Klassifizierung und Entwicklung von Automatiken für Eingriffe in der Socially Augmented Microworld (SAM)

Diplomarbeit

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

eingereicht von:	Andreas Wickert
geboren am:	15. Februar 1980
in:	Berlin
Matrikelnummer:	195945
Gutachter:	Professor Dr. K. Bothe
	Professor Dr. H. Wandke
Betreuer:	Dipl. Inf. Nicolas Niestroj
eingereicht am:	7. August 2013
in:	Berlin
verteidigt am:	.....

Abstract

The project ATEO (**A***rbeits***t***eilung* **E***ntwickler* **O***perateur*) is part of the exploration focus no. 8 whithin the graduate lecture prometei (**Pro***spektive Gestaltung von* **Me***nsch-***Te***chnik-***I***nteraktion*). Inside this context an experimental environment named SAM (**S***ocially* **A***ugmented* **M***icroworld*) has been implemented. It is a complex socio-technical system that allows the exploration of interdependencies between the resources available to the system-developers and the quality of their automations for the target-system.

This work describes the development process of those automatic functions, which were invented and gathered into concepts by developers. The concepts will be analyzed to obtain identifying attributes. Their help provide the classification of the concepts and the summarization of redundant operations. Furthermore several software-tools, that support the concept implementation, will be presented. Afterwards more information about the specification of the original concepts and their functions will be described. At last the tests and their results will show, that the implemented agents fulfill the designated specifications and restrictions. Finally the work-results will get discussed and possible future improvements presented.

Zusammenfassung

Im Projekt ATEO (**A***rbeits***t***eilung* **E***ntwickler* **O***perateur*), das im Rahmen des Forschungsschwerpunktes 8 des Graduiertenkollegs prometei (**Pro***spektive Gestaltung von* **Me***nsch-***Te***chnik-***I***nteraktion*) durchgeführt wird, wurde die Versuchsumgebung SAM (**S***ocially* **A***ugmented* **M***icroworld*) entwickelt. Es handelt sich um ein komplexes soziotechnisches System, das bei der Erforschung der Abhängigkeiten zwischen Ressourcen von Entwicklern und der Qualität ihrer entwickelten Automatiken für dieses System helfen soll.

In dieser Arbeit wird der Entwicklungsprozess von Automatikfunktionen behandelt, die auf den genannten Automatikkonzepten beruhen. Dazu werden die Konzepte zunächst analysiert, um Attribute zu bestimmen, die sie klassifizieren. Außerdem wird die Entwicklung von Hilfswerkzeugen zur Umsetzung der Konzepte beschrieben. Die Konzepte werden spezifiziert, deren Funktionsweisen dargestellt und Testergebnisse präsentiert, die zeigen, dass die entwickelten Agenten die Spezifikationen und Restriktionen erfüllen. Abschließend werden die Ergebnisse der Arbeit diskutiert und mögliche zukünftige Erweiterungen vorgestellt.

Hinweis: Aus Gründen der besseren Lesbarkeit wird auf die gleichzeitige Verwendung männlicher und weiblicher Sprachformen verzichtet. Sämtliche Personenbezeichnungen gelten gleichermaßen für beide Geschlechter.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Akronyme

AAF	**A***TEO* **A***utomation* **F***ramework* 2, 3, 12, 13, 15, 19, 22–24, 70, 103
AAFGT	**AAF G***raph* **T***ool* 19, 99
AMD	**A***TEO* **M***aster* **D***isplay des Operateursarbeitsplatzes* 3, 4, 22
ATEO	**A***rbeits***t***eilung* **E***ntwickler* **O***perateur* 1, 2, 6, 7, 9, 11, 21, 23, 24, 36, 39, 55, 69, 85, 99
CSV	**C***omma-***S***eparated* **V***alues* 42, 43, 47–49, 98, 106–111
CTE XL	**C***lassification* **T***ree* **E***ditor* **XL** 12, 27
GUI	*graphische Benutzungsoberfläche beziehungsweise* **G***raphical* **U***ser* **I***nterface* 3, 18, 74, 78–80, 98
IDE	**I***ntegrated* oder **I***nteractive* **D***evelopment* **E***nvironment* 6, 9
IfI	**I***nstitut* **f***ür* **I***nformatik (Lehrstuhl für Software-Technik) der Humboldt-Universität zu Berlin* 11
IfP	**I***nstitut* **f***ür* **P***sychologie (Lehrstuhl für Ingenieurpsychologie und Kognitive Ergonomie) der Humboldt-Universität zu Berlin* 11
OA	**O***perateurs***a***rbeitsplatz* 2, 4, 5, 12, 21, 22, 24
prometei	**Pro***spektive Gestaltung von* **Me***nsch-***Te***chnik-***I**nteraktion* 1
SAM	**S***ocially* **A***ugmented* **M***icroworld* 2–7, 11–15, 17–21, 23–25, 51, 61–63, 69–71, 74, 75, 79, 90, 92, 98, 99, 104

Kapitel 1

Einleitung

1.1 ATEO - ein Beitrag zum Graduiertenkolleg prometei

Das interdisziplinäre Graduiertenkolleg **Pro***spektive Gestaltung von* **Me***nsch-***Te***chnik-***I**nteraktion* (prometei) wird vom Zentrum für Mensch-Maschnine-Systeme an der Technischen Universität Berlin betreut. Die Verantwortung für Forschungsschwerpunkt 8 mit dem Titel *Funktionsteilung Mensch-Maschine und Arbeitsteilung Entwickler-Operateur: Zwei Perspektiven auf Mensch-Maschine-Systeme* trägt Prof. Dr. Wandke vom *Lehrstuhl für Ingenieurpsychologie und Kognitive Ergonomie an der Humboldt-Universität zu Berlin.* Innerhalb dieses Forschungsschwerpunktes wird das Projekt **A***rbeits***t***eilung* **E***ntwickler* **O***perateur* (ATEO) durchgeführt. Das erklärte Projektziel von ATEO hat sich mit jedem Phasenwechsel weiterentwickelt. In früheren Phasen des Projektes (ATEO 1.0, 2004 bis 2007 - ATEO 2.0, 2007 bis 2010) wurde untersucht, „welche internen und externen Ressourcen“ für Entwickler und Operateure „ihre jeweilige Leistungsfähigkeit ausmachen“ [prometei (2006–2012)]. Das heißt, es wurden nicht die Leistungen von Menschen mit denen von Maschinen ins Verhältnis gesetzt, „sondern die Leistung zweier Personengruppen zu unterschiedlichen Zeiptunkten empirisch verglichen“ [prometei (2006–2012)]. Es werden also die Leistungen der Entwickler, die zu einem früheren Zeitpunkt die Interaktion von Operateuren mit einem automatisierten System antizipieren und planen müssen, mit den Leistungen der Operateure, die zu einem späteren Zeitpunkt mit den von den Entwicklern implementierten automatischen Systemen in Echtzeit interagieren müssen, verglichen.

Die aktuelle Projektphase ATEO 3.0 verfolgt „Ansätze kooperativer Automation bzw. dynamischer Funktionsallokation“ [prometei (2006–2012)]. Damit ist gemeint, dass sich Operateur und Automation je nach Situation, Funktion und / oder Aufgabe die Verantwortung gegenseitig übertragen können. Das heißt, in einer bestimmten Situation gibt die Automatik beispielsweise Verantwortung an den Operateur zurück, um beispielsweise dessen Situationsbewusstsein in Anspruch zu nehmen, oder in einer anderen Situation überträgt der Operateur der Automatik die Verantwortung (oder einen Teil davon), um sich selbst von im-

mer wiederkehrenden Handlungen zu entlasten. Durch die geänderte Aufgabenstellung ergeben sich für Entwickler und Operateure neue Herausforderungen. Entwickler müssen nun für eine reibungslose Kommunikation und Kooperation zwischen Operateur und Automation sorgen. Operateure hingegen müssen lernen, mit den neuen Möglichkeiten korrekt umzugehen. Dazu ist zusätzliches Wissen darüber notwendig, wie sich die zur Verfügung stehenden Automationen auswirken und in welchen Situationen es sinnvoll ist, sie einzusetzen.

Um die zuvor genannten Leistungen von Operateuren mit denen von Entkicklern messen und anschließend vergleichen zu können ([Wandke und Nachtwei (2008)], [ATEO (2008)]), wurde die Versuchsumgebung **S***ocially* **A***ugmented* **M***icroworld* (SAM) entwickelt, in der die entwickelten Automationen zum Einsatz kommen. Abbildung 1.1 zeigt alle Komponenten und Personen, die an ATEO beteiligt sind.

Aufgaben der dargestellten Personen(gruppen):

Entwickler antizipieren Operateur- und Systemverhalten und konzipieren Automatiken zur Unterstützung der Mikroweltbewohner beziehungsweise der Operateure.

Operateure überwachen das System SAM und beeinflussen es über das ATEO Master Display.

Informatiker implementieren, warten, erweitern und verbessern die Softwarekomponenten.

Mikroweltbewohner steuern das Trackingobjekt und erhöhen durch ihr Verhalten die Komplexität von SAM.

Wissenschaftler führen Untersuchungen durch.

Für diese Arbeit sind besonders die Komponenten SAM, welches das Herzstück des gesamten Versuchsaufbaus darstellt, und das AAF von Bedeutung. Diese werden in den folgenden Abschnitten einleitend beschrieben.

1.1.1 Socially Augmented Microworld (SAM)

Bei SAM handelt sich um ein - wie Sommerville in [Sommerville (2011)] es nennt - *soziotechnisches System*, das sowohl aus Hard- und Software-Subsystemen besteht, als auch menschliche Akteure beinhaltet. Das Prinzip von SAM besteht darin, dass ein kreisrundes Trackingobjekt eine vorgegebene Strecke entlang manövriert werden soll (vergleiche dazu Abbildung 1.2).

Dieser Prozess wird zum Teil auch von Operateuren und / oder Automatiken steuernd beziehungsweise regelnd begleitet. Eine Sicht auf den Versuchsaufbau mit SAM und dem von [Schwarz (2009)] geschaffenen **O***perateurs***a***rbeitsplatz* (OA) ist in Abbildung 1.3 dargestellt. Das zu überwachende System (SAM) und der OA sind zwar räumlich voneinander getrennt aber über ein Netzwerk miteinander verbunden. Die Mikroweltbewohner (beschriftet in der genannten Abbildung mit *MWB 1* und *MWB 2*) steuern gleichzeitig das Trackingobjekt. Sie dürfen während des gesamten Vorgangs nicht miteinander kommunizieren. Die Strecke und das Trackingobjekt werden auf dem *SAM-Display* angezeigt.

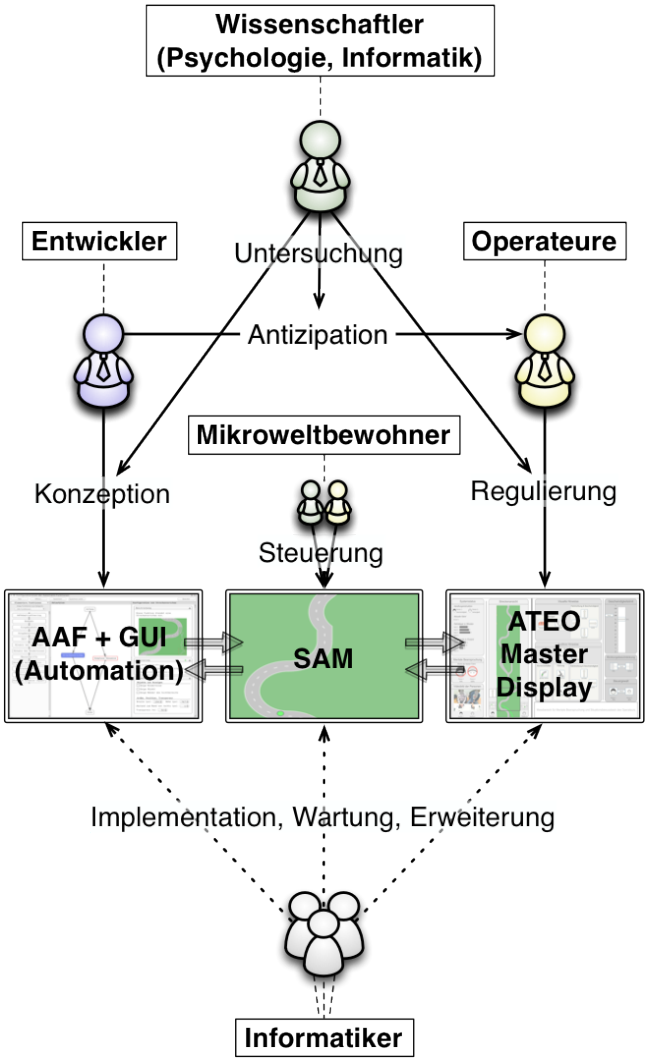


Abbildung 1.1: Beziehungen zwischen den einzelnen Rollen untereinander und zu den beteiligten Software-Komponenten von ATEO-SAM (**A***TEO* **A***utomation* **F***ramework* (AAF) / *graphische Benutzungsoberfläche beziehungsweise* **G***raphical* **U***ser* **I***nterface* (GUI), SAM und **A***TEO* **M***aster* **D***isplay des Operateursarbeitsplatzes* (AMD)) existieren. (eine weiterentwickelte Grafik aus der Diplomarbeit von [Kosjar (2012)])

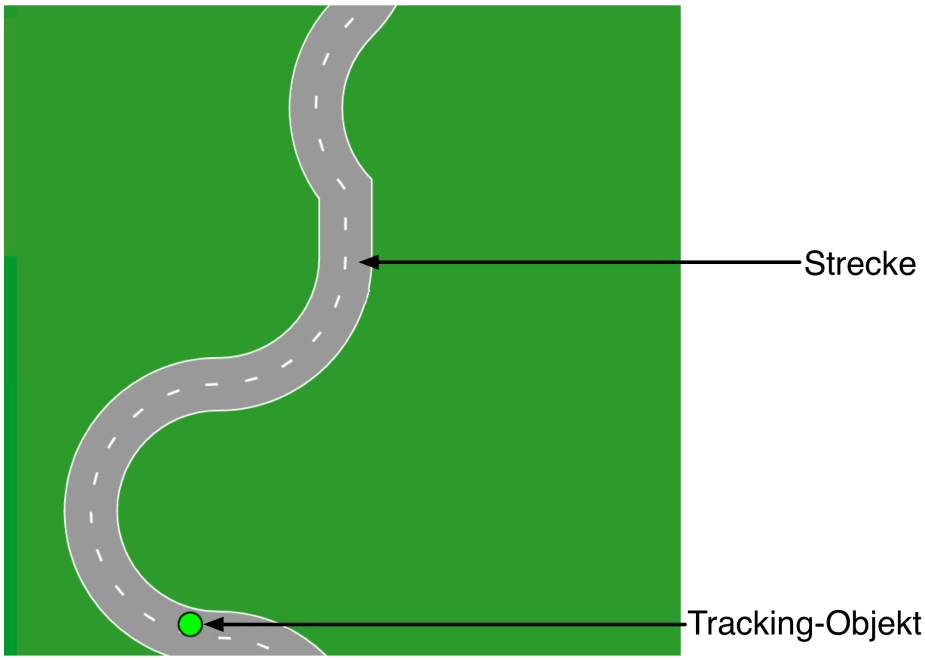


Abbildung 1.2: Ein Ausschnitt aus dem SAM-Prozess: Zu sehen sind das Trackingobjekt auf einem Streckenabschnitt.

SAM selbst wird auf dem Rechner *ATEO 1* ausgeführt. Zudem werden den Mikroweltbewohnern auf ihren Laptops zusätzliche Informationen dargeboten. Die Webcam dient dem Operateur für einen weiteren „Blickwinkel ins System“, über den er den Stresszustand der Mikroweltbewohner beobachten kann. Die Operateuranwendung wird auf dem Rechner *ATEO 2* ausgeführt und auf dem AMD angezeigt. Dem Operateur werden auf diese Weise diverse Informationen über den aktuellen Zustand des Systems (SAM) mitgeteilt und Möglichkeiten in das System einzugreifen zur Verfügung gestellt.

Alle notwendigen Informationen über die Vernetzung von SAM und OA sind der Arbeit von [Niestroj (2009)] zu entnehmen.

1.1.2 ATEO Automation Framework (AAF)

Wie bereits erwähnt, kann der Trackingprozess in SAM von Automatiken unterstützt werden. Diese Unterstützung kann ganz unterschiedlicher Natur sein. Eine Übersicht über die Vielfalt der möglichen Attribute einzelner Automatikfunktionen gibt Abbildung 3.3 in Abschnitt 3.5, wo die einzelnen Unterschiede ausführlich beschrieben werden. An dieser Stelle sei nur erwähnt, dass Automatikfunktionen entweder in Form von Hinweisen oder Eingriffen in die Objektsteuerung aktiv werden können.

Damit eine einheitliche Schnittstelle zur Implementierung von Automatikfunktionen eingehalten werden kann, wurde von [Hasselmann (2013)] ein Framework

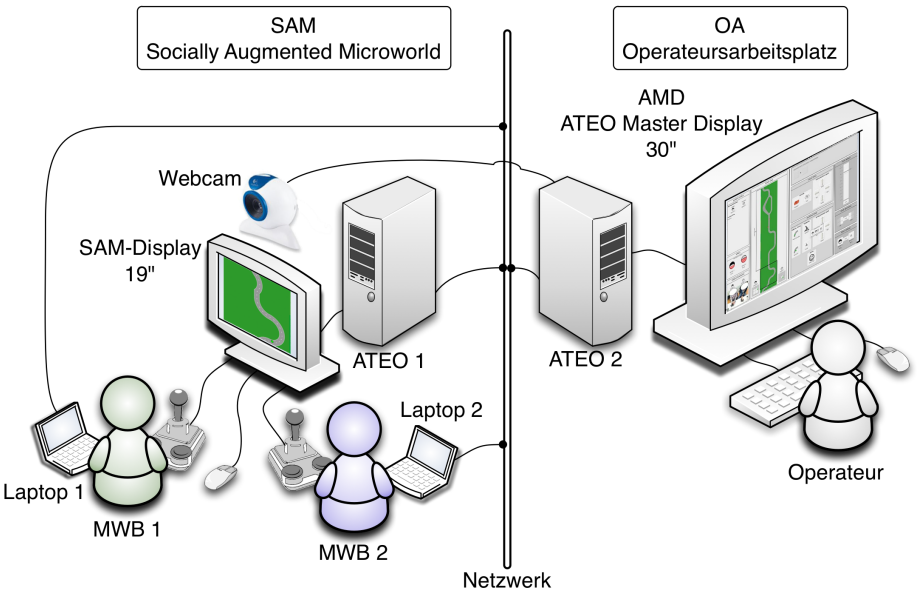


Abbildung 1.3: Versuchsaufbau: SAM mit OA. Diese Abbildung entstand auf Basis der *figure 2* in [Nachtwei (2010)].

entwickelt. Weitere technische Details zu diesem Thema mit Relevanz für diese Arbeit werden in Abschnitt 2.5 behandelt.

1.1.3 ATEO Versuchsreihen

Als Referenz für alle folgenden Versuche wurde ein Single-Tracking durchgeführt, wobei ein (menschlicher) Mikroweltbewohner das Trackingobjekt *ohne* Unterstützung durch den Parcours steuerte. Um die Komplexität von SAM zu erhöhen, wurde ein zweiter Mikroweltbewohner dem System hinzugefügt, wodurch es eine nichtdeterministische Note erhielt. Durch den zweiten Mikroweltbewohner wurde die Vorhersagbarkeit des Systems weiter herabgesetzt, da beide Mikroweltbewohner den gleichen Einfluss auf das Trackingobjekt ausüben, was zum Beispiel bei gegensätzlichem Lenkverhalten dazu führt, dass sich die Eingaben gegenseitig aufheben und sich das Trackingobjekt einfach geradeaus bewegt. Zudem haben beide Mikroweltbewohner unterschiedliche Instruktionen: Der eine hat den Auftrag, zwar schnell, aber dabei besonders genau zu steuern. Der andere hat einen komplementären Auftrag: Er soll genau, aber dabei besonders schnell fahren. Hierdurch wird der Konflikt zwischen den beiden Mikroweltbewohnern noch verstärkt ebenso wie der nichtdeterministische Teil des Systems. Alle bisher durchgeführten Versuche werden in Tabelle 1.1 aufgezeigt.

