klasse HotDog in der carriedItems Liste befindet.

4.5.3 Agentennetzwerke

Anylogic bietet dem Entwickler die Möglichkeit Agenten einer Simulation zu vernetzen. Vernetzte Agenten können sich Nachrichten zukommen lassen, auf die Reaktionen implementiert werden können. *Senso* bietet keine Erweiterung zu den Anylogic Agentennetzwerken. Jedoch ist es dem Entwickler möglich, einen Wissensspeicher von Agenten zu erstellen, mit denen der Agent bereits verbunden war. Der Wissensspeicher kann dann wie gewohnt mit *Senso* Funktionen ausgelesen werden. Weitere Informationen zu Anylogics Agentennetzwerken sind der Anylogic Hilfe Seite zu entnehmen [15].

5 ZUSAMMENFASSUNG

Im Rahmen dieser Arbeit wurden Funktionen vorgestellt, die es Entwicklern von agentenbasierten Simulationen erlaubt, ihre(n) Agenten in Anylogic mit einer Sensorik auszustatten. Die Bibliothek *Senso* wurde für die Fallstudie „Back to the Woods“ entwickelt und erweitert sie in einem sinnvollen Rahmen. Daher sind mit ihrer Entwicklung nicht alle Möglichkeiten einer Sensorik vollständig ausgeschöpft.

Die *Senso* Bibliothek ermöglicht externe sensorische Abfragen an ihre Umwelt. So können gezielt Servicepunkte und Agenten aus der Simulation gefiltert werden. Außerdem wurden in *Senso* Funktionen bereitgestellt, die das Sichtfeld eines Agenten berücksichtigen. Servicepunkte können mit Hilfe von *Senso* in ihrer Funktion und der Länge ihrer Warteschlangen unterschieden werden. Außerdem kann durch die *Weather* Klasse das Wetter für eine Simulation simuliert werden.

Für eine interne Sensorik kann der Entwickler beliebige Wissensspeicheragentenlisten erzeugen. Diese Listen können dann über *Senso* Funktionen sowohl gefüllt werden, als auch Abfragen an sie richten. Die *Item* Klasse ermöglicht zusätzlich die Funktionalität Hierarchien von Gegenständen zu erstellen. Diese können dann gezielt aus dem Wissensspeicher für Gegenstände wieder abgefragt werden.

Allgemein ist anzumerken, dass externe sensorische Abfragen in angebrachten Abständen aufgerufen werden, da manche Multi-Agentensysteme eine Vielzahl an Agenten haben können. Durch exzessive Aufrufe von *Senso* Funktionen kann die Laufzeit der Simulation stark ansteigen.

Senso trägt somit dazu bei, Fußgängersimulationen realistischer zu gestalten. Dadurch können gegebenenfalls Problemzonen, Engpässe oder ähnliches besser ermittelt und Präventivmaßnahmen ergriffen werden.

6 QUELLEN

[1] 2008. Agenten. In *Methoden wissensbasierter Systeme*. Vieweg+Teubner, Wiesbaden, 338–364. DOI=10.1007/978-3-8348-9517-2_11.

[2] *Anylogic Features - Warum Anylogic?* http://www.anylogic.de/features. Accessed 2 March 2016.

[3] Arnellos, A., Vosinakis, S., Anastasakis, G., and Darzentas, J. 2008. Autonomy in Virtual Agents: Integrating Perception and Action on Functionally Grounded Representations. In *Artificial intelligence. Theories, models and applications : 5th Hellenic Conference on AI, SETN 2008, Syros, Greece, October 2-4, 2008 ; proceedings / John Darzentas ... [et al.] (eds.)*, J. Darzentas, Ed. Lecture notes in artifical intelligence 5138. Springer, Berlin, 51–63. DOI=10.1007/978-3-540-87881-0_6.