Unterstützung	ohne	Operateur	Automatik	Opermatik
SingleTracking	✓ (ATEO 1.0)	—	—	—
MultiTracking	✓ (ATEO 2.0)	✓ (ATEO 2.0)	◯ (ATEO 3.0)	◯ (ATEO 3.0)

Tabelle 1.1: Diese Tabelle zeigt die Versuchsdurchführungsmatrix. Die mit Kreisen versehenen Versuchskombinationen befinden sich in Planung.

1.2 Situations- und Problembeschreibung

Im Zuge von Saskia Kains Dissertation [Kain] soll nicht mehr die Leistung eines Operateurs mit der Leistung von Automatiken verglichen werden. Denn eine Automatik ist immer nur so gut, wie die Antizipation des zu automatisierenden Prozesses durch den Automatik-Entwickler.

Antizipationsfähigkeit wird beeinflusst durch einige Ressourcen: Interdisziplinarität, Erfahrung, Kontakt mit dem System wirken leistungssteigernd. Die Ressourcen Informationsmenge und Erfahrung sowie Zeitkomponente wurden variiert. Zusammen mit einem weiteren Untersuchungsfeld, bei dem der Operateur und dessen Ressourcen im Mittelpunkt stehen, sollen kompetenzfördernde und Zuverlässigkeit steigernde Maßnahmen zur Funktionsteilung zwischen Mensch und Maschine ermittelt werden.

In Kains Experiment bekamen 30 3-köpfige Entwicklerteams, die jeweils unterschiedliche Mengen der genannten Ressourcen zur Verfügung hatten, die Aufgabe, Automatikkonzepte für ATEO SAM zu entwerfen.

1.3 Zielstellung

Die im vorigen Abschnitt angesprochenen Entwicklerteams ersonnen 30 Konzepte und schrieben diese auf einheitlichen Konzept-Formularen nieder. Kain stellte die darin enthaltenen Informationen in Form einer Tabelle und von den Teams per Hand angefertigte Skizzen digital zur Verfügung.

Das Hauptziel dieser Arbeit ist die Implementierung der Automatikfunktionen, die sich auf die Steuerung des Trackingobjekts auswirken. Nur wenige Teams wollen den Mikroweltbewohner gar keine eingreifenden Hilfen geben. Die Mehrheit entschied sich aber unter Befolgung der Vorgabe, dass keine Vollautomatik konzipiert werden soll, auf die Steuerung einzuwirken.

Um das Hauptziel zu erreichen bedarf es zunächst einer Analyse der vorliegenden Konzepte. Ziel der Analyse ist, Konzepte zu klassifizieren, die gegebenenfalls nur mit unterschiedlichen Worten die selbe (oder eine ähnliche) Funktion meinen. Wenn diese Klassen identifiziert sind, können einzelne Komponenten derart umgesetzt werden, dass sie durch etwaige Parameter die Funktionen unterschiedlicher Konzepte erfüllen. Für die leichtere Konfiguration der Automatik-Parameter sollen eigens dazu passende Einstellungsdialoge geschaffen werden.

Manuelle und automatische Tests sollen zeigen, dass die Implementationen der

Konzepte korrekt funktionieren und zudem den Wirkungs-Vorstellungen der Entwickler entsprechen. Weiterhin müssen die Einstellungsdialoge die vom Benutzer vorgegebenen Werte fehlerfrei an die Automatikfunktionen übertragen und deren eingestellten Werte in geeigneter Form und korrekt anzeigen.

Für die Umsetzung muss die Smalltalk-**I***ntegrated* oder **I***nteractive* **D***evelopment* **E***nvironment* (IDE) Squeak (beschrieben in Abschnitt 2.2) verwendet werden, welche im ATEO Projekt bereits seit vielen Jahren eingesetzt wird.

1.4 Übersicht über den Aufbau der Arbeit

Kapitel 2 behandelt grundlegendes Wissen, dass zum Verständnis der vorliegenden Arbeit zwingend erforderlich ist. Es wird auf die verwendete Programmiersprache und Software-Entwicklungsumgebung eingegangen. Außerdem wird ein detaillierter Einstieg in das Thema ATEO SAM geliefert und welche Voraussetzungen bestanden, bevor diese Arbeit entstand.

In Kapitel 3 werden die Ergebnisse der Konzept-Analyse vorgetragen. Es werden Attribute ermittelt, die zur Klassifizierung der Entwicklerkonzepte verwendet werden und ungenaue, fehlende sowie widersprüchliche Informationen in den Konzepten aufgeführt.

Kapitel 4 stellt zum einen Hilfswerkzeuge vor, die bei den Konzeptumsetzungen von entscheidender Bedeutung waren, und zum anderen werden Komponenten vorgestellt, die in vielen Agenten zum Einsatz kommen.

In Kapitel 5 werden detailliert die interessanten Algorithmen vorgestellt, die letztendlich die Funktionalität der einzelnen Agenten darstellen.

Auf welche Weise die Automatikfunktionen getestet und von den meisten Fehlern befreit wurden, wird in Kapitel 6 vorgestellt.

Und schließlich wird in Kapitel 7 die Arbeit resümiert und kritisch hinterfragt.

In Ergänzung zur Arbeit wird in mehreren Anhängen zusätzlich darüber informiert, welche Schnittstellen in einem Agenten[1] implementiert werden müssen (Anhang A), welche Methoden in zusätzlichen Klassen implementiert wurden (AAFTrackInfo: B.1; AAFTrackTools: B.2; AAFBoundTools: B.3), und wie die entwickelten Werkzeuge zu bedienen sind (Streckenanalyse-Werkzeug: C.1; Codegenerator: C.2; Datenstruktur-Erweiterung: C.3). Im letzten Anhang (D) wird der Inhalt der beigefügten DVD aufgelistet.

[1] synonym zu Automatikfunktion

Kapitel 2

Grundlagen

Dieses Kapitel dient sowohl der Einführung und Erklärung einiger Begriffe und Bezeichnungen, die in dieser Arbeit häufig Verwendung finden, als auch der Eingrenzung des Themas.

2.1 Smalltalk

Die ursprüngliche Idee Alan Kays war es, eine Sprache zu entwickeln, mit denen selbst Laien programmieren konnten, um dadurch „beliebige Informationsbedürfnisse und kreatives Denken zu unterstützen“ [Stritzinger (1999), 2.1]. So begann Ende der sechziger, Anfang der siebziger Jahre die Enwicklung von Smalltalk am Xerox PARC (Palo Alto Research Center). Da Smalltalk auf seiner eigenen Hardware als Betriebssystem zum Einsatz kam, besitzt es viele Betriebssystem-typische Merkmale (aus [Sharp (1997), Introduction - About Smalltalk] und [Goldberg und Robson (1983), Preface]):

- verzweigte Prozesse
- Semaphore - unter anderem für das Ausschließungsphänomen (mutual exclusion effect)
- automatische Speicherverwaltung
- ein eigenes Dateisystem
- Darstellungsverwaltung
- Text- und Bildbearbeitung
- Tastatur- und Zeigegeräteingaben (zum Beispiel Maus oder Grafikstift)
- Werkzeuge zur Fehlerbeseitigung (Debugger)
- Werkzeuge zur Leistungsüberwachung (Performance Spy)
- Prozessor Verwaltung (Processor Scheduling)
- Werkzeuge zum Kompilieren und Dekompilieren

Nach mehreren Entwicklungszyklen, an denen die Entwickler Adele Goldberg, Daniel Ingalls und Alan Kay maßgeblich beteiligt waren, wurde 1981 *Smalltalk-80* veröffentlicht [Goldberg und Robson (1983), Preface].

Smalltalk ist aber oft vielmehr als nur eine objektorientierte Programmiersprache. Häufig ist die Rede von *Smalltalk-Systemen*, da zusammen mit der Programmiersprache auch eine passende IDE sowie eine virtuelle Maschine, die zur Ausführung der Smalltalk-Programme dient, mitgeliefert werden. Die bekanntesten kommerziellen Smalltalk-Systeme sind *Visual Works* und *Visual Smalltalk* (beide von ParcPlace-Digitalk) sowie *Visual Age für Smalltalk* von IBM. Ein Open-Source-System ist *Squeak*, das im ATEO Projekt verwendet wird (siehe Abschnitt 2.2).

Bei Smalltalk wurde die Idee der objektorientierten Programmierung äußerst strikt verfolgt. Im Prinzip gibt es nur fünf Vokabeln, die Smalltalks Hauptkonzepte bezeichnen: *Objekt, Nachricht, Klasse, Instanz* und *Methode* [Goldberg und Robson (1983)]. Die Objektorientierung bei Smalltalk geht soweit, dass alle Daten einheitlich als Objekt repräsentiert werden. Sogar Klassen werden als Objekte dargestellt, welche wiederum durch Metaklassen beschrieben werden. Variablen werden dynamisch typisiert. Das heißt, der Typ einer Variablen wird erst während der Laufzeit festgelegt. Details zu diesen und weiteren Konzepten von Smalltalk können in [Stritzinger (1999)] im Abschnitt *2.3 - Grundkonzepte der Sprache* nachgelesen werden.

2.2 Squeak

Squeak ist ein Open-Source Smalltalk-System, bestehend aus mehreren separaten Komponenten, die im Zusammenspiel das *SqueakSystem* bilden (dargestellt in Abbildung 2.1). Die Geschichte von Squeak beginnt bei Apple, wo es von einer Forschungsgruppe entwickelt wurde, die unter anderem als Entwicklungsziel die Erschaffung von Prototypen für Lernsoftware und Benutzungsschnittstellen-Experimente, unter Verwendung einer expressiven und direkten Programmiersprache wie Smalltalk, verfolgte. Squeak enthält Klassenbibliotheken und Erweiterungen für die virtuelle Maschine (VM) für diverse Multimedia-Anwendungsfälle, darunter 2D-Kantenglättung (anti-alised 2D), 3D Beschleunigung, Audio- und Musik-Synthese in Echtzeit und mehr als 600 andere Erweiterungen.[1]

Die Squeak-VirtualMachine
Sie ist das Herzstück des Systems. Mit ihrer Hilfe wird das Arbeiten mit Squeak erst möglich. Die *SqueakVM* funktioniert aber nur zusammen mit den *Squeak Quellen (Sources)* und einem *Squeak Image* (plus der zugehörigen *Changes-Datei*, in der alle Änderungen protokolliert werden).

Die Squeak Quellen
Diese Bibliothek beinhaltet die grundlegenden Funktionen von Squeak. Alle Standard-Klassen und deren Methoden sind hierin gespeichert und werden beim Start der Squeak-VM geladen und stehen während des Ausführungszeit zur Verfügung.

Das Squeak-Image

[1] http://squeak.org, About-Seite

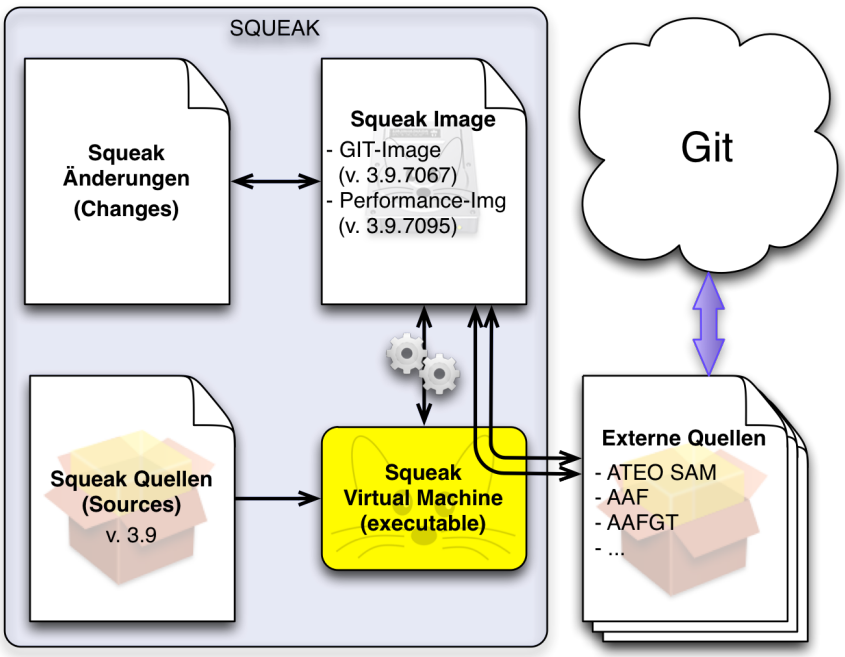


Abbildung 2.1: Die Komponenten des Squeak-Systems und deren Anbindung an das Versionenkontrollsystem Git (siehe Abschnitt 2.3).

Es stellt eine Art Wechseldatenträger, vergleichbar mit einer virtuellen Festplatte, dar, der von der Squeak-VM eingelesen und verwendet wird. In einem Image wird mit einer Sicherung immer der aktuelle Entwicklungsstand der zu entwickelnden Software gespeichert. Mit 'Sicherung' ist ein globales *Save* gemeint, nicht jedoch das Abspeichern von Änderungen. Letzere werden nur in der *Changes-Datei* gesichert und erst mit einem globalen Save ins Image geschrieben und damit persistent gemacht.

Die externen Quellen
Zusätzlich zu den herkömmlichen Squeak Quellen gibt es im ATEO Projekt auch noch die externen Quellen, welche das gesamte SAM System und alle zugehörigen Komponenten enthalten. Um sie nutzen zu können, müssen sie manuell in das Squeak-Image geladen werden. Änderungen, die im ATEO Git (siehe Abschnitt 2.3) veröffentlicht werden sollen, müssen zunächst manuell aus dem Squeak-Image exportiert werden. Anschließend sind die Squeak-Quelldateien (erkennbar an der Erweiterung 'st') bereit im Git aktualisiert oder eingestellt zu werden.

2.3 Versionenkontrollsystem Git

Die Entwicklung von Software findet vielerorts dezentral, das heißt verteilt, statt. Um dabei nicht den Überblick zu verlieren, ist der Einsatz eines Versionen-Kontrollsystems unabdingbar. Im ATEO Projekt fiel die Wahl auf ein Open-Source-System, das den Namen *Git* trägt. Über die Anwendung von Git soll hier nicht weiter eingegangen werden. Für weitere Informationen kann die Git-Dokumentation auf der Entwickler-Webseite[2] zu Rate gezogen werden.

2.4 Social Augmented Microworld (SAM) im Detail

Zu Beginn der Entwicklung der Software für ATEO wurde unsystematisch und ungeplant vorgegangen. Erweiterungen und Änderungen wurden anfangs ad-hoc von einem einzigen Entwickler des **I***nstitut* **f***ür* **P***sychologie (Lehrstuhl für Ingenieurpsychologie und Kognitive Ergonomie) der Humboldt-Universität zu Berlin* (IfP) verwirklicht. Da sich stetig die Anforderungen an die Software änderten oder neue Funktionen benötigt wurden, entstand eine äußerst chaotische Software, die ohne jegliche Struktur, Architektur und Dokumentation gewachsen war. Viel Quelltext wurde einfach stehengelassen, obwohl er nicht weiter benötigt wurde. Der Grund dafür war, dass man sich auf diese Weise die Möglichkeit bewahren wollte, jederzeit zu einer früheren Version zurückkehren zu können. Die einzigen Dokumente, die in dieser ersten Entwicklungszeit entstanden, waren lediglich eine Bedienanleitung von [Gross und Sacklowski (2007)] und eine Beschreibung des Aufbaus der Protokolldatei.

Aufgrund der begrenzten Ressourcen, die dem IfP zur Verfügung standen, wandte man sich an das **I***nstitut* **f***ür* **I***nformatik (Lehrstuhl für Software-Technik) der*

[2] http://git-scm.com

Humboldt-Universität zu Berlin (IfI). Daraufhin entstand eine interdisziplinäre Zusammenarbeit in Form einer Projektseminarreihe. In der ersten Auflage des Projektseminars [Bothe u. a. (2007–2013)] aus dem Wintersemester 2007/2008 wurde ein Reverse Engineering der bestehenden Software durchgeführt. Daraus entstanden eine Software-Spezifikation in Form eines Pflichtenhefts (aktuelle Version: [Bothe u. a. (2009)]) inklusive Use-Cases, eine Übersicht über die zu Grunde liegende Software-Architektur und dem vorhandenen Quellcode wurden Kommentare hinzugefügt.

In weiteren Veranstaltungen der Seminarreihe wurden unter anderem die folgend in chronologischer Reihenfolge aufgelisteten Arbeiten durchgeführt.

2007/2008:

- Reverse Engineering der SAM v. 1.5.
- Erstellen einer Software-Spezifikation (Pflichtenheft) einschließlich Use Cases.
- Kommentieren des bestehenden Smalltalk Quelltextes.
- Ermitteln der Software-Architektur.

2008/2009:

- Einrichtung eines Versionenkontrollsystems (siehe Abschnitt 2.3)
- Entfernung von unbenutztem Quelltext („toter Code“)
- Umsetzung einer neuen Software-Architektur: SAMs 2.0; Details können in der Studienarbeit [Hildebrandt (2009)] nachgelesen werden.
- Testfallermittlung mit der Klassifikationsbaummethode; verwendetes Tool: **C***lassification* **T***ree* **E***ditor* **XL** (CTE XL)[3]
- Testfallimplementierung mit dem SUnit-Testframework
- Review bestehender Dokumente

2009/2010:

- Review bestehender Dokumente und Anpassen an die neue Architektur
- Neue Testfälle für die aktuelle SAMs-Version 2.2 identifiziert und implementiert - wieder mit SUnit und CTE XL

2010/2011 sowie *2011/2012:*

- Mehrere studentische Entwicklerteams fertigen eine Analyse der Anwendungsdomäne (SAM v. 2.3) an
- Entwurf eigener Konzepte für Automatik-Funktionen
- Erstellen eines objektorientierten Softwareentwurfs im Rahmen des AAF.
- Implementieren der Entwürfe.

Aus den Seminaren gingen auch zahlreiche Studien- beziehungsweise Diplomarbeiten hervor, deren Gegenstand die Ergänzung oder Erweiterung von SAM um

[3] http://www.berner-mattner.com/de/berner-mattner-home/produkte/cte/

weitere Module war. Beispielsweise wurden eine Geschwindigkeitsanzeige, Inputmanipulation und Hindernisse in der Studienarbeit von Nicolas Niestroj (2008) realisiert. Auch der bereits genannte OA [Schwarz (2009)] und dessen Vernetzung mit SAM und dem AAF [Niestroj (2009)] waren Themen weiterer Arbeiten. Das Vernetzungsschema ist in Abbildung 1.3 zu sehen. Kürzlich wurden die Diplomarbeiten von [Kosjar (2012)] und [Seid (2012a)] fertiggestellt. Sie behandeln visuelle und auditive Hinweis- und Informations-Automatikfunktionen sowie die Schaffung eines Eventsystems.

2.4.1 Experiment, Versuchsschritt, Simulationsschritt

Mit der Versuchsumgebung SAM können vielfältige Experimente durchgeführt werden. Diese werden über eine Konfigurationsdatei[4] definiert. Diese Datei enthält eine bestimmte Anzahl >= 1 an Zeilen, wobei jede einzelne Zeile für einen *Versuchsschritt*[5] steht. Jeder Step enthält folgende Attribute:

- Versuchsschrittnummer
- Mikroweltbewohner -Konfiguration:
  - 1 = nur Mikroweltbewohner 1 steuert
  - 2 = nur Mikroweltbewohner 2 steuert
  - 12 = beide Mikroweltbewohner steuern
- Name der Streckenkonfigurationsdatei
- Hinderniskonfiguration (obstacleConfig_0 bis 9)
- optional: Zum Einsatz kommende (komplexe) Automatik (Name der AAF-Konfigurationsdatei)

Die Streckenkonfigurationsdatei enthält eine Liste mit Kachel-Namen (siehe Unterabschnitt 2.4.2). Der erste Eintrag in der Liste entspricht der Startkachel, es folgt eine Menge weiterer Kachel-Namen, der drittletzte Eintrag entspricht der Zielkachel und die letzten beiden dienen als Pufferzone, damit bei Erreichen des Ziels kein weißer Balken am oberen Bildrand entsteht. Listing 2.1 zeigt die Syntax für die Konfigurationsdatei steps.txt und Listing 2.2 enthält ein kurzes Beispiel, wie deren Inhalt aussehen kann.

```
1 <step-number>';'<mwi-control>';'<track>';'<obstacle-
    configuration>[';'<automatic-file>]
```

Listing 2.1: Syntax der Konfigurationsdatei steps.txt

```
1 1;  1; lernstrecke; obstacleConfig_0
2 2;  2; lernstrecke; obstacleConfig_0
3 3; 12; testabschnitt; obstacleConfig_1
4 4; 12; hauptabschnitt_lang; obstacleConfig_2;
    Leitplanken
```

[4] steps.txt
[5] synonyme Verwendung: *Step*

```
5; 12; hauptabschnitt_lang; obstacleConfig_7;
   Hindernisvermeidung
...
```

Listing 2.2: Beispiel-Inhalt der Konfigurationsdatei steps.txt

Während das Experiment läuft, wird jeder Step nacheinander simuliert. Indem einzelne *Simulationsschritte*[6] hintereinander ausgeführt werden wird der Eindruck einer Bewegung erzeugt. Dazu wird in jedem Tick eine Reihe von zyklischen Operationen ausgeführt. Eine Flussdiagramm-Darstellung der SAM-Hauptschleife gibt Abbildung 2.2 wieder.