[4] Biedermann, D. H., Dietrich, F., Handel, O., Kielar, P. M., and Seitz, M. 2015. *Using Raspberry Pi for scientific video observation of pedestrians during a music festival*. https://www.researchgate.net/profile/Daniel_Biedermann/publication/279535242_Using_Raspberry_Pi_for_scientific_video_observation_of_pedestrians_during_a_music_festival/links/5596417b08ae21086d209c48.pdf. Accessed 7 March 2016.

[5] Christoph Urban and Bernd Schmidt. *PECS - Agent-Based Modelling of Human Behaviour*. https://www.aaai.org/Papers/Symposia/Fall/2001/FS-01-02/FS01-02-027.pdf. Accessed 2 March 2016.

[6] Handel, O. 2016. Modeling Dynamic Decision-Making of Virtual Humans. *Systems* 4, 1, 4.

[7] Handel, O., Gümüs, E., Papoutsis, E., and Amann, J. 2015. Dynamic Visualization of Pedestrian Simulation Data. In *Proceedings of the 27th Forum Bauinformatik*, Aachen, Germany.

[8] Helbing, D. and Mukerji, P. 2012. Crowd disasters as systemic failures. Analysis of the Love Parade disaster. *EPJ Data Sci.* 1, 1.

[9] Jennings, N. R., Sycara, K., and Wooldridge, M. 1998. A Roadmap of Agent Research and Development. *Autonomous Agents and Multi-Agent Systems* 1, 1, 7–38.

[10] Joo, J. 2013. Perception and BDI Reasoning Based Agent Model for Human Behavior Simulation in Complex System. In *Human-Computer Interaction. Towards Intelligent and Implicit Interaction*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum and M. Kurosu, Eds. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, 62–71. DOI=10.1007/978-3-642-39342-6_8.

[11] Lee, S. and Son, Y.-J. Integrated human decision making model under Belief-Desire-Intention framework for crowd simulation. In *2008 Winter Simulation Conference (WSC)*, 886–894. DOI=10.1109/WSC.2008.4736153.

[12] 2001. Multi-Agent Systems. In *Objective coordination in multi-agent system engineering. Design and implementation / Michael Schumacher*, M. Schumacher, Ed. Lecture notes in computer science. Lecture notes in artificial intelligence 2039. Springer, Berlin, London, 9–32. DOI=10.1007/3-540-44933-7_2.

[13] Ronald, N. and Sterling, L. Modelling pedestrian behaviour using the BDI architecture. In *IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, 161–164. DOI=10.1109/IAT.2005.104.

[14] Schmidt, B. *Modelling of Human Behaviour The PECS Reference Model*. http://www.scs-europe.net/services/ess2002/PDF/inv-0.pdf. Accessed 2 March 2016.

[15] The Anylogic Company. *Objects Networks*. www.anylogic.com/anylogic/help/index.jsp?topic=/com.xj.anylogic.help/html/agentbased/Objects Networks.html. Accessed 3 April 2016.

[16] Vosinakis, S. and Panayiotopoulos, T. 2003. Programmable Agent Perception in Intelligent Virtual Environments. In *Intelligent virtual agents. 4th International Workshop, IVA 2003, Kloster Irsee, Germany, September 15-17, 2003 : proceedings / Thomas Rist ... [et al.], (eds.)*, T. Rist, Ed. Lecture notes in computer science. Lecture notes in artificial intelligence 2792. Springer, Berlin, London, 202–206. DOI=10.1007/978-3-540-39396-2_33.

[17] Weyns, D., Steegmans, E., and Holvoet, T. 2004. Towards active perception in situated multi-agent systems. *Applied Artificial Intelligence* 18, 9-10, 867–883.

[18] Wooldridge, M. and Jennings, N. R. 1995. Agent theories, architectures, and languages: A survey. In *Intelligent agents. ECAI-94 Workshop on Agent Theories, Architectures, and Languages, Amsterdam, The Netherlands, August 8-9, 1994 : proceedings / Michael J. Wooldridge, Nicholas R. Jennings (eds.)*, M. J. Wooldridge and N. Jennings, Eds. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–39. DOI=10.1007/3-540-58855-8_1.