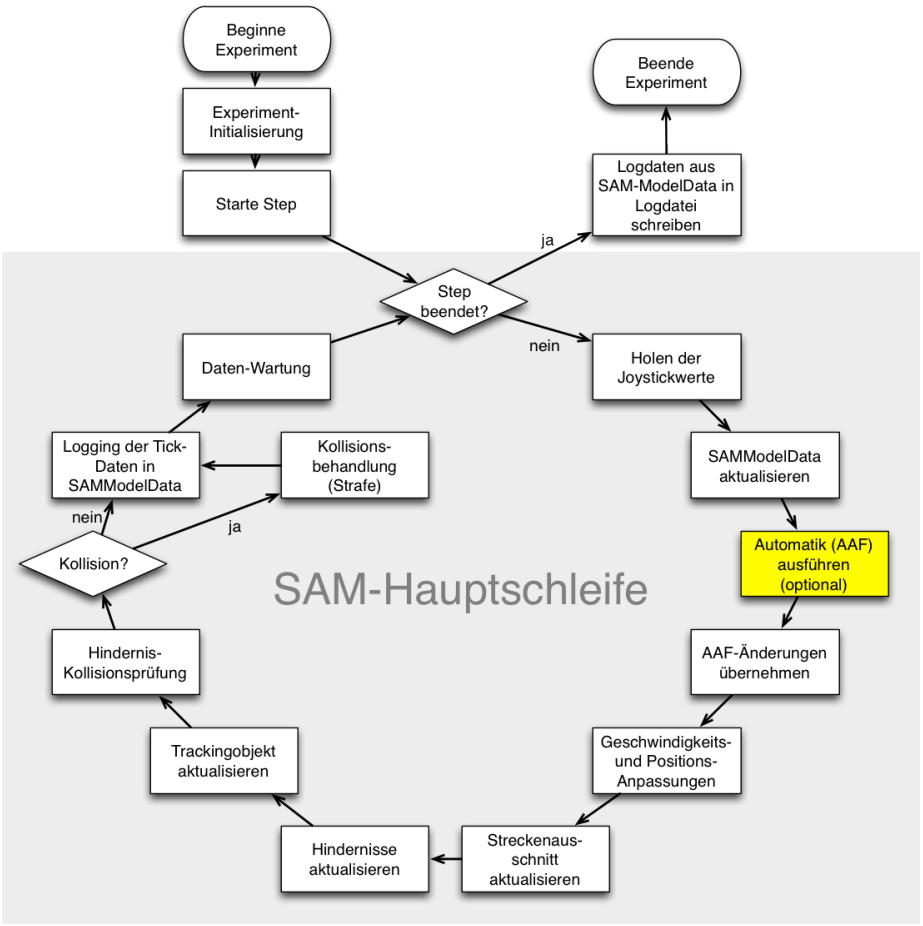


Abbildung 2.2: Diese Flussdiagramm stellt den Ablauf in der SAM-Hauptschleife dar.

Solange das Ende einer Strecke nicht erreicht ist, werden zu Beginn eines Ticks die aktuellen Joystickwerte empfangen und in die Datenstruktur SAMModel-Data geschrieben. Der folgende Schritt ist durch gelbe Färbung hervorgehoben:

[6] synonyme Verwendung: *Ticks*

Hier findet die Ausführung der (optional) konfigurierten Automatik statt. Sie hat für diese Arbeit besondere Relevanz. Nachdem die Automatik ihre Berechnungen und Aktionen durchgeführt hat, werden die zurückgelieferten Daten im SAM-Prozess aktualisiert und die neue Geschwindigkeit beziehungsweise Position des Trackingobjekts werden berechnet. Die nächsten Schritte dienen der grafischen Aktualisierung: Der sichtbare Streckenausschnitt, eventuell anzuzeigende Hindernisse und das Trackingobjekt selbst werden neu gezeichnet. Danach wird geprüft, ob eine Kollision mit einem Hindernis aufgetreten ist. Ist dies der Fall, so wird das Trackingobjekt neben die Strecke gesetzt und eine Strafsekunde gewartet, bevor der Prozess fortgesetzt wird. Anschließend werden alle Prozessdaten geloggt. Zuletzt finden noch einige Wartungsarbeiten[7] an den Datenstrukturen des Prozesses statt, um die Tick-Ausführungszeit von 40 ms konstant einzuhalten, bevor der Zyklus von vorne beginnt.

2.4.2 Streckenelemente

Für die in Unterabschnitt 2.4.1 erwähnten Steps wird jeweils eine Strecke definiert. Eine solche Strecke setzt sich aus einer beliebigen Anzahl von Streckenelementen (auch *Tiles* oder *Kacheln* genannt) zusammen. Die zur Zeit existierende Menge an Kacheln umfasst 34 Elemente. Die einzelnen Kacheln enthalten Kurven, Gabelungen sowie Geraden. Zudem gibt es eine Kachel, welche als Ziellinie fungiert. Während sich die Geraden nur durch ihre Länge unterscheiden, gibt es vier verschiedene Kurvenarten je Richtung (links beziehungsweise rechts, dargestellt in Abbildung 2.3) und vier verschiedene Gabelungen: Runde Gabelungen mit breitem Arm links beziehungsweise rechts und eckige Gabelungen mit breitem Arm links beziehungsweise rechts (dargestellt in Abbildung 2.4).

Zudem gibt es noch Hindernisse, die in der Step-Konfiguration aktiviert und positioniert werden können. Grundsätzlich wird zwischen *dynamischen* und *statischen* Hindernissen unterschieden. Während sich dynamische Hindernisse während des Trackings (der Fahrt auf einer Strecke) zu einem bestimmten Zeitpunkt von links nach rechts über die Strecke bewegen, befinden sich statische Hindernisse an einer fixen Position auf der Strecke. Dabei ist die Geschwindigkeit des dynamischen Hindernisses gerade so gewählt, dass es zwangsläufig – ohne Änderung der Geschwindigkeit und Richtung des Trackingobjekts durch die Mikroweltbewohner– zur Kollision kommt. Statische Hindernisse ragen entweder zu 25 oder 50 Prozent auf die Fahrbahn. Sie kommen ausschließlich paarweise vor wobei das eine Hindernis die Fahrbahn von links und das andere von rechts bedeckt. Welche Reihenfolge dieser *Slalom* hat, kann ebenfalls konfiguriert werden. Derzeit kommen aber nur links-rechts-Slaloms zum Einsatz (das erste Hindernis bedeckt die Fahrbahn auf der linken Seite), sodass es sich in dieser Arbeit immer nur um diese Slalom-Art handelt, wenn von statischen Hindernissen die Rede ist. Der Unterschied von statischen und dynamischen Hindernissen ist in Abbildung 2.5 zu sehen.

[7] Details sind in [Wickert (2012)] nachzulesen

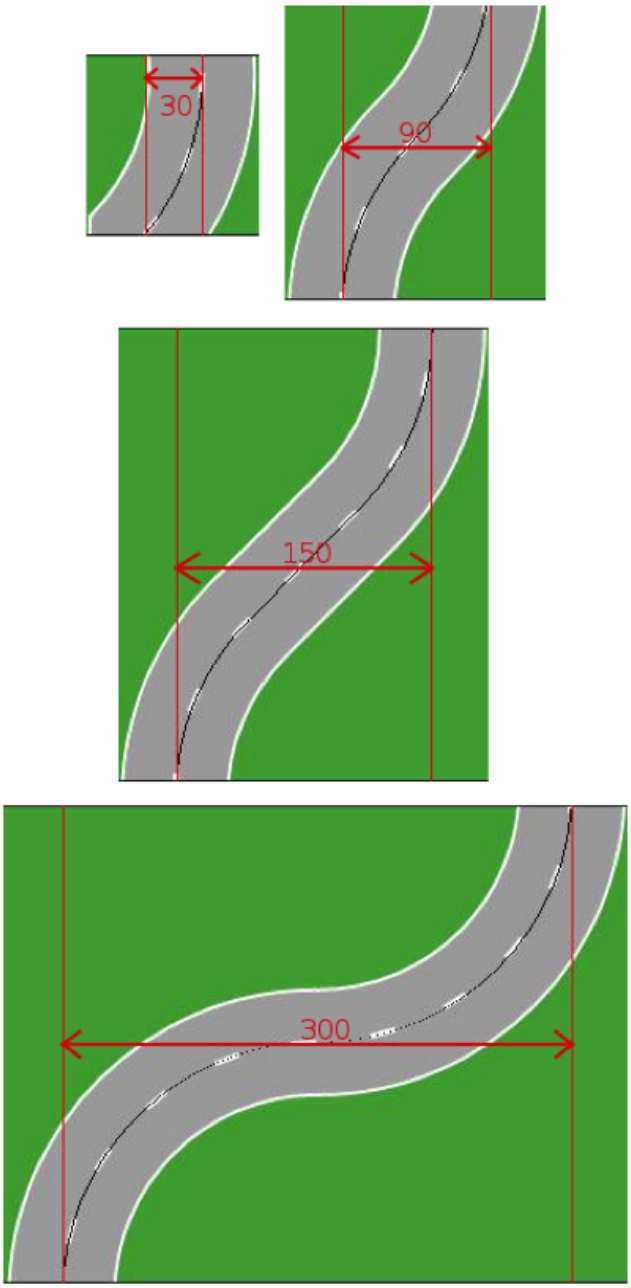


Abbildung 2.3: Verschiedene Kurvenarten: Leichte Kurven sind 30 Pixel (px) breit, mittlere Kurven 90px, schwere Kurven 150px und sehr schwere Kurven 300px — Quelle: [Kosjar (2012), Abb. 6.7]

.

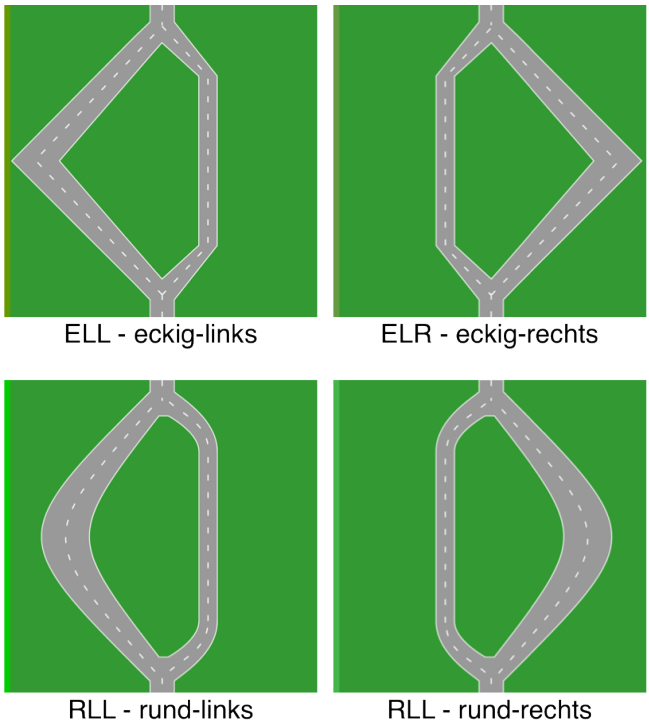


Abbildung 2.4: Verschiedene Gabelungsarten: Eckig mit breitem Arm links bzw. rechts und rund mit breitem Arm links bzw. rechts.

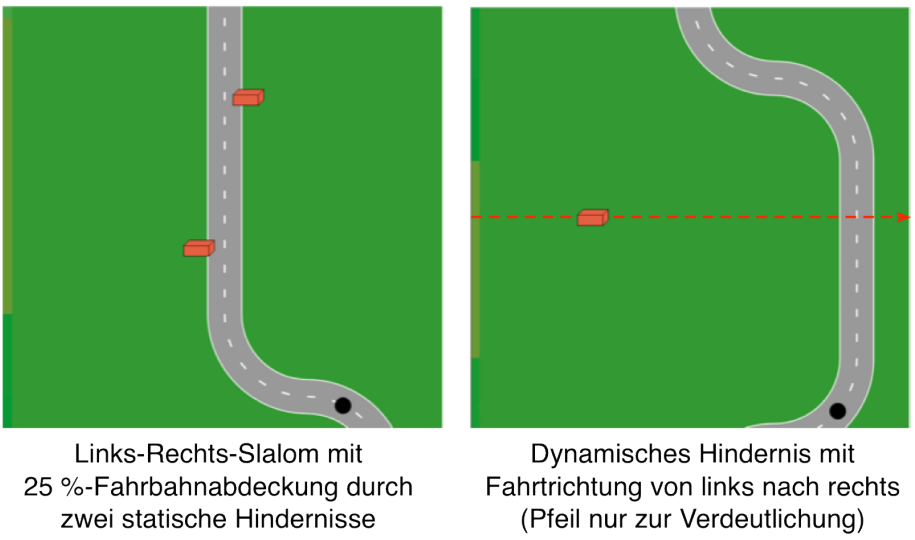


Abbildung 2.5: Hindernisarten in SAM — Quelle: [Kosjar (2012), Abb. 2.4]

2.5 ATEO Automation Framework (AAF) im Detail

Für die Implementierung von Automatikfunktionen (im Zusammenhang mit SAM auch *Agenten* genannt) wurde von [Hasselmann (2013)] ein spezieller Framework entwickelt: Das *ATEO Automation Framework*, kurz AAF. Es gibt den Entwicklern von Automatikfunktionen ein Interface vor, das in einem neuen Agenten umgesetzt werden muss, damit er schließlich korrekt in den Ablauf von SAM eingebunden werden kann. Außerdem stellt das Framework einige Klassen und Methoden bereit, die das Einbinden neuer Agenten erleichtern.

Technisch gesehen ist eine *Automatik* ein gerichteter Graph. Jeder Graph beginnt bei genau einer Wurzel und endet in genau einer Senke. Bis auf jene Wurzel und Senke hat jeder Knoten genau einen Vorgänger und genau einen Nachfolger. Jeder Knoten des Graphen repräsentiert einen Agenten. Das heißt, dass eine Menge von Agenten eine Automatik bilden.

Der Datenfluss innerhalb des Graphen erfolgt über den Erhalt und die Weitergabe eines speziellen Datenobjekts, dem sogenannten SAMState, durch die einzelnen Agenten. Der SAMState enthält einen Teil der Daten, die zu Beginn eines Ticks (was ein Tick ist, wird in Unterabschnitt 2.4.1 erläutert) im Hauptdatenobjekt von SAM, dem SAMModelData, gespeichert sind. Somit erhält jeder Agent, der unmittelbar mit der Wurzel verbunden ist, eine Kopie des Original-SAMStates. Zuletzt werden alle SAMStates in der Senke zusammengeführt. Dieser Vorgang benötigt aber noch weitere Entwicklungsarbeit, da es unter Umständen zu erheblichen Konflikten kommen kann, wenn von unterschiedlichen Automatikfunktionen die selben Einträge in einem SAMState parallel bearbeitet wurden. Die derzeitige Lösung der Verschmelzung einzelner SAMStates zu einem neuen ist im grunde nicht einsetzbar. Es wird daher dringend empfohlen, auf die parallele Anordnung von Automatikfunktionen im Graphen zu verzichten, um den Verschmelzungsprozess zu vermeiden. Selbst wenn Agenten unterschiedliche Attribute des SAMStates verändern, werden die geänderten Daten beim Merging verfälscht. Stattdessen sollten alle Automatikfunktionen in einem einzigen Pfad seriell angeordnet werden. Somit ist sichergestellt, dass die Nachfolger die Daten des Vorgängers zwar verändern, aber keine Seiteneffekte durch unkorrektes Vereinigen mehrerer SAMStates mehr auftreten können. Die Senke schließlich gibt den durch die Automatikfunktionen veränderten SAMState an den SAM-Prozess zurück. Dort werden die Daten des SAMStates in das SAMModelData-Objekt kopiert und für die weitere Verarbeitung bereitgestellt. Auf diese Weise übt eine Automatik ihren Einfluss auf den Trackingprozess aus.

Da jeder Automatik-Graph genau eine Wurzel und eine Senke hat, können als Knoten in einer Automatik nicht nur (atomare) Agenten eingefügt werden; auch bereits existierende Automatiken können dem Graphen als Knoten hinzugefügt werden.

Für eine unkomplizierte Erstellung von Automatiken wurde von [Fuhrmann (2010)] ein GUI konzipiert und umgesetzt, mit dessen Weiterentwicklung sich [Kosjar (2011)] befasst hat.

Alle weiteren Informationen erhält jeder Interessierte aus der Bedienungsanlei-

tung im Anhang C in [Kosjar (2012)].

2.6 IST-Zustand

Vor und während der Entwicklungsarbeiten wurde eine Bestandsübersicht erstellt, die aufzeigt, welche Software-Komponenten existieren und welche für die Zwecke dieser Arbeit wiederverwendet werden können.

Die Basis für die Umsetzung der einzelnen Automatikfunktionen bilden der AAF und das **AAF G***raph* **T***ool* (AAFGT). Das AAF wurde von Michael Hasselmann im Rahmen seiner Diplomarbeit [Hasselmann (2013)] entwickelt; das AAFGT wurde von Esther Fuhrmanns Diplomarbeit [Fuhrmann (2010)] entwickelt und von Nikolai Kosjar in dessen Studienarbeit [Kosjar (2012)] weiterentwickelt.

Zudem werden vorhandene Funktionen von SAM extrahiert und auf spezielle Bedürfnisse angepasst wiederverwendet (zum Beispiel die Eingabewert-Berechnung aus Joystick-Rohdaten).

Im Zuge der voranschreitenden Entwicklung wurden von Nikolai Kosjar das *Ereignis- und Test-Framework* und von Aydan Seid eine Funktion zur Geschwindigkeitsermittlung realisiert. Diese finden in Teilen dieser Arbeit Anwendung; die jeweiligen Stellen wurden entsprechend mit Quellverweisen markiert.

Kapitel 3

Analyse der Entwicklerkonzepte

Der erste Abschnitt dieses Kapitels wird die Hintergründe für die Motivation der Diplomarbeit aufzeigen. Dazu wird kurz auf Kains Experiment (beschrieben in [Kain]) eingegangen, in dessen Verlauf die Entwicklerkonzepte entstanden sind.

In den weiteren Abschnitten wird die Analyse der Konzepte aus Sicht der Informatik präsentiert. Es werden Attribute eingeführt, die zur Klassifikation von Automatikfunktionen dienen und schließlich eine Übersicht dargeboten, welche Konzepte Eingriffs-Automatikfunktionen vorsehen und bei welchen Ereignissen diese jeweils zum Einsatz kommen. Zuletzt werden Ungenauigkeiten sowie fehlende oder widersprüchliche Informationen in den Entwicklerkonzepten beispielhaft erläutert.

3.1 Das Experiment von Saskia Kain

Als Grundlage für alle in dieser Arbeit entwickelten Automatikfunktionen dient die Sammlung an Entwicklerkonzepten[1], die im Rahmen von Saskia Kains Dissertation (in Vorbereitung) entstand. Wie genau diese Konzepte entstanden sind wird in [Kain] nachzulesen sein. Für das Verständnis der Analysen in dieser Arbeit genügen die im Folgenden beschriebenen Informationen.

Jedes der 30 Konzepte wurde von je einem Team bestehend aus 3 Personen entwickelt. Alle an der Entwicklung beteiligten Personen haben Erfahrung mit Automatisierungs-, Assistenz- oder sonstigen komplexen, dynamischen Systemen. Unter den Teilnehmern sind namhafte Unternehmen wie Airbus, BMW, Diehl, DLR Braunschweig, EADS, Honda und NEC zu nennen.

Der Ablauf des Entwicklungsprozesses selbst war bei allen Teams ähnlich. In der ersten Phase wurde SAM vorgestellt, die Aufgabe[2] erläutert, ein Video eines

[1] Nummeriert von 20 bis 49, damit kein Team das Gefühl bekommen sollte, es sei das erste.

[2] Konzeption von Automatikfunktionen zur Unterstützung beziehungsweise Führung der Mikroweltbewohner mit dem Ziel, den menschlichen Operateur zu ersetzen

Trackings mit SAM und dem Operateur gezeigt und Informationsmaterial über SAM und seine Komponenten ([Bothe u. a. (2009)]) zur Anschauung gereicht. Dabei wurde die Art und Menge der zur Verfügung gestellten Informationen in drei Stufen variiert, sodass nicht alle Teams die selbe Informationsgrundlage besaßen. Jeweils zehn Teams wurden den drei Stufen zugewiesen.

1. Einführungsvortrag, Systembeschreibung von SAM.
2. wie Stufe 1; zusätzlich wurde ein Video dargeboten, in dem die Objektsteuerungsaufgabe gezeigt wurde, ein erfahrener Mikroweltbewohner konnte interviewt und die Objektsteuerungsaufgabe von den Entwicklern selbst durchgeführt werden.
3. wie Stufe 2; zusätzlich erhielten die Entwickler grobe Einblicke in einen exemplarischen OA, konnten selbst als Operateur aggieren und durften einen erfahrenen Operateur interviewen. Außerdem erhielten sie das Ergebnis einer hierarchischen Aufgabenanalyse als Informationsquelle.

Im Einführungsvortrag wurden den Entwicklerteams strikte Einschränkungen vorgegeben, die es unbedingt einzuhalten galt (Arten wie auf das System eingewirkt werden kann, bestimmte Informationsverbote, EinflussnahmeRichtlinien, etc.). Nach dieser Vorbereitung begann dann die Entwicklungsphase, für die die Entwickler nur zwei Stunden zur Verfügung hatten. In dieser Zeit produzierte jedes Team eine Fülle von Funktionen, Automatiken und Skizzen, die in Form von Konzeptformularen aus Papier gespeichert und anschließend von Kain u. a. digitalisiert wurden.

3.2 Abgrenzung

Zu Beginn der ATEO Projektphase 3.0 erfolgte eine Aufteilung, durch die jedem Beteiligten des Projekts die Verantwortung für ein Sach- beziehungsweise Teilgebiet übertragen wurde. Es wurde diskutiert, die Aufteilung nach Entwicklerteams vorzunehmen. Der Vorteil dieser Aufteilung hätte darin bestanden, dass die Bearbeitung eng miteinander verbundener Automatikfunktionensprich kompletter Entwicklerkonzepte- aus einer Hand erfolgt wäre. Stattdessen wurde entschieden, folgende Rubriken (und deren Zuweisungen zu Studenten) zu bilden:

- Darstellung und Anzeige der aktuellen und optimalen Joystick- sowie Objektposition, -richtung und -geschwindigkeit: [Seid (2012a)].
- Darstellung und Anzeige von visuellen sowie Wiedergabe von auditiven Hinweisen vor speziellen Kurven, Gabelungen und Hindernissen inklusive der Erkennung bestimmter Ereignisse (zum Beispiel ob ein Hindernis speziellen Typs voraus liegt oder sich eine Gabelung nähert): [Kosjar (2012)].
- Optimaler Rückweg zur Strecke beziehungsweise Berechnung der Streckenmittellinie (genannt *RacingLine*): [Weidner-Kim].
- Eingriffe in die Trackingobjekt Steuerung: Andreas Wickert, Gegenstand der vorliegenden Arbeit.

Da sich die vorliegende Arbeit ausschließlich mit Automatikfunktionen befasst, die sich direkt auf die Steuerung des Trackingobjekts auswirken, konzentriert sich der Verfasser bei der Analyse auch nur auf solche Automatikfunktionen mit dieser Eigenschaft.

3.2.1 Grobklassifikation von Automatikfunktionen

Im Zuge der Entwicklung des in Abschnitt 2.5 vorgestellten AAF wurden speziell für den OA beziehungsweise das AMD (siehe Abbildungen 1.1 und 1.3) verschiedene elementare Automatikfunktionen realisiert. Löst der Operateur eine entsprechende Funktion aus, bewirken sie beispielsweise *Veränderungen der Eingabeverteilung* oder die *Anzeige beziehungsweise das Abspielen von Hinweisen* während der Versuchsdurchführung.

Aufgrund der unterschiedlichen Wirkungsweisen werden Automatikfunktionen zwei Klassen zugeordnet: Die Klasse der *harten* beziehungsweise *weichen* Automatikfunktionen (siehe folgende Unterabschnitte).

Die ersten Arbeiten und Implementierungen von Automatiken sind in Kai Kesselrings Diplomarbeit [Kesselring (2009)] beschrieben und dokumentiert.

Harte Automatikfunktionen

Automatikfunktionen dieser Klasse bewirken Eingriffe, die das Trackingobjekt direkt beeinflussen. Zum Beispiel kann eine harte Automatikfunktion verhindern, dass das Trackingobjekt sich in eine bestimmte Richtung bewegen kann oder eine maximale Geschwindigkeit nicht überschreiten darf.

Weiche Automatikfunktionen

Automatikfunktionen dieser Klasse haben keinerlei direkten Einfluss auf das Trackingobjekt. Vielmehr wirken sie auf die Mikroweltbewohner. Durch akustische und / oder visuelle Hinweise sollen sie die Verhaltensweise der Mikroweltbewohner beeinflussen. Ein akustischer Hinweis kann ein Warnton in einer bestimmten Situation oder eine Richtungsempfehlung in Form einer gesprochenen Anweisung sein. Im Fall einer visuellen Unterstützung wird zum Beispiel eine Pfeilgrafik eingeblendet, die die Richtung, in die gesteuert werden soll, anzeigt.

Zu dieser Klasse gehören aber nicht nur alle Hinweisarten. Auch die Veränderung der Einflussverteilung für die Mikroweltbewohner ist eine weiche Automatikfunktion, da sie keinerlei Einfluss auf das Trackingobjekt direkt ausübt, sondern den Mikroweltbewohner unterschiedlich starken Einfluss auf die Steuerung zuteilt.

3.3 Gründe für Eingriffe

Die meisten Teams begründen die Eingriffe oder Einflussveränderungen hauptsächlich damit, dass sie Zeit- und Flächenfehler[3] minimieren möchten.

Die meisten Teams erachten das *Verlassen der Fahrbahn* (sei es durch zu schnelles oder durch zu ungenaues Fahren), *Kollisionen mit Hindernissen* und die *Konfliktlösung an Gabelungen* als Hauptfehlerquellen. Dementsprechend wurden Möglichkeiten entwickelt, um das Verlassen der Fahrbahn zu verhindern (wodurch ein Flächenfehler erzeugt würde) oder die Gefahr von Kollisionen mit Hindernissen zu minimieren, da hier als Strafe eine Zwangspause und eine Versetzung des Trackingobjekts neben die Strecke folgen würde.

Einige Teams sprechen sich auch für eine Geschwindigkeitsanpassung aus, um die gesamte Fahrdauer zu optimieren. Hier soll die Geschwindigkeit erhöht werden, wenn die gegenwärtige Sitiuation dies erlaubt.

Zudem gibt es Konzepte, die zwar das Verlassen der Fahrbahn erlauben, aber bei extremem „Off-Road-Fahren“ eine automatische Streckenrückführung vorsehen - auch hier mit dem Ziel, den entstehenden Flächenfehler zu verringern und auf optimalem Weg zurück zur Idealline zu gelangen.

Durch adaptives Lenken soll in einigen Konzepten das Verlassen der Fahrbahn bei hohem Tempo vermieden und Übersteuern verhindert werden. Zudem sollen diese Funktionen dazu beitragen, dass das Trackingobjekt einfacher auf der Ideallinie gehalten werden kann.

Einflussänderungen sollen in gewissen Situationen die gleichen Ziele verfolgen. Ein höherer Einfluss für den „schnelleren“ Mikroweltbewohner auf Geraden kann den Zeitverlust und in Kurven für den „genaueren“ Mikroweltbewohner den Flächenfehler verringern.

3.4 Konzeptanalyse aus IT-Sicht

Im Zuge der Sichtung der Entwicklerkonzepte wurde die ATEO-Arbeitsgruppe in zwei Teams aufgeteilt: Dem einen Team gehörten Nikolai Kosjar, Nicolas Niestroj und Helmut Weidner-Kim und dem anderen Aydan Seid und Andreas Wickert (dem Autor der vorliegenden Arbeit) an. Die Aufgabe bestand darin, die Entwicklerkonzepte aus Sicht der Informatik zu analysieren. Damit die Ergebnisse vergleichbar waren, fand die Analyse auf Grundlage eines gemeinsamen Fragenkatalogs statt. Jede Automatikfunktion wurde auf zwölf Eigenschaften[4] hin untersucht und bewertet:

A *Leichtigkeit der Algorithmusfindung:* Das Finden eines Algorithmus für diese Funktion ist leicht.

B *Leichtigkeit der Implementierung:* Die Implementierung dieser Funktion mittels Squeak unabhängig vom AAF, GUI und SAM ist leicht.

[3] Fehlerarten in SAM werden in [Weidner-Kim] detailliert behandelt.

[4] Antwortskala je Eigenschaft: 1 = gar nicht; 2 = wenig; 3 = mittelmäßig; 4 = ziemlich; 5 = völlig.

C *Zeitlicher Aufwand „Algorithmusfindung“:* Die Algorithmusfindung für diese Funktion unabhängig von allen anderen Funktionen ist zeitlich aufwändig.

D *Zeitlicher Aufwand „Implementierung“:* Die Implementierung (Quellcode, Testung) für die Funktion unabhängig von allen anderen Funktionen ist zeitlich aufwändig.

E *Funktionsumfang:* Optionale Erweiterungen des Funktionsumfangs (beispielsweise zusätzliche Parametrisierung), welche nicht direkt im Konzept definiert sind, sind möglich.

F *Systemintegration AAF:* Die Integration (beispielsweise Codeanpassung, Parametrisierung) der Funktion in das AAF und dessen GUI zur Konfiguration ist aufwändig.

G *Wiederverwendbarkeit für andere Funktionen des Konzepts:* Diese Funktion ist für andere Funktionen des Konzepts wiederverwendbar (unter Berücksichtigung optionaler Parametrisierung).

H *Wiederverwendbarkeit von vorhandenen Funktionen:* Bereits implementierte Funktionen im AAF, dem OA oder SAM sind für diese Funktion verwendbar.

I *Wiederverwendbarkeit für andere Konzepte:* Diese Funktion ist für Funktionen anderer Konzepte wiederverwendbar (unter Berücksichtigung optionaler Parametrisierung).

J *Störung:* Diese Funktion interferiert mit anderen Funktionen des Konzepts, was bei der Implementierung berücksichtigt werden muss.

K *Detailliertheit der Funktionsbeschreibung:* Die Funktion ist ausreichend detailliert für die Implementierung beschrieben.

L *Implementierungsstatus (vom jeweiligen Bearbeiter zu bewerten):* Diese Funktion ist implementiert.

Anhand der Bewertung der Konzepte konnten viele Komponenten identifiziert werden, die für mehrere Automatikfunktionen wiederverwendet werden konnten. Zum Beispiel wird die Berechnung der aktuellen beziehungsweise nächsten Trackingobjekt-Position in vielen Automatikfunktionen verwendet. Zudem werden Funktionen wie die Roh-Joystick-Daten- oder Geometrie-Berechnung einige Male eingesetzt (beschrieben in Abschnitt 4.5). Ebenso kommt die Benutzung von *Toleranzkorridoren* in mehreren (auch nicht eingreifenden) Automatikfunktionen zum Einsatz (Details dazu in den Abschnitten 4.1 und 4.2). Auch das *Subpixelverhalten* von Squeak spielt häufig eine besonders zu berücksichtigende Rolle (beschrieben in Unterabschnitt 4.4).

Außerdem wurden bei fast allen Konzepten teilweise erhebliche Mängel entdeckt. Deren Diskussion innerhalb der ATEO-Arbeitsgruppe führte zur Beseitigung einiger Konzept-Lücken und zum Ausschluss weiterer Konzepte aus diversen Gründen:
Alle Konzepte, deren Arbeitsgrundlage die *Qualität, Genauigkeit* oder *Güte* der individuellen Fahrleistungen der Mikroweltbewohner bildete, mussten abgelehnt werden, da keine geeignete Methode zur Bestimmung der Fahrgüte gefunden werden konnte. Der Grund dafür ist das nichtdeterministische Verhalten der

Mikroweltbewohner, wodurch die individuellen Fahrleistungen unmöglich bestimmbar sind. Wenn hingegen die gemeinsame Fahrleistung als Bewertungsgrundlage verwendet wird, so kann darüber eine Aussage getätigt werden, wie gut beide Mikroweltbewohner zusammen aggieren. Denn für jede Strecke kann ein Bestwert berechnet werden, an dem die Leistung der Mikroweltbewohner gemessen werden kann: Die Zeit, die das Trackingobjekt benötigt, um die Racingline mit optimaler Geschwindigkeit entlang zu fahren.
Ein weiterer Grund für die Ablehnung einiger Konzepte war die Verwendung von Sensoren zur Orientierung in der unmittelbaren Umgebung des Trackingobjekts. Alle entwickelten Sensorenanordnungen sind in Abbildung 3.1 dargestellt. Die linke Variante ist die ursprüngliche Version, die auch in SAM Anwendung findet. In der mittleren Version wurden die Standard-Sensoren auf die Positionen projeziert, die sie mit maximalen Steuersignalen erreichen können. Die rechte Version zeigt ein Array von Sensoren, welche eine lückenlose Abtastung in dem Bereich garantiert, den das Trackingobjekt innerhalb eines Ticks erreichen kann.

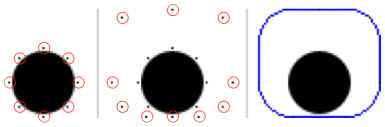


Abbildung 3.1: Drei verschiedene Ansätze für Sensoren am Trackingobjekt.

Wie sich bei der Entwicklung der Konzepte zeigte, sind sensorenbasierte Methoden nicht sehr exakt, wodurch Widersprüche in den Konzepten in Erscheinung traten. So kann nicht gewährleistet werden, dass das Trackingobjekt ausschließlich durch die Verwendung von Sensoren auf der Fahrbahn gehalten wird, da in gewissen Situationen die Fahrbahn zwischen den Sensoren hindurch ragen kann und sich das Trackingobjekt somit über den Fahrbahnrand hinaus bewegt. Beispielhafte Fälle werden in Abbildung 3.2 aufgezeigt. In allen Situationen schlägt kein Sensor Alarm, aber dennoch befindet sich das Trackingobjekt bereits nicht mehr vollständig auf der Fahrbahn beziehungsweise es besteht die Gefahr, dass das Trackingobjekt die Fahrbahn innerhalb des nächsten Ticks verlässt.

Selbst wenn die dritte Variante für den Einsatz herangezogen wird, kann nur das Verlassen der Fahrbahn *detektiert* werden. Die optimale Position, in der sich das Trackingobjekt noch gerade so auf der Fahrbahn befinden würde, kann also mit Sensoren nicht – beziehungsweise nur sehr rechenintensiv – bestimmt werden.

Weiterhin wurden Konzepte mit ungenügend detaillierter Beschreibung abgelehnt. Beispielsweise hat Team 26 eine Automatikfunktion vorgesehen, die dann eingreift, wenn eine Kollision mit einem statischen Hindernis möglich ist und so lange aktiv bleibt, bis eine Kollision verhindert wurde. Aus diesen Informationen lässt sich keine eindeutige Automatikfunktion ableiten. Im Grunde ist eine Vielzahl an Automatikfunktionen denkbar, die sehr unterschiedlich sein können, aber alle eine Kollision mit statischen Hindernissen verhindern. Es ist aber unmöglich zu sagen, welche davon genau der im Konzept vorgesehenen Automatikfunktion entspricht.

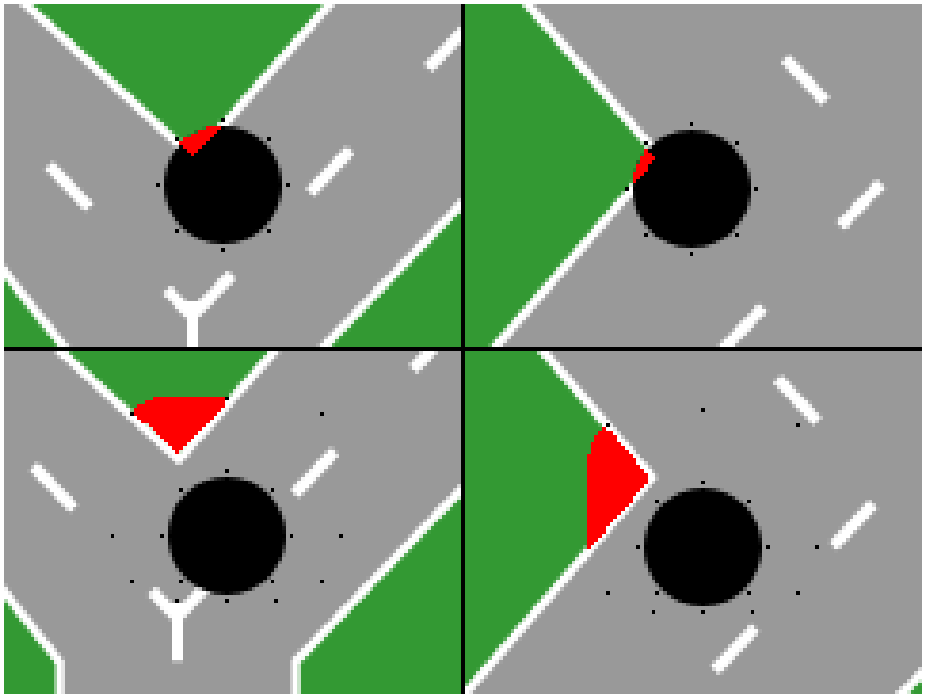


Abbildung 3.2: Situationen, in denen Sensoren nicht verhindern, dass das Trackingobjekt auf der Fahrbahn bleibt.

3.5 Attribute der Entwicklerkonzepte

Im Zuge der Analyse der speziell im Fokus dieser Arbeit stehenden Automatikfunktionen sind einige Merkmale (im Folgenden *Attribute* genannt) identifiziert worden, mit deren Hilfe sie klassifiziert werden können. Als Darstellungsform für die identifizierten Attribute wurde eine tabellarisch-hierarchische Form gewählt. Ein Versuch, die Attribut-Struktur mit dem Werkzeug CTE XL zu erfassen, ist gescheitert, da eine Mehrfachauswahl einzelner Attribute unterhalb einer Klasse damit nicht möglich ist. Mit CTE XL konnte beispielsweise die Darstellung eines Konzepts, das alle möglichen Hindernisarten berücksichtigt, nicht umgesetzt werden.

In Abbildung 3.3 sind viele der identifizierten Attribute dargestellt. Um Attribute zu referenzieren wird in dieser Arbeit die Punkt-Notation verwendet. Um beispielsweise das Attribut für 25-Prozent-Hindernisse zu referenzieren, verwendet man:

$$AF.Ereignis.Hindernis.statisch.25$$

Auch eine Sammel-Darstellung ist damit möglich. Das folgende Beispiel umfasst alle Hindernis-Attribute:

$$AF.Ereignis.Hindernis.*$$

Dabei erhebt die abgebildete Attribut-Übersicht keinen Anspruch auf Vollständigkeit. Über *Hinweis-Attribute*[5] sowie *Ereignis-Attribute*[6] – wie zum Beispiel *Hindernis* und *Fahrbahnverlauf* – werden an anderer Stelle detailliertere Informationen gegeben. Die in der Abbildung 3.3 aufgeführten Attribute sind zum großen Teil relevant für die vorliegende Arbeit. Andere – speziell die *Hinweis-Attribute* – sind mit aufgenommen, um zu verdeutlichen, dass neben den Attributen für Eingriffsautomatikfunktionen auch noch weitere existieren. Auch die Granularität der aufgeführten Attribute ist nicht bis ins Kleinste ausgereizt. Zum Beispiel ist das Attribut

$$AF.Ereignis.Steuerungsverhalten.Genauigkeit.Lenkung$$

nicht vollständig aufgeschlüsselt. Es wurde nur angedeutet, was mit *Lenkung* in Bezug auf das Fahrverhalten gemeint ist. Zudem hätten auch unter anderem die Gabelungsarten noch in *links* beziehungsweise *rechts* weiter unterteilt werden können. Da aber in allen Konzepten dazu keine explizite Unterscheidung getroffen wurde, ist die Granularität für eine Klassifizierung ausreichend.

Die Attribute mit Relevanz für die vorliegende Arbeit wurden in der Grafik eingefärbt. Im Folgenden werden alle Attribute, die neu hinzu kommen und in keiner anderen Arbeit erwähnt werden, detailliert erläutert.

[5] Arbeiten von [Kosjar (2012)] und [Seid (2012a)]
[6] Arbeit von [Kosjar (2012)]

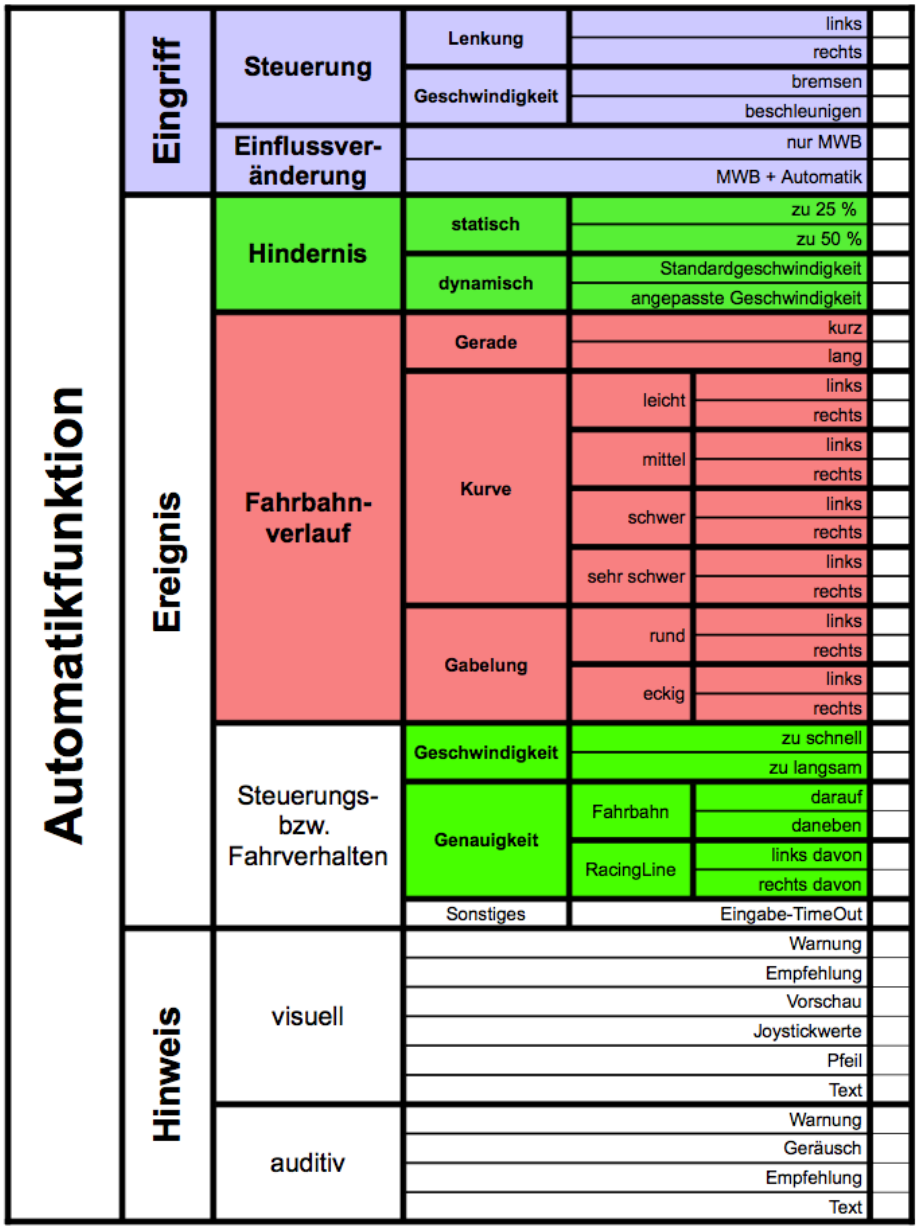


Abbildung 3.3: Attribute von Automatikfunktionen- farbig dargestellte Attribute sind für die vorliegende Arbeit von besonderem Interesse.

3.5.1 AF .Eingriff .Steuerung .*

Steuerungseingriffe sind von allen Eingriffsattributarten die komplexesten, da sie auf alle Dimensionen der Steuerung Einfluss nehmen können. Das heißt sowohl auf alle Lenkrichtungen, als auch auf die Geschwindigkeit des Trackingobjekts kann eine Automatikfunktion einwirken. Sinnvoll ist so eine Eingriffsvielfalt, wenn signifikant *schwierige* Stellen zu passieren sind. Als bestes Beispiel kommt das dynamische Hindernis in Frage. Unter Umständen ist ein dynamisches Hindernis durch einen Lenk- und Beschleunigungseingriff zu passieren, bevor es die Strecke gequert hat. Erfolgt dieser Eingriff aber nur eingeschränkt[7] und die Mikroweltbewohner bremsen so sehr, dass die Gefahr einer Kollision wieder akut wird, so hat die Automatikfunktion auch die Möglichkeit, beim Bremsen unterstützend zu helfen, um das Hindernis erst passieren zu lassen und dann die Fahrt fortzusetzen.

AF .Eingriff .Steuerung .Lenkung .*

Lenkeingriffe beeinflussen teilweise oder vollständig die Horizontalsteuerung des Trackingobjekts. Sie haben dabei keinerlei Auswirkungen auf die Geschwindigkeit des Trackingobjekts. Dennoch bleiben bei dieser Eingriffsart die Möglichkeiten bestehen, dass die Mikroweltbewohner sowohl Zeitfehler als auch Flächenfehler verursachen, indem sie nicht mit maximal möglicher Geschwindigkeit fahren beziehungsweise zu schnell durch Kurven fahren, in denen eine zu hohe Geschwindigkeit zum Verlassen der Ideallinie führt. Lenkeingriffe sollen dabei den Flächenfehler so gering wie möglich halten.

AF .Eingriff .Steuerung .Geschwindigkeit .*

Geschwindigkeitseingriffe wirken je nach Bedarf *beschleunigend* oder *bremsend* auf das Trackingobjekt. Ein mögliches Einsatzgebiet für diese Eingriffsart ist wieder der Bereich um dynamische Hindernisse. Der Vorteil gegenüber solchen Automatikfunktionen, die entweder nur bremsen oder nur beschleunigen können, ist der, dass sie bei dynamischen Hindernissen in der Lage sind, flexibler zu reagieren. Ist eine Passage des Hindernisses möglich, bevor es die Strecke quert, so kann das Trackingobjekt beschleunigen. Anderenfalls besteht für die Automatikfunktion die Möglichkeit, dem Hindernis durch Abbremsen den Vorrang zu lassen und dadurch eine Kollision zu vermeiden.

AF .Eingriff .Steuerung .Geschwindigkeit .Bremsen

Diese Eingriffsart ist ausschließlich zur Verringerung der Vorwärtsbewegung des Trackingobjekts bestimmt. Zum einen sollen Bremseingriffe das Verlassen der Strecke in scharfen Kurven und somit eine eine Vergrößerung des Flächenfehlers vermeiden. Zum anderen verhindert man durch Bremseingriffe Kollisionen mit Hindernissen, wodurch eine Zeitstrafe vermieden und der Zeitfehler nicht zusätzlich in die Höhe getrieben wird.

[7] Die Automatik hat nicht zu 100 % Einfluss

AF .Eingriff .Steuerung .Geschwindigkeit .Beschleunigen

Diese Form des Eingriffs wird an Stellen eingesetzt, wo die Mikroweltbewohner das Trackingobjekt schneller voran bewegen könnten, als sie es tatsächlich tun – hauptsächlich kommen hier gerade Fahrbahnstücke der Strecke in Frage. Dies soll die Fahrzeit verringern, also den Zeitfehler verkleinern.

Eine andere Einsatzstelle ist dort, wo dynamische Hindernisse auftreten. Wurde der Punkt für die Geschwindigkeitsmessung des dynamischen Hindernisses[8] so langsam durchfahren, dass es möglich ist, das Hindernis mit höherer Geschwindigkeit noch vor dessen Kreuzen der Strecke zu passieren, so wird ein Beschleunigungseingriff vorgenommen. Dadurch wird die möglicherweise bevorstehende Kollision und eine damit verbundene Zeitstrafe vermieden und die Fahrzeit weiter verringert.

3.5.2 AF .Eingriff .Einflussänderung .*

Eine Einflussänderung bewirkt, dass das Einflussverhältnis zwischen den Mikroweltbewohnern auf die Steuerung des Trackingobjekts entweder *situativ*, *permanent* (bzw. bis zur nächsten Einflussänderung) oder *zeitlich begrenzt* verändert wird. Voreingestellt ist ein Einflussverhältnis von 50 % zu 50 %. Wird der Wert eines Mikroweltbewohners geändert, so ändert sich auch im gleichen Maße der Einfluss des anderen, sodass insgesamt immer eine Summe von 100 % bestehen bleibt. Findet zum Beispiel eine Verringerung des Einflusses für den ersten Mikroweltbewohner von 50 % auf 30 % statt, so steigt der Einfluss des zweiten Mikroweltbewohners auf 70 %.

Ein *situationsabhängiges* Ereignis tritt beispielsweise dann ein, wenn das Trackingobjekt eine Gabelung erreicht oder sich irgendeiner Art von Hindernissen nähert.

Eine *permanente* Änderung kann zum Beispiel dann erfolgen, wenn über einen gewissen Zeitraum keine Steuersignale von einem Mikroweltbewohner registriert worden sind. Dann bekommt der noch steuernde Mikroweltbewohner permanent mehr Einfluss, um die Tracking-Aufgabe alleine zu bewältigen.

Eine *zeitlich begrenzte* Verschiebung des Einflusses kann durch Verschulden eines Mikroweltbewohners ausgelöst werden, indem dieser *grobe Fehler* macht. Unter groben Fehlern versteht man unter anderem:

- unnötiges Langsamfahren
- kollidieren mit einem Hindernis
- sich um mehr als einen bestimmten Wert von der Optimallinie (Racingline) der Strecke entfernen
- die Strecke verlassen
- in einigen Fällen auch, wenn der Streckenrand überschritten wird

[8] Details dazu in [Niestroj (2008)]

Merkmale	AF1	AF2	AF3	AF4
Merkmal A	10	20	∅	30
Merkmal B	∅	∅	40	50
Merkmal C	60	70	∅	∅

Tabelle 3.1: Beispiel für die Klassifizierung von Automatikfunktionen

3.5.3 AF .Ereignis .Fahrverhalten .Genauigkeit .Toleranzkorridor .*

Ein Toleranzkorridor ist eine Zone in konstantem Abstand zur Racingline. Diese Zone ist immer relativ zum Trackingobjekt zu werten. Befindet sich der Mittelpunkt des Trackingobjekts innerhalb oder außerhalb eines Toleranzkorridors, so kann verschieden darauf reagiert werden. Beispielsweise gibt es einen Toleranzkorridor, der den Fahrbahnrand markiert.

3.6 Klassifizierung von Entwicklerkonzepten

Wie bereits im voran gegangenen Abschnitt 3.5 erwähnt, spielen bei der Klassifizierung die Attribute die Hauptrolle. Die Klasse einer Automatikfunktion wird durch die Ausprägung ihrer Attribute bestimmt. Das heißt, alle Automatikfunktionen, die die selben Attribute mit (egal welchen) Werten belegen, gehören der selben Klasse an. Tabelle 3.1 zeigt beispielhaft, auf welche Weise Automatikfunktionen einer Klasse zugeordnet werden: *AF1* und *AF2* gehören zur selben Klasse, da sie *Merkmal A* und *Merkmal C* belegen. *AF3* und *AF4* sind in disjunkten Klassen, da sie unterschiedliche Merkmale belegen. Bei der Klassifizierung spielt der Belegungswert keine Rolle, nur die Tatsache, dass ein bestimmtes Merkmal einen Wert erhält und nicht undefiniert (∅) ist. Das Ergebnis der Beispiel-Klassifizierung ist also:

- Klasse I (Merkmal A, Merkmal C) → { AF1, AF2 }
- Klasse II (Merkmal B) → { AF3 }
- Klasse III (Merkmal A, Merkmal B) → { AF4 }

Bei der Analyse der Entwicklerkonzepte wurden sehr viele Klassen ermittelt, die sich jedoch nur wenig voneinander unterschieden. Aus diesem Grund ist auf eine Auflistung, die *alle* Attribute berücksichtigt, verzichtet worden, da diese zu unübersichtlich geworden wäre. Statt einer Klassifizierung auf niedrigster Attribut-Ebene wurden mehrere Klassen zusammengefasst, sodass die folgende Ergebnispräsentation sehr viel überschaubarer gestaltet werden konnte.

Eine grobe Unterscheidung hätte bereits dadurch getroffen werden können, indem Automatikfunktionen mit Ereignisbezug (zum Beispiel *Kurve* oder *Gabelung*) von solchen ohne entsprechenden Bezug (also ständig aktive Automatikfunktionen) getrennt worden wären. Da aber das Ereignis- beziehungsweise Event-System von [Kosjar (2012)] zur Verfügung steht, ist diese Unterscheidung hinfällig. Dieses System ermöglicht allen Automatikfunktionen, die nur bei speziellen Ereignissen aktiv sein sollen, auch solche Automatikfunktionen wieder

zu verwenden, die normaler Weise permanent arbeiten würden, sofern diese für das Event-System zugänglich sind[9]. Anhand der aus den Entwicklerkonzepten zusammengetragenen Automatikfunktionen wurden folgende Automatikklassen (-kombinationen) abgeleitet, mit denen die meisten Automatikfunktionen (mit Einfluss auf Objektsteuerung und MikroweltbewohnerEinflussverteilung) der Konzepte umgesetzt werden können:

Adaptives Lenken

Je nach Geschwindigkeit wird das maximale Lenkvermögen eingeschränkt.
Attribute:

- AF.Eingriff.Steuerung.Lenkung.*
- AF.Eingriff.Steuerung.Geschwindigkeit.bremsen

(Teams 28 und 47)

Adaptives Lenken (mit Toleranzkorridor-Bezug)

Wie Adaptives Lenken, wird aber erst aktiv, wenn der Toleranzkorridor (in diesem Fall der Fahrbahnrand) berührt wird.
Attribute:

- AF.Eingriff.Steuerung.Lenkung.*
- AF.Eingriff.Steuerung.Geschwindigkeit.bremsen
- AF.Ereignis.Fahrverhalten.Genauigkeit.Toleranzkorridor.außerhalb

(Team 30)

Beschleunigungsautomatik (Gerade)

Eine Automatikfunktion, die das Trackingobjekt auf Geraden *nur* beschleunigt.
Attribute:

- AF.Eingriff.Steuerung.Geschwindigkeit.beschleunigen
- AF.Ereignis.Fahrbahnverlauf.Gerade.*

(Teams 24, 37, 41 und 47)

Beschleunigungsautomatik (dynamisches Hindernis)

Eine Automatikfunktion, die das Trackingobjekt an dynamischen Hindernissen beschleunigt.
Attribute:

- AF.Eingriff.Steuerung.Geschwindigkeit.beschleunigen
- AF.Ereignis.Hindernis.dynamisch.*

(Teams 24, 25, 26, 27, 37, 41 und 47)

Bremsautomatik

[9] ebenfalls beschrieben in [Kosjar (2012)]

Eine Automatikfunktion, die das Trackingobjekt bei bestimmten Fahrbahnverläufen *nur* abbremst.
Attribute:

- AF.Eingriff.Steuerung.Geschwindigkeit.bremsen
- AF.Ereignis.Fahrbahnverlauf.Kurve.* oder
- AF.Ereignis.Fahrbahnverlauf.Gabelung.*

(Teams 24, 30, 37 und 41)

Einflussänderung (Fahrbahnverlauf)

Wenn ein bestimmter Fahrbahnverlauf vorliegt, wird der Einfluss, den die Mikroweltbewohner auf das Trackingobjekt haben, nach den gesetzten Vorgaben geändert.
Attribute:

- AF.Eingriff.Einflussänderung.*
- AF.Ereignis.Fahrbahnverlauf.*

(Teams 36 und 45)

Einflussänderung (Toleranzkorridor-Bezug)

Wenn ein bestimmter Toleranzkorridor erreicht wird, wird der Einfluss, den die Mikroweltbewohner auf das Trackingobjekt haben, entsprechend den Vorgaben geändert.
Attribute:

- AF.Eingriff.Einflussänderung.*
- AF.Ereignis.Fahrverhalten.Genauigkeit.Toleranzkorridor.*

(Teams 36, 39, 42, 43, 45 und 47)

Einflussänderung (fehlerhaftes Verhalten)

Wird bei wenigstens einem Mikroweltbewohner ein Fehlverhalten[10] festgestellt, wird der Einfluss, den die Mikroweltbewohner auf das Trackingobjekt haben, entsprechend den Vorgaben angepasst.
Attribute:

- AF.Eingriff.Einflussänderung.*
- AF.Ereignis.Fahrverhalten.fehlerhaft.* oder
- AF.Ereignis.Genauigkeit.Steuerungsvorgabe ignorieren.*

(Teams 23, 28, 30, 38, 25, 26, 39, 40 und 47)

Einflussänderung (Hindernis statisch)

Wenn ein bestimmtes Ereignis eintritt, wird der Einfluss, den die Mikroweltbewohner auf das Trackingobjekt haben, geändert.
Attribute:

[10] Bewusst so unscharf formuliert, da die betroffenen Konzepte ebenfalls sehr unscharfe Informationen dazu geliefert haben.

- AF.Eingriff.Einflussänderung.*
- AF.Ereignis.Hindernis.statisch.*

(Team 36)

Einflussänderung (mit Racingline-Bezug)

Wenn ein bestimmtes Ereignis mit Racingline-Bezug eintritt, wird der Einfluss, den die Mikroweltbewohner auf das Trackingobjekt haben, geändert.
Attribute:

- AF.Eingriff.Einflussänderung.*
- AF.Ereignis.Fahrverhalten.Genauigkeit.Racingline.*

(Teams 30 und 47)

Einflussänderung (mit Toleranzkorridor-Bezug)

Wenn ein bestimmtes Ereignis mit Toleranzkorridor-Bezug eintritt, wird der Einfluss, den die Mikroweltbewohner auf das Trackingobjekt haben, geändert.
Attribute:

- AF.Eingriff.Einflussänderung.*
- AF.Ereignis.Fahrverhalten.Genauigkeit.Toleranzkorridor.*

(Teams 36, 39, 42, 43 und 45)

Leitplanken

Das Trackingobjekt darf einen bestimmten Toleranzkorridor nicht verlassen.
Attribute:

- AF.Eingriff.Steuerung.*
- AF.Ereignis.Fahrverhalten.Genauigkeit.Toleranzkorridor.*

(Teams 20, 22, 37, 40, 45 und 46)

Streckenrückführung

Erreicht das Trackingobjekt den Rand eines bestimmten Toleranzkorridors, so wird es automatisch in einen anderen Toleranzkorridor (zum Beispiel zurück zur Fahrbahn) oder sogar bis zur Racingline zurück manövriert.
Attribute:

- AF.Eingriff.Steuerung.*
- AF.Ereignis.Fahrverhalten.Genauigkeit.Toleranzkorridor.*
- optional: AF.Ereignis.Fahrverhalten.Genauigkeit.Racingline.*

(Teams 21, 24, 23, 27, 29, 31, 37, 40, 41, 44, 45, 47 und 49)

Hindernisnotbremse (dynamisch)

Unmittelbar bevor eine Kollision mit einem *dynamischen* Hindernis stattfindet, wird automatisch gebremst.
Attribute:

- AF.Eingriff.Steuerung.Geschwindigkeit.bremsen
- AF.Ereignis.Hindernis.dynamisch.*

(Teams 20, 23, 24, 25, 26, 27 29, 31, 36, 40, 41 und 44)

Hindernisnotbremse (statisch)

Unmittelbar bevor eine Kollision mit einem *statischen* Hindernis stattfindet, wird automatisch gebremst.
Attribute:

- AF.Eingriff.Steuerung.Geschwindigkeit.bremsen
- AF.Ereignis.Hindernis.statisch.*

(Teams 20, 23, 24, 29, 31, 36, 37 und 44)

Hindernisumfahrung

Hindernisse (egal ob dynamisch oder statisch) werden automatisch umfahren.
Attribute:

- AF.Eingriff.Steuerung.*
- AF.Ereignis.Hindernis.*

(Teams 28, 31, 33, 35, 39, 45 und 49)

Hindernisumfahrung (statisch)

Speziell statische Hindernisse werden automatisch umfahren.
Attribute:

- AF.Eingriff.Steuerung.*
- AF.Ereignis.Hindernis.statisch.*

(Teams 20, 26, 27, 36, 40 und 42)

Manche Teams treten in ähnlichen Klassen mehrfach auf. Dies liegt daran, dass sie relativ unscharfe Automatikfunktionen konzipiert haben. Wenn sich ein Team generell dafür ausgesprochen hat, dass vor jedweden Hindernissen eine Notbremsung durchzuführen ist, so taucht dieses Team in den Klassen *Hindernisnotbremse statisch* und *Hindernisnotbremse dynamisch* auf. Kommt ein Team in total unterschiedlichen Klassen mehrfach vor, so liegt das in der Regel daran, dass diese Teams mehrere Automatikfunktionen konzipiert haben, die den jeweiligen Klassen zugeordnet werden konnten und disjunkt zueinander sind.

3.7 Ungenaue, fehlende sowie widersprüchliche Informationen der Konzepte

Aufgrund der geringen Entwicklungszeit, die die Teams für ihre Konzepte zur Verfügung hatten, musste sowohl mit Ungenauigkeiten als auch fehlenden Informationen gerechnet werden. Manche Konzepte wiesen sogar Widersprüche auf. Diese zeichneten sich meist dadurch aus, dass unterschiedliche Automatikfunktionen des Konzepts miteinander interferierten.

Um diesen Umständen habhaft zu werden, wurden – wie bereits in Abschnitt 3.4 erwähnt – regelmäßige Treffen der ATEO-Arbeitsgruppe abgehalten. Dabei wurde ein Großteil der Ungenauigkeiten und fehlenden Informationen diskutiert und nach Lösungen gesucht, indem die Original-Konzeptbögen nach eventuell übersehenen Informationen durchsucht und im Extremfall sogar die Videoaufzeichnungen des jeweiligen Teams zu Rate gezogen wurden. Viele Konzepte konnten auf diese Weise sinnvoll komplettiert oder die betroffenen Automatikfunktionen zumindest soweit parametrisiert werden, dass eine Fülle von möglichen Lösungen zur Verfügung stand.

In dieser Arbeit wird aber nicht weiter auf die *einzelnen* Unklarheiten jedes einzelnen Teams eingegangen, da dies den Rahmen sprengen würde. Stattdessen soll eine kleine Übersicht dem Leser ein Gefühl für die Unzulänglichkeiten der Entwicklerkonzepte und die daraus resultierenden Probleme für die Umsetzung geben, die oft im Zusammenhang mit Eingriffsautomatikfunktionen aufgetreten sind.

Die häufigste Ursache für Ungenauigkeiten beziehungsweise ungenügenden Detailgrad ist die *unpräzise Formulierung der jeweiligen Automatikfunktion-Spezifikationen*. So ist häufig nicht klar, auf *welche Art und Weise* ein Eingriff erfolgen soll: Wird er unmittelbar umgesetzt oder durch eine langsame Anpassung (*adaptiv*) vorgenommen? Ebenso fehlen oft *Informationen über Abstände, Grenzen, Toleranzen*, etc. Zum Beispiel wird zwar gefordert, dass in einer bestimmten Situation der Mikroweltbewohner mit der Genauigkeitspriorität mehr Einfluss erhalten soll; es wird aber nicht explizit die Höhe der Einflussänderung festgelegt. Ein weiteres Beispiel macht deutlich, dass die Identifikation fehlender Informationen dazu benutzt werden kann, um Parameter für Automatikfunktionen einzuführen. Viele Konzepte sehen das *Bremsen vor Hindernissen* vor, geben aber keine Auskunft darüber, in welchem Abstand zum Hindernis sich dabei das Trackingobjekt befinden muss oder wie stark gebremst werden soll.

Zudem ist in einigen Fällen die Verwendung von Sensoren als Mittel zur Erkennung des Fahrbahnrandes vorgesehen. Nach eingehender Prüfung kann mit diesem Verfahren aber nicht gewährleistet werden, dass sich das Trackingobjekt zu jedem Zeitpunkt vollständig auf der Fahrbahn befindet (siehe Abschnitt 3.4), dies aber die betroffenen Konzepte (zumindest Teilweise) voraussetzen. Es gibt aber auch noch andere Fälle von Widersprüchen innerhalb von Konzepten. Beispielsweise sehen einige Konzepte eine Hindernisnotbremse in Kombination mit einem Beschleuniger auf geraden Fahrbahnstücken vor. Da Hindernisse ausschließlich auf Geraden auftreten, kommt es hier zum Konflikt, der von den jeweiligen Teams nicht bedacht wurde. An gleicher Stelle tritt ein Konflikt auf, wenn gleichzeitig Hindernisse in bestimmtem Abstand umfahren *und* das Tra-

ckingobjekt auf der Fahrbahn gehalten werden soll. Abbildung 3.4 verdeutlicht diesen Konflikt visuell.

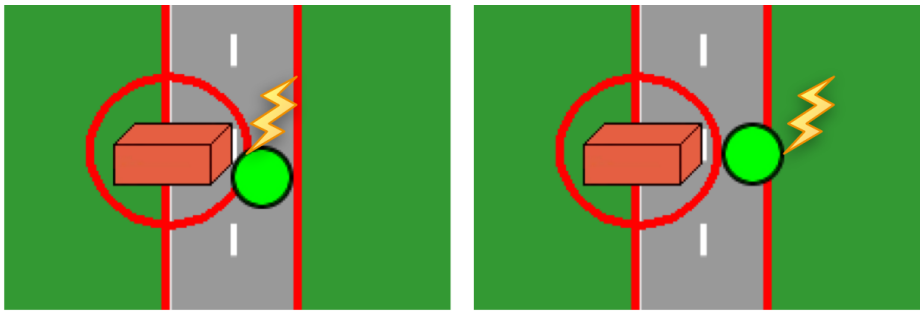


Abbildung 3.4: Wenn Hindernisse in bestimmtem Abstand umfahren werden sollen, dass Trackingobjekt aber die Fahrbahn nicht verlassen darf, kommt es zu einem Konflikt.

Kapitel 4

Theorien und Werkzeuge

In diesem Kapitel werden die Werkzeuge (Tools) – insbesondere zur Analyse der Streckenelemente[1] – und Unterstützungsalgorithmen beschrieben, die während der Entwicklung der Automatikfunktionen benötigt wurden beziehungsweise während ihrer Ausführung – unter Umständen auch von verschiedenen Agenten – verwendet werden.

4.1 Streckenanalyse

Die Entwicklerkonzept-Analyse (siehe Kapitel 3) hat gezeigt, dass die überwiegende Anzahl an Teams eine Art *Toleranzkorridor* für ihre Automatikfunktionen verwendet. Zwar wird dieser Toleranzkorridor unterschiedlich genannt – wie zum Beispiel „*bei berühren des Randes*“[2] oder „*bei Verlassen der Strecke*“[3] – aber sie haben dennoch alle das gleiche Ziel: Wenn das Trackingobjekt eine bestimmte Grenze überschreitet, soll eine Funktion ausgeführt werden. Das gilt im Übrigen auch für alle Automatikfunktionen, die Sensoren als Mittel zur Grenzerkennung verwenden, denn – wie bereits in Abschnitt 3.4 erwähnt – führt diese Methode zu Widersprüchen mit den Konzepten selbst. Zwar hätte man zur bloßen Erkennung von Toleranzkorridor -Verletzungen die dritte Sensoren-Variante (siehe Abbildung 3.1) oder eine ähnliche Methode benutzen können. Jedoch wäre dann die Suche nach einer geeigneten Folgeposition sehr viel rechenintensiver geworden. Für kleine Bereiche hätte es wahrscheinlich noch funktioniert, aber wenn größere Abstände zum Fahrbahnrand bestimmt werden müssten, wäre man mit Sicherheit an die Leistungsgrenzen des Systems gestoßen und die Tick-Dauer von etwa 40ms hätte nicht mehr eingehalten werden können. Gleiches gilt für die folgend beschriebenen Algorithmen: Die ad-hoc-Ermittlung des erlaubten Abstands zum Fahrbahnrand in der jeweiligen Situation und im 40-ms-Zeitrahmen übersteigt die Leistungsfähigkeit von Squeak.

Im Rahmen dieser Arbeit wurde eine flexible Lösung entwickelt, um beliebige Toleranzkorridore außerhalb der Objektsteuerungsaufgabe zu bestimmen und

[1] vgl. Unterabschnitt 2.4.2
[2] Team 20
[3] Teams 22, 27, 28, 36, 37, 39, 40, 41 und 42

dann während des Tracking-Prozesses nutzbar zu machen[4]. In den folgenden Unterabschnitten werden zwei Varianten des entwickelten Streckenanalysewerkzeugs vorgestellt, die dazu benutzt werden können, um beliebige Toleranzkorridore sowohl auf der Fahrbahn (*onTrack*), als auch abseits davon (*offTrack*) zu ermitteln. Beiden Varianten liegt das gleiche Prinzip zu Grunde, das im Folgenden erläutert wird.

Bei der Suche nach einem bestimmten Toleranzkorridor geht es darum, alle Positionen in jeder Pixelzeile zu finden, in welchen das Trackingobjekt den gesuchten Abstand zum Fahrbahnrand hat. Um dieses Ziel zu erreichen, wurde eine Methode entwickelt, die alle vorhandenen Kacheln anhand ihrer Farben analysiert. Da in keinem Konzept explizit angegeben wurde, wie der Abstand zum Fahrbahnrand zu messen ist – ob nur horizontal, nur vertikal oder tangential (vergleiche Abbildung 4.1) – wurde von der ATEO Arbeitsgruppe beschlossen, für alle Konzepte anzunehmen, dass der tangentiale Abstand zum Fahrbahnrand gemeint ist.

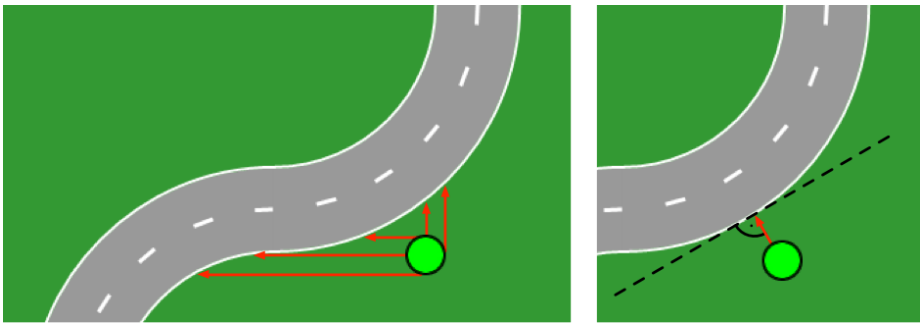


Abbildung 4.1: Drei Möglichkeiten, wie der Abstand zum Fahrbahnrand interpretiert werden kann: Horizontal beziehungsweise vertikal auf der linken Seite; der kürzeste Abstand zu einer Kurventangente auf der rechten Seite.

Zur Vereinfachung der Analyse wird ein *Suchkreis*[5] um den Mittelpunkt des Trackingobjekts gebildet. Der tatsächliche Abstand (also genau der gesuchte Toleranzkorridor) vom Trackingobjekt zum Fahrbahnrand ist dabei gerade

$$Abstand_{Toleranzkorridor} = Radius_{Suchkreis} - Radius_{TrackingObjekt}.$$

Wie dies grafisch aussieht, ist in Abbildung 4.2 erkennbar. Möchte man beispielsweise den Toleranzkorridor bestimmen, sodass das Trackingobjekt gerade so den Fahrbahnrand berührt, so wählt man einen Suchkreisradius der dem des Trackingobjekts entspricht: 15 (Pixel). Es sind aber auch größere und kleinere Radien möglich.

Die Koordinaten des Toleranzkorridors werden ermittelt, indem von jedem Pixel einer jeden Zeile aus der Kreis um das aktuelle Pixel herum nach Fahrbahnfarben abgesucht werden. Der Suchkreis wird zu Beginn des Analyseprozesses erzeugt[6]

[4] Dazu mehr in Abschnitt 4.2.

[5] dargestellt auf Abbildung 4.3

[6] Die Hilfsmethode AAFTrackTools radialCoordinatesForCircleWithRadius: aRadius liefert die Suchkreiskoordinaten für einen bestimmten Radius.

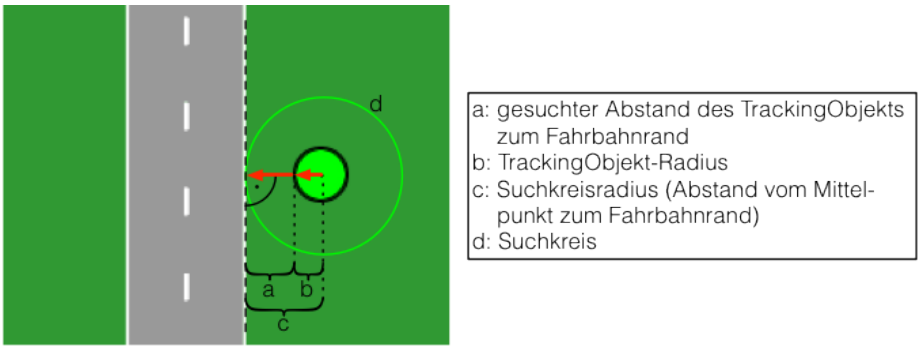


Abbildung 4.2: Die Darstellung von Trackingobjekt, Suchkreis und Streckenausschnitt verdeutlicht das Prinzip, das hinter der Benutzung des Suchkreises zur Ermittlung von Toleranzkorridoren steht.

und in Form von Relativkoordinaten bezüglich dessen Mittelpunktes in einem Array gespeichert. Dies ermöglicht den schnellen Zugriff auf die Umkreis-Pixel einer beliebigen Stelle in einer Grafik.

Während des Analyseprozesses wird nacheinander jedes einzelne Pixel jeder Zeile abgearbeitet. Zu den Koordinaten des jeweils aktuellen Pixels werden die Relativkoordinaten des Suchkreises aus dem Array hinzuaddiert und die jeweils darunter liegende Farbe zwischengespeichert. Dieser Vorgang ist in drei Phasen gegliedert:

- Phase 1: Das blaue Pixel symbolisiert die aktuelle Position, deren Umkreis untersucht werden soll. Der Ring aus schwarzen Pixeln stellt den Suchkreis dar.
- Phase 2: Die Farbwerte unter den Suchkreiskoordinaten werden zwischengespeichert.
- Phase 3: Alle Farbwerte wurden bestimmt und ausgewertet (je nach Variante). Der X-Wert der aktuellen Position wird um eins erhöht – die Umgebung des folgenden Pixels kann nun untersucht werden...

Abbildung 4.4 zeigt die drei Phasen für einen einzelnen Iterationsschritt.

Die so gewonnenen Informationen können auf zwei verschiedene Arten ausgewertet werden:

- *Variante 1* ermittelt den Abstand zum Fahrbahnrand **auf** der Fahrbahn – diese *on-Track-Variante* wird in Unterabschnitt 4.1.1 erläutert.
- *Variante 2* ermittelt den Abstand zum Fahrbahnrand **neben** der Fahrbahn – diese *off-Track-Variante* wird in Unterabschnitt 4.1.2 erläutert.

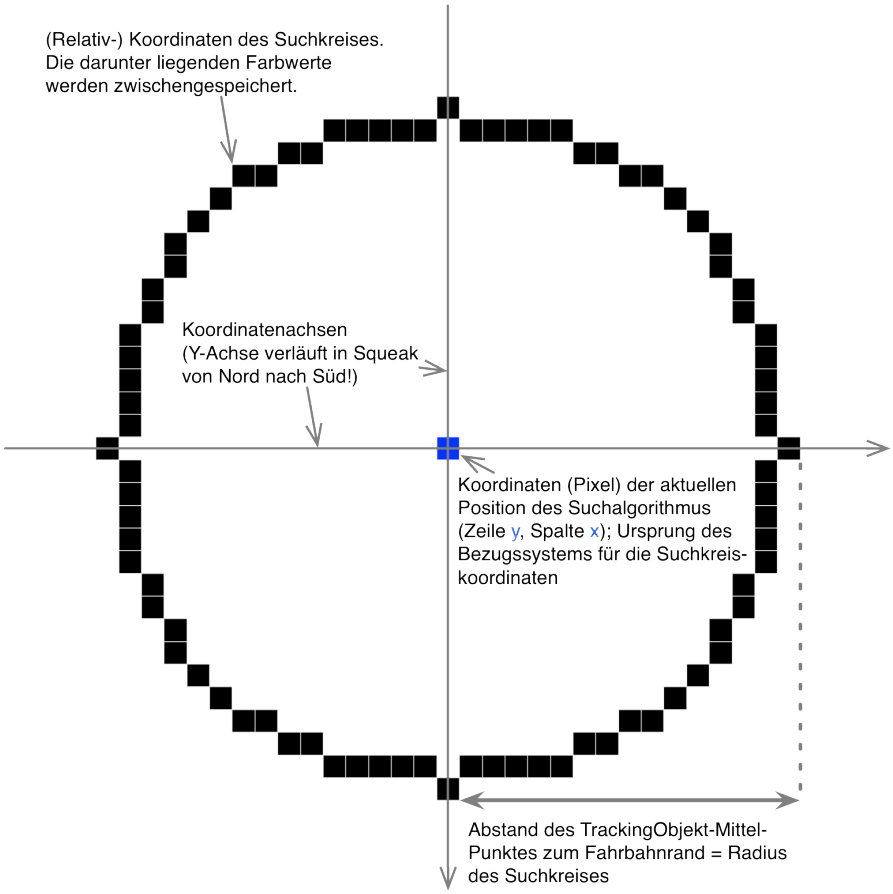


Abbildung 4.3: Suchkreis für die Umgebungsanalyse einer bestimmten Position auf einer Grafik. Für dieses Beispiel wurde ein Radius von 15 Pixeln gewählt, was dem des Trackingobjekts entspricht.

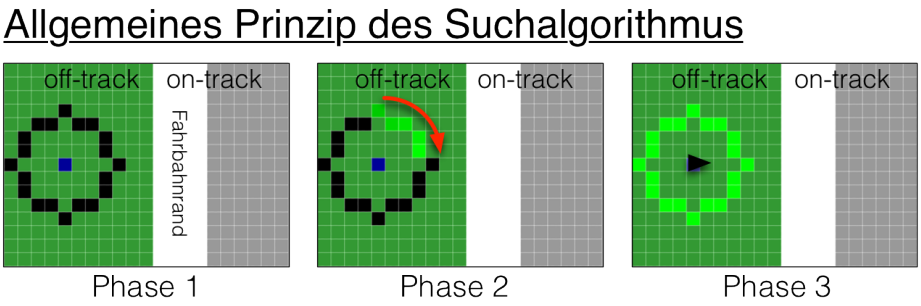


Abbildung 4.4: Das grundlegende Prinzip, wie bei der Toleranzkorridor-Ermittlung der Suchkreis verwendet wird.

4.1.1 Variante 1: Analyse auf der Fahrbahn

Die zuerst entwickelte Variante ermittelt Toleranzkorridore, die sich in einem bestimmten Abstand von den weißen Fahrbahnrändern auf der Fahrbahn befinden. Die Klassenmethode, die dies für eine einzelne Kachel durchführt, ist in der Klasse AAFTrackTools zu finden und hat folgende Signatur:

AAFTrackTools
locateBoundsOnTrack: *aSourceFile*
in: *aSourceFolder*
to: *aTargetFolder*
with: *aRadius*

Die zu analysierende Kachel wird über die Parameter aSourceFile und aSourceFolder lokalisiert. aSourceFile enthält den Kachelnamen ohne Erweiterungen wie zum Beispiel'.bmp' und aSourceFolder einen absoluten oder zum Squeak-Image[7] relativen Pfad zu dem Ordner, in dem sich die Kachel befindet. Der gesuchte Abstand wird über den Parameter aRadius definiert.

Das Resultat der Methode ist eine **C***omma*-**S***eparated* **V***alues* (CSV)-Datei, in der die Koordinaten jedes einzelnen Punktes des Toleranzkorridors gespeichert wurden. Die CSV-Datei wird im Zielpfad – angegeben über den Parameter aTargetFolder – abgelegt.

Der Algorithmus geht zeilen- beziehungsweise pixelweise vor. Die Suche lässt sich in vier Phasen einteilen:

- Phase 1: Der Suchkreis wird solange nach rechts verschoben, bis er die Fahrbahn erreicht. Er wird dann solange weiter nach rechts verschoben, bis...
- Phase 2: ... alle Suchkreiskoordinaten über Fahrbahnfarben liegen. Die aktuellen Koordinaten werden der Ergebnisliste hinzugefügt (erste x-Koordinate).
- Phase 3: Der Suchkreis wird weiter nach rechts verschoben, bis eine Situation entdeckt wird, in der...
- Phase 4: ... entweder unter dem Suchkreis ein grüner Farbwert registriert wird oder der Rand der Kachel erreicht wurde. In beiden Fällen wird die vorherige Koordinate der Ergebnisliste hinzugefügt, da sich der Suchkreis zu diesem Zeitpunkt noch komplett auf der Fahrbahn beziehungsweise auf der Strecke befand.

Die vier Phasen sind in Abbildung 4.5 dargestellt.

Der Algorithmus hat allerdings natürliche Grenzen. Ist der Suchradius größer als die Hälfte der schmalsten Stelle auf der Fahrbahn, so kann an diesen schmalen Stellen kein Toleranzkorridor identifiziert werden, da der Suchkreis dort nie komplett auf der Fahrbahn ist, sonder immer ein Stück daneben. Wird der Suchradius sogar größer gewählt, als die breiteste Fahrbahnstelle, so überlappt der Suchkreis die Fahrbahn ständig und es werden überhaupt keine Koordinaten ermittelt.

[7] siehe Abschnitt 2.2

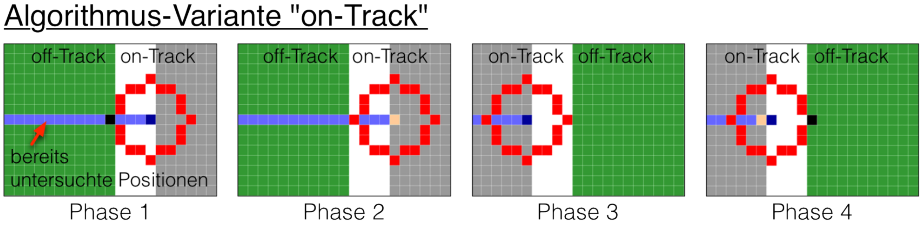


Abbildung 4.5: Die abgebildeten Phasen zeigen die Methode, wie ein Toleranzkorridor auf der Fahrbahn ermittelt wird (maximale linke beziehungsweise rechte Position auf der Fahrbahn).

Wenn der Algorithmus mit einer Kachel fertig ist und die CSV-Datei abgelegt wurde, wird zur Überprüfung eine Färbemethode aufgerufen, die die identifizierten Koordinaten auf der Kachel einfärbt und die veränderte Grafik in einer Kopie speichert. Abbildung 4.6 zeigt das Ergebnis einer OnTrack-Analyse mit einem Suchradius von 15 Pixeln auf der Kachel *ELl.bmp*, welche eine eckige Gabelung enthält. Die Grenzen des Toleranzkorridors wurden blau markiert. Dieses Beispiel wurde gewählt um zu verdeutlichen, dass auch mehrere Fahrbahnzweige korrekt erkannt werden.

Der Nutzen, den diese Analyse und die gewonnenen Koordinaten mitsich bringt, ist eine einfachere und zugleich exaktere Methode zur Feststellung, ob sich das Trackingobjekt innerhalb beziehungsweise außerhalb eines vorgegebenen Toleranzkorridors bewegt. Hier sei noch einmal an die Konzepte erinnert, die mit Automatikfunktionen verhindern möchten, dass das Trackingobjekt die Fahrbahn verlässt. Wurde eine On-Track-Analyse mit einem Suchkreisradius von 15 (= Radius des Trackingobjekts) durchgeführt, so erhält man genau den Toleranzkorridor bei dem das Trackingobjekt gerade so den Fahrbahnrand nicht überschreitet, wenn dessen Mittelpunkt zwischen zwei ermittelten, zusammengehörenden Koordinaten liegt, die sich mit dem Trackingobjekt-Mittelpunkt auf gleicher Höhe befinden. Allerdings gibt es bei der Bestimmung, ob sich ein Punkt nun auf der Fahrbahn oder abseits davon befindet, noch einige Probleme, die in Abschnitt 4.4 behandelt und gelöst werden.

4.1.2 Variante 2: Analyse neben der Fahrbahn

Eingige Konzepte sehen Toleranzkorridore außerhalb der Fahrbahn vor, weshalb diese zweite Methode zur Analyse von Kacheln erforderlich wurde. Sie bestimmt Toleranzkorridore links und rechts von den Fahrbahnrändern neben der Fahrbahn. Die dafür zuständige Klassenmethode befindet sich ebenfalls in der Klasse AAFTrackTools und besitzt die gleiche Signatur, wie die On-Track-Methode:

AAFTrackTools
locateBoundsOffTrack: *aSourceFile*
in: *aSourceFolder*
to: *aTargetFolder*
with: *aRadius*

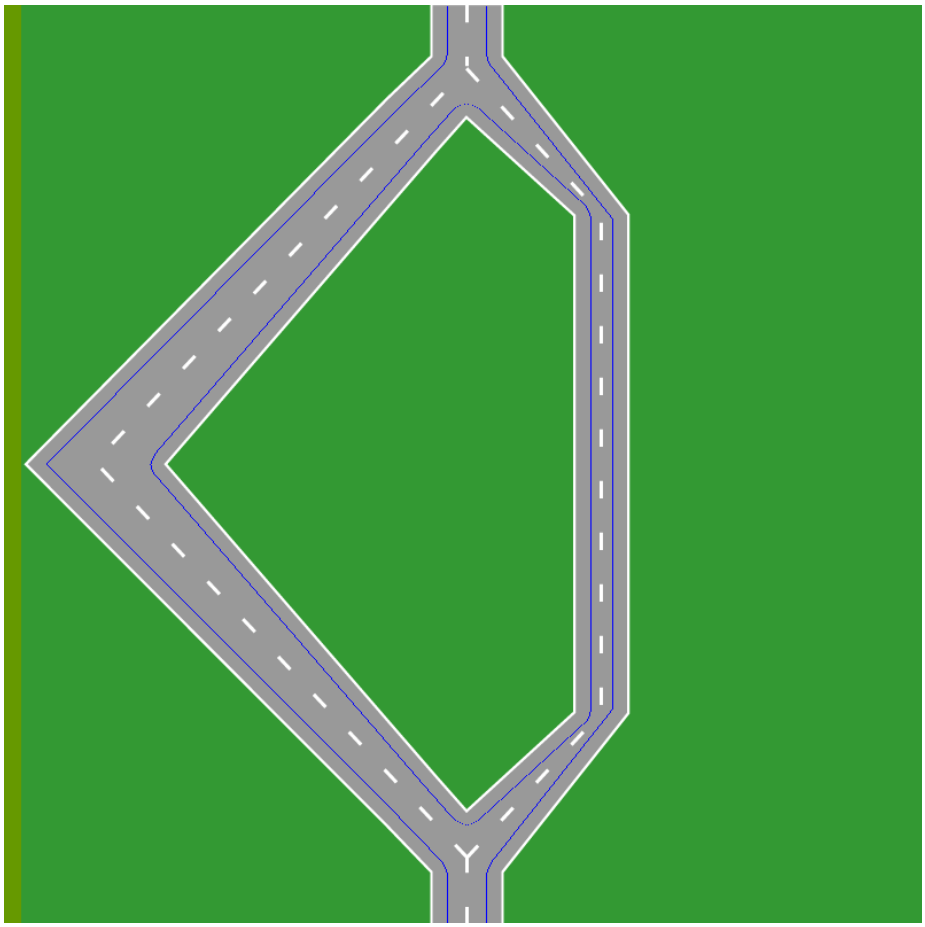


Abbildung 4.6: Auf der Kachel ELl.bmp wurden die Ergebniskoordinaten für die 15-Pixel-OnTrack-Analyse blau gefärbt.

Da diese Methode die selben Parameter verwendet, wie die On-Track-Methode, können deren Bedeutungen im vorherigen Unterabschnitt 4.1.1 nachgelesen werden.

Das Prinzip des Off-Track-Analysealgorithmus ähnelt dem der On-Track-Analyse: Auch diese Version geht zeilen beziehungsweise pixelweise vor, unterscheidet sich aber bei der Auswertung der zwischengespeicherten Farbwerte des Suchkreises.

- Phase 1: Der Suchkreis erreicht die Fahrbahn: Unter den zwischengespeicherten Farbwerten befindet sich erstmals eine Fahrbahnfarbe!
- Phase 2: Die Koordinaten der vorherigen Position werden in die Ergebnisliste aufgenommen.
- Phase 3: Der Suchkreis wird solange weiter verschoben, bis eine Situation entdeckt wird, in der...
- Phase 4: ...unter dem Suchkreis nur noch grüne Farbwerte registriert werden oder der Rand der Kachel erreicht wurde. Die gerade aktuellen Koordinaten werden der Ergebnisliste hinzugefügt.

Die vier Phasen werden in Abbildung 4.7 dargestellt.

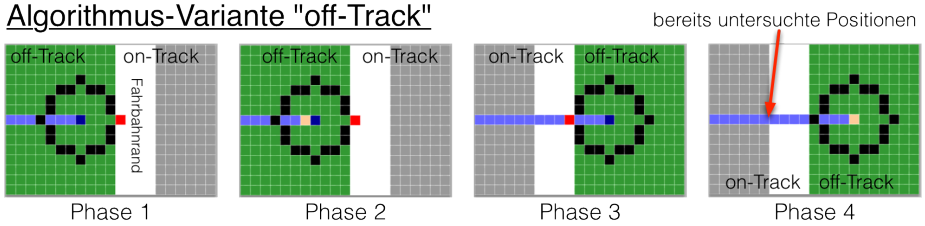


Abbildung 4.7: Die abgebildeten Phasen zeigen die Methode, wie ein Toleranzkorridor außerhalb der Fahrbahn ermittelt wird (links beziehungsweise rechts neben der Fahrbahn).

Auch dieser Algorithmus hat einige Besonderheiten, die zu beachten sind. Wird unter den Suchkreiskoordinaten gleich zu Beginn einer Zeile eine Fahrbahnfarbe gefunden, so wird per se die erste Intervall-X-Koordinate auf 15[8] plus 15[9] gesetzt. Dabei handelt es sich um den Minimalabstand für das Trackingobjekt zum linken Streckenrand. Ähnlich verhält es sich am Ende einer Pixelzeile. Wird bis zur Position [Streckenbreite – 15][10] keine Situation gefunden, in der keine Fahrbahnfarben unter den zwischengespeicherten Farbwerten des Suchkreises sind, so wird als zweite (rechte) X-Koordinate 785 gespeichert. Somit ist sichergestellt, dass sich alle gespeicherten Koordinaten im Bildbereich befinden. Außerdem erlauben es die äußeren Grenzen nicht, dass sich das Trackingobjekt außerhalb der Strecke aufhalten dürfte.

[8] Für den Radius des Trackingobjekts.
[9] Offset für die Colorcodes am linken Bildrand welcher 15 Pixel breit ist.
[10] Entspricht $800 - 15 = 785$ (15 Pixel Abzug für den Radius des Trackingobjekts).

Auch nachdem diese Analysemethode beendet wurde, findet der Aufruf der Färbemethode statt, die die identifizierten Koordinaten auf der Kachel einfärbt und die veränderte Grafik in einer Kopie speichert. Abbildung 4.8 zeigt die Einfärbung auf der Kachel *ELl.bmp* für die Außenabstände 15, 45, 75 und 105 Pixel. Man kann darauf gut erkennen, dass sich am linken Bildrand die Toleranzkorridore zu einer einzelnen Linie vereinen. Dies ist der zuvor beschriebenen Besonderheit geschuldet, da hier gleich an der ersten Position Fahrbahnfarbwerte entdeckt wurden und somit der genannte Offset als erste X-Koordinate verwendet wurde.

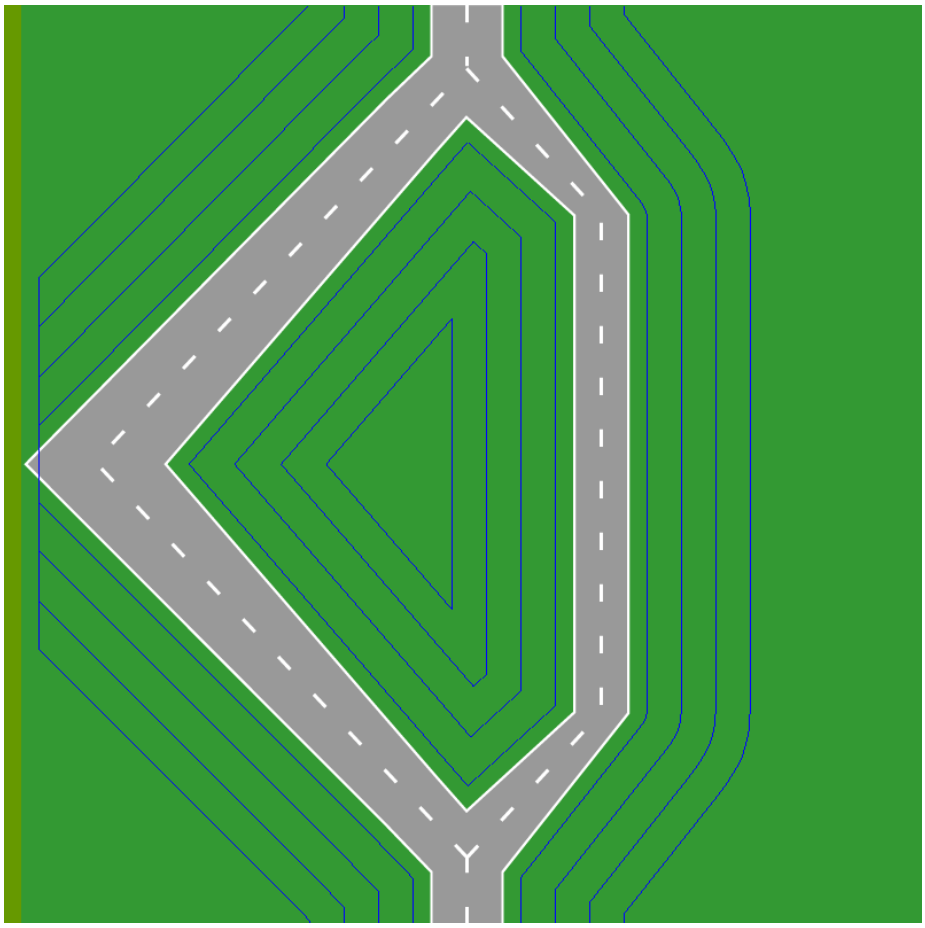


Abbildung 4.8: Auf der Kachel ELl.bmp wurden die Ergebniskoordinaten für die 15-, 45-, 75- und 105-Pixel-OffTrack-Analysen blau gefärbt. Gut zu erkennen ist, wie am linken Bildrand Offset beziehungsweise Minimalabstand realisiert wurden.

4.2 Eine Datenstruktur für Toleranzkorridor-Koordinaten

Dieses Kapitel behandelt den Umgang mit den Daten, die bei der Streckenteilanalyse (siehe Abschnitt 4.1) ermittelt worden sind.

Ursprünglich wurden die Daten direkt von den jeweiligen Agenten, die sie benötigten, mit Hilfe einer Methode der Klasse AAFTrackTools aus den bei der Analyse generierten CSV-Dateien eingelesen, geparst und dann als Bibliothek (Squeak Dictionary) bereitgestellt. Dieser Vorgang benötigte jedoch relativ viel Zeit (bis zu 45 Sekunden pro Step), weshalb eine Neuorganisation der Datenhaltung in drei Phasen vorgenommen wurde.

In der ersten Phase der Neuorganisation wurde eine Struktur entworfen, die es ermöglicht, die vorhandenen Daten schneller bereitzustellen. Außerdem sollte sie die Möglichkeit bieten, weitere Datensätze dem Datenbestand ohne großen Aufwand hinzuzufügen. Ein Datensatz besteht hierbei nicht aus einer Folge unterschiedlicher Attribute, sondern lediglich aus einer großen Zeichenkette (String). Diese Zeichenkette entspricht ganz genau dem Inhalt einer CSV-Datei, um auch weiterhin vom selben Parser eingelesen werden zu können. Wie die Datenstruktur aufgebaut ist, wird in Unterabschnitt 4.2.1 beschrieben. Realisiert wurde die Datenhaltung, indem die Grenze zwischen Programm und Daten aufgehoben und der ganze Datenbestand in den Quelltext integriert wurde. Eine Rechtfertigung für diesen Schritt liefert die Natur der Daten selbst, denn im Grunde handelt es sich dabei nur um sehr große Konstanten, da die Koordinaten auf keine Weise bearbeitet werden müssen. Sie werden lediglich gelesen und dienen auf diese Art nur Informationszwecken.

Um die Inhalte der CSV-Dateien auf einfache Art und Weise in ihre jeweilige Datenklasse zu integrieren, wurde in der folgenden Phase ein Codegenerator entwickelt (siehe Unterabschnitt 4.2.2). Dieser ist zudem auch künftig wiederverwendbar, da er in der Lage ist, auch aus neuen Analysedaten weitere Datenklassen zu erzeugen.

Die letzte Phase sah dann die Anpassung der Methoden vor, die bisher die Daten aus den CSV-Dateien eingelesen und bereitgestellt haben. Diese mussten vom externen Datenimport aus den CSV-Dateien zum Zugriff auf die interne Datenstruktur umgestellt werden. Die nächsten Unterabschnitte erläutern den Aufbau und die genaue Organisation der Datenklassen, den Codegenerator für die Überführung von CSV-Daten in die Datenklassen, stellen den Datenstruktur-Zugriffsalgorithmus vor und beschreiben, wie die Daten der einzelnen Kacheln zu einer Datensammlung für eine komplette Strecke zusammengesetzt werden (Unterabschnitt 4.2.3).

4.2.1 Aufbau der Datenstruktur

Für den Aufbau der Datenstruktur wurde die bereits existierende Ordnerstruktur der CSV-Dateien als Vorlage verwendet. Diese sah vor, dass für jede Analyse ein eigener Ordner angelegt wurde, in dem sich die CSV-Dateien mit den Tole-

ranzkorridor-Koordinaten befanden[11]. Diese Ordner wurden nach einem besonderen Schema benannt:

<Art der Analyse><Analyseradius>[12].

Auf ähnliche Weise wurden auch die Datenklassen in der Kategorie AAF-TrackData benannt, nur dass sie noch das Präfix AAF erhielten. Jede dieser Datenklassen enthält nach den Kacheln benannte Methoden, die als Getter fungieren und einen Datenstring zurückgeben, dessen Inhalt exakt dem Inhalt der entsprechenden CSV-Datei entspricht. Zusätzlich gehören jeder dieser Datenklassen eine Reihe von Methoden, die den Zugriff auf die Daten über die Interface-Klasse AAFTrackData[13] ermöglichen, an.

Tatsächlich liegen alle Daten in Form eines Dictionary[14] vor. Die Schlüssel dieses Dictionary sind Strings, die den Namen einer jeweiligen Datenklasse enthalten. Zu jedem Schlüssel wird wieder ein Dictionary gespeichert. Deren Schlüssel sind ebenfalls Strings. Sie haben als Inhalt die Namen aller Kacheln. Zu jedem Dictionary-Eintrag wird in dieser zweiten Ebene der oben genannte Datenstring der jeweiligen Kachel gespeichert. Abbildung 4.9 zeigt den Aufbau auf grafische Weise. Die Datenstrings sind natürlich nur angedeutet, da sie im System deutlich länger sind. Die Syntax dieser Strings wird durch folgenden regulären Ausdruck beschrieben:

'(<Zeilennummer>(;<linkeXKoordinate>;<rechteXKoordinate>)$^+$#)$^+$'

Der ganze String wird von einfachen Hochkommata umschlossen ('). Ein Semikolon (;) trennt einzelne Zeilennummern beziehungsweise X-Koordinaten voneinander und ein Doppelkreuz (#) ganze Zeilen inklusive deren zugehörige Zeilennummern und Koordinaten. Zu jeder Zeilennummer gehört mindestens ein Koordinatenpaar und je Kachel muss mindestens eine Zeile existieren (jeweils realisiert durch den +-Operator).

4.2.2 Codegenerator

Wie bereits zu Beginn dieses Abschnitts erwähnt, entstand die Datenstruktur aus der vorhandenen Ordnerstruktur der CSV-Dateien. Die Inhalte der bereits existierenden CSV-Dateien (onTrack15, offTrack15, 45, 75 und 105) hätten auch manuell in die Struktur kopiert und die notwendigen Methoden manuell angelegt werden können.

Zwei Gründe sprachen für die Entwicklung eines automatisierten Imports. Zum einen entstehen bei händischen Copy&Paste-Vorgängen häufig Fehler (siehe [Wickert (2012)]). Zum anderen ermöglicht ein Codegenerator auch das problemlose Hinzufügen zukünftig erzeugter Analyseergebnisse zum Bestand und die automatische Erzeugung der redundanten Getter-Methoden.

[11] Jeweils benannt nach der Kachel, deren Analysedaten darin enthalten waren
[12] Zum Beispiel „onTrack15“ oder „offTrack45“
[13] siehe Unterabschnitt 4.2.3
[14] Eine Squeak-Datenstruktur mit Schlüssel-Attribut-Beziehungen

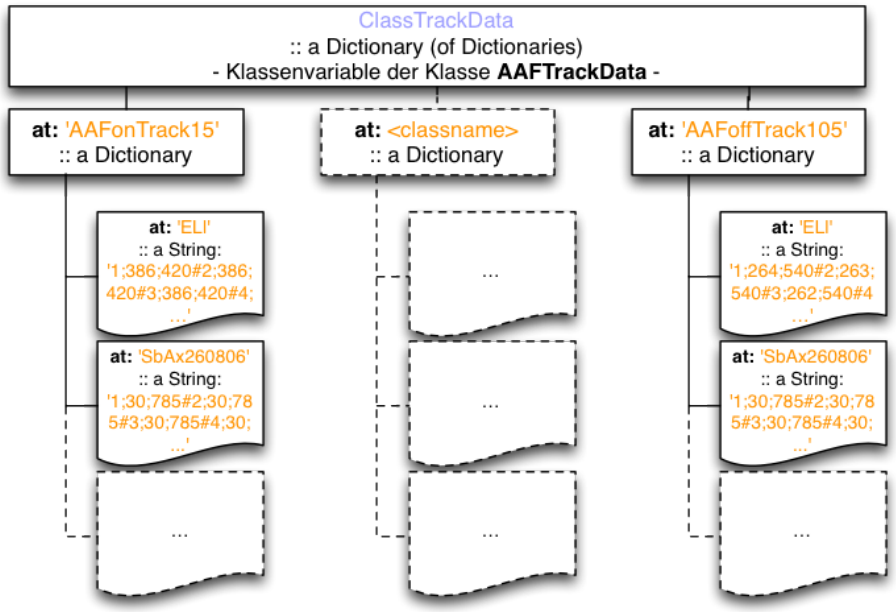


Abbildung 4.9: Aufbau der Datenstruktur.

Der Codegenerierungsprozess selbst ist nicht sehr komplex, da der überwiegende Teil aus statischem Quelltext besteht. Nur an wenigen Stellen müssen minimale dynamische Aspekte berücksichtigt werden. Beispielsweise ist der Name der Datenklasse solch eine Stelle. Die Anzahl und Benennung der Methoden ist für alle Datenklassen gleich. Lediglich der Inhalt der in Unterabschnitt 4.2.1 erwähnten und nach den entsprechenden Kacheln benannten Getter-Methoden ist selbstverständlich verschieden. Hier kommt ein CSV-Datei-Parser zum Einsatz, der den Datenstring aus den jeweiligen CSV-Dateien heraus kopiert und in die Ausgabedatei[15] einfügt. Die resultierende Quelltextdatei ist so strukturiert und formatiert, dass sie von Squeak problemlos als externe Klasse erkannt und in die Kategorie AAF-TrackData importiert werden kann. Die Syntax einer solchen Squeak-Quelltextdatei ist nachfolgend als regulärer Ausdruck aufgeführt:

Object subclass: #<ClassName>
    instanceVariableNames: '(<InstanceVariableName>)*'
    classVariableNames: '(<ClassVariableName>)*'
    poolDictionaries: '(<PoolDictionary>)*'
    category: '<CategoryName>'!
(! <ClassName>
    commentStamp: '<CoderShortName> <TimeStamp>'
    prior: <aNumber> !
  <CommentText>
!)*

"- "!

[15] eine Squeak-Quelltextdatei mit der Erweiterung st

<ClassName> class
    instanceVariableNames: '(<InstanceVariableNames>)*'!

(! <MethodName> (class)?
    methodsFor: '(<MethodCategory>)?'
    stamp: '<CoderShortName> <TimeStamp>'!
  <MethodBlock> ! !)*

Eine Squeak-Quelltextdatei besteht aus zwei Hauptbereichen: Dem Kopfbereich und dem Methodenbereich, der unterhalb der gestrichelten Linie beginnt.

Der Kopfbereich enthält den Klassennamen und die Kategorie, der die Klasse angehört. Optional können die Namen von Klassen- und Instanz-Variablen als auch Pool-Dictionaries vordefiniert werden. Zudem wird dort der Klassenkommentar mit den zugehörigen Metainformationen (zum Beispiel der CommentStamp mit dem Kürzel des Programmierers und dem Datum der letzten Änderung) abgelegt.

Im Methodenbereich sind die Klassen- und Instanz-Methoden enthalten. Jede Methode hat einen Namen (MethodName → Methoden-Signatur) und kann optional einer Methodenkategorie (MethodsFor) zugeordnet werden. Im Methodenrumpf (MethodBlock) folgt dann der eigentliche Methodenquelltext.

Nach diesem Muster werden für jede Datenklasse – zusätzlich zu den Datenstring-Gettern – folgende Methoden erzeugt:

- **getData:** aTileName :: liefert zu einer bestimmten Kachel deren Datenstring zurück – Verwendung nur datenklassenintern.
- **loadAllData** :: diese *Hauptlademethode* füllt die Datenstruktur mit den Daten aller vorhandenen Kacheln – Verwendung über das AAFTrackData-Inerface.
- **loadIndividualDataFor:** aTileName :: liefert für eine bestimmte Kachel eine OrderedCollection zurück, die alle Toleranzkorridor-Koordinaten für jede Pixelzeile beinhaltet – Verwendung über das AAFTrackData-Interface.

4.2.3 Initialisierung und Zugriff auf die Datenstruktur

Der Zugriff auf die Datenstruktur erfolgt über eine zentrale Interface-Klasse: AAFTrackData. Diese Klasse ist dafür verantwortlich, dass die Datenstruktur geladen wird, damit auf eine Anfrage hin die Daten bereitstehen und zurückgeliefert werden können. Die Methode, über die der *Zugriff* auf die Datenstruktur generell erfolgt, hat folgende Signatur:

AAFTrackData
    getBounds: aTileName
    for: onOrOffTrackSymbol
    with: aRadius
    at: yValue

aTileName ist der Name einer Kachel als String; onOrOffTrackSymbol ist ein Symbol und darf die Werte #onTrack oder #offTrack annehmen; aRadius gibt den ganzzahligen Abstand zum Fahrbahnrand in Pixeln an; yValue

ist eine ganzzahlige Pixelzeilennummer. Diese Parameter identifizieren genau eine *Kollektion von Toleranzkorridor-Koordinaten* einer speziellen Pixelzeile mit einem bestimmten Abstand zum Fahrbahnrand auf oder neben der Fahrbahn auf der angegebenen Kachel.

Beim allgemeinen Initialisieren der Datenstruktur über die Methode AAFTrackData loadCompleteBoundData werden nacheinander die Hauptlademethoden (siehe Unterabschnitt 4.2.2) aller Datenklassen aufgerufen, die in der Sammlung mit Klassennamen enthalten sind. Wurde diese Initialisierung nicht durchgeführt, bevor über die oben genannte Methode eine Anfrage an die Datenstruktur gestellt wird, können auch ad-hoc einzelne (komplette) Datenklassen in die Datenstruktur geladen werden. Dazu wird anhand der Parameter ein Schlüssel aus OnTrack beziehungsweise OffTrack und dem Fahrbahnabstand gebildet. Daraus ergibt sich der Name der zu ladenden Datenklasse[16], deren Hauptlademethode aufgerufen wird. Sind alle angeforderten Daten in die Datenstruktur eingefügt worden, findet – mit dem gleichen Schlüssel – eine geschachtelte Dictionary-Abfrage statt, die als Ergebnis die Toleranzkorridor-Koordinaten der angegebenen Pixelzeile (yValue) als OrderedCollection zurück gibt. Diese wird dann in dem Algorithmus (siehe folgenden Unterabschnitt 4.2.4) weiterverwendet, der den Toleranzkorridor für eine komplette Strecke zusammensetzt.

4.2.4 Datenbereitstellung zur Verwendung in Agenten

Zum leichteren Verständnis des Algorithmus, der die Daten zur Verwendung in den Agenten bereitstellt, sei noch einmal auf die Unterabschnitte 2.4.1 und 2.4.2 hingewiesen. Darin werden die Versuchskonfigurationsdatei *steps.txt*, die Streckenkonfigurationsdateien und die verschiedenen Kacheln beschrieben.

Der Ablauf des Algorithmus beginnt damit, dass eine Kachelnamensammlung angelegt wird, die dem Streckenaufbau entspricht. Das heißt es wird die Reihenfolge und Art jeder Kachel – beginnend mit der Start-Kachel und abschließend mit der Ziel-Kachel – in einer geordneten Sammlung abgelegt. Anschließend wird jeder einzelne Eintrag der Sammlung abgearbeitet.

Zunächst wird die Höhe der aktuellen Kachel bestimmt. Diese wird als Abbruchkriterium für die nachfolgende Schleife benötigt, denn nun wird für jede Pixelzeile der aktuellen Kachel die zugehörige Sammlung von Toleranzkorridor-Koordinaten in die Streckenbibliothek trackDictionary an ihrer endgültigen Position eingefügt. Diese Sammlung wird über die Methode **AAFTrackData getBounds:** currentTile **for:** aSymbol **with:** aRadius **at:** currentTileLine direkt aus der Datenstruktur geholt. Dabei musste auf eine Besonderheit geachtet werden: Die Pixelzeilen der Kacheln wurden bei der Streckenanalyse in der *Squeak-Orientierung* abgelegt. Das bedeutet die Nummerierung der Zeilen erfolgte von oben nach unten aufsteigend. In SAM ist die Orientierung jedoch umgekehrt, also von unten nach oben – vom Start zum Ziel – aufsteigend. Daher ergibt sich für das Erstellen der Streckenbibliothek folgendes: Die Pixelzeilen müssen *absteigend* in die Streckenbibliothek eingefügt

[16] zum Beispiel AAFonTrack15

werden; für jede Kachel wird also mit der letzten Pixelzeile[17] begonnen und mit der ersten abgeschlossen. Wie das genaue Mapping der einzelnen Kachelzeilen auf die Streckenzeilen erfolgt ist in Abbildung 4.10 detailliert dargestellt.

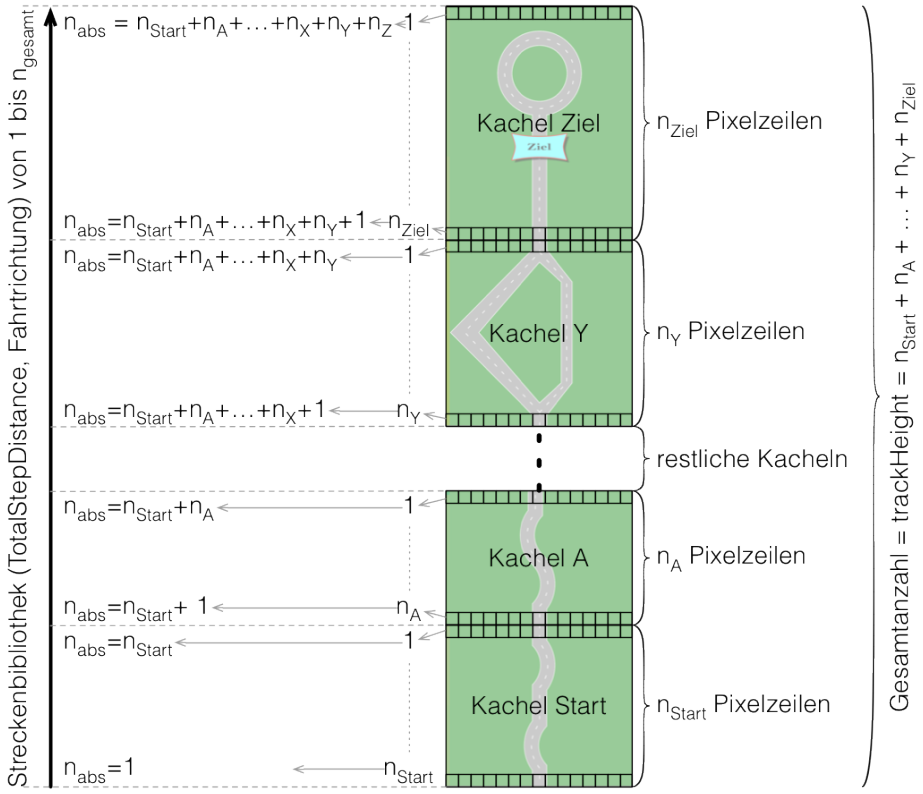


Abbildung 4.10: Mapping von Kachelzeilen auf Streckenzeilen.

Wurden alle Koordinatensammlungen in der Streckenbibliothek abgelegt, wird diese als Rückgabeobjekt an den aufrufenden Agenten zurück geschickt.

4.3 Geometrie

Für die Abschnitte weiter hinten in diesem Kapitel ist ein kurzer Exkurs in das Fachgebiet der *Geometrie* erforderlich. In diesem Abschnitt wird kurz auf die *Geradengleichung in der Ebene*, deren *Zwei-Punkte-Form* und weitere geometrische Operationen in der Ebene eingegangen, um die Grundlagen für die folgenden Abschnitte zu schaffen.

Die Steigungs–y-Abschnittsform der Geradengleichung
Die wohl bekannteste Form, wie eine Gerade in der Ebene beschrieben werden

[17] diese Zeilennummer ist gleich der Kachelhöhe

kann, ist die Steigungs–y-Abschnittsform (dargestellt in Abbildung 4.11):

$$f(x) = y = m \cdot x + n$$

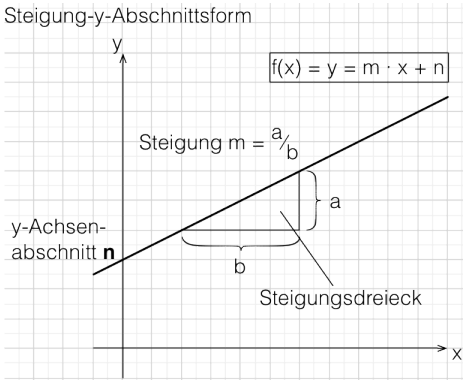


Abbildung 4.11: Steigung–y-Abschnittsform der Geradengleichung.

Die Punkt–Steigungsform (Einpunktform) der Geradengleichung
Die Punkt–Steigungsform

$$y - y_1 = m \cdot (x - x_1)$$

wird benutzt, um weitere Punkte zu bestimmen, die auf einer Geraden liegen, welche mit einer bestimmten Steigung m durch den gegebenen Punkt P_1 verläuft. Zur Veranschaulichung wird diese Form in Abbildung 4.12 visualisiert.

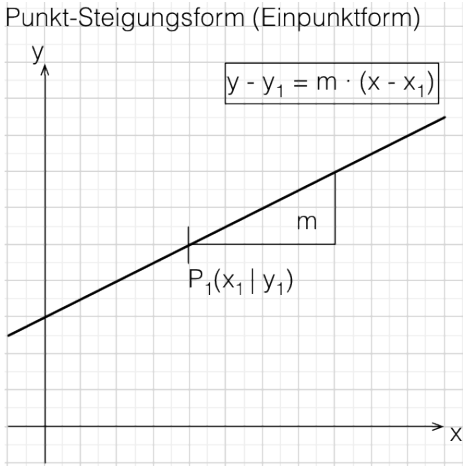


Abbildung 4.12: Punkt-Steigungsform (oder auch Einpunktform) der Geradengleichung.

Zweipunkteform der Geradengleichung
Man gebraucht diese Gleichungsform, um aus zwei gegebenen Punkten P_1 und

P_2 die Gleichung der Geraden zu bestimmen, welche durch die beiden Punkte verläuft. Mit den Koordinaten der gegebenen Punkte lassen sich alle notwendigen Komponenten der Gleichung bestimmen. Die Steigung:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

Wendet man den Strahlensatz an, erhält man für jeden beliebigen Punkt $P(x|y)$:

$$m = \frac{y - y_1}{x - x_1}$$

Setzt man beides gleich, kann man nach y auflösen:

$$\begin{aligned}
\frac{y - y_1}{x - x_1} &= \frac{y_2 - y_1}{x_2 - x_1} \\
\frac{y}{x - x_1} - \frac{y_1}{x - x_1} &= \frac{y_2 - y_1}{x_2 - x_1} && | + \frac{y_1}{x - x_1} \\
\frac{y}{x - x_1} &= \frac{y_1}{x - x_1} + \frac{y_2 - y_1}{x_2 - x_1} && | \cdot (x - x_1) \\
y &= y_1 + \frac{y_2 - y_1}{x_2 - x_1} \cdot (x - x_1)
\end{aligned}$$

Danach erhält man den y-Achsenabschnitt, indem man $x = 0$ setzt:

$$n = y_1 - \frac{y_2 - y_1}{x_2 - x_1} \cdot x_1$$

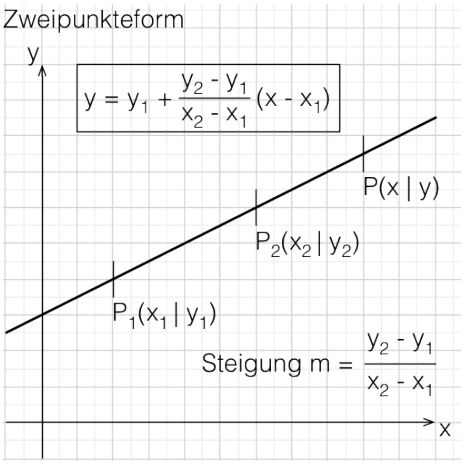


Abbildung 4.13: Zweipunkteform der Geradengleichung.

Die Bestimmung der (relativen) Lage eines Punktes bezüglich einer Geraden

Ebenfalls von großer Bedeutung für die folgenden Abschnitte ist die Bestimmung der Lage eines Punktes bezüglich einer bestimmten Geraden in der Ebene. Für die nachfolgend erklärten Mechanismen ist es wichtig zu erkennen, ob ein Punkt

oberhalb oder unterhalb von der Geraden oder sogar darauf liegt. Im Falle der Parallelität von Gerade und y-Achse muss lediglich der x-Wert der Geraden mit dem des Punktes verglichen werden: Ist der Wert des Punktes kleiner beziehungsweise größer, so liegt er links beziehungsweise rechts von der Geraden. Ist er exakt gleich, so befindet er sich darauf. Existiert eine Geradenfunktion, so lässt sich durch Einsetzen der Koordinaten des Punktes in die Geradengleichung ermitteln, ob der Punkt oberhalb, unterhalb oder auf der Geraden liegt. Ein Fall, in dem die Gerade parallel zur x-Achse verlaufen würde, kann nicht eintreten, da aufeinanderfolgende Toleranzkorridor-Koordinaten immer unterschiedliche y-Werte haben. Für y setzt man den y-Wert des Punktes ein und der x-Wert wird entsprechend für das x in der Gleichung eingesetzt. Rechnet man den hinteren Teil aus und ist dieser größer beziehungsweise kleiner als der vordere eingesetzte y-Wert, so befindet sich der Punkt oberhalb beziehungsweise unterhalb von der Geraden. Sind beide Seiten gleich, so liegt der Punkt exakt auf der Geraden.

Der Punkt auf einer Geraden für einen bestimmten Funktionswert
Wie schon eingangs erwähnt, kann für einen gegebenen x-Wert der zugehörige y-Wert (Funktionswert) berechnet werden. Daneben ist aber auch die nach x umgestellte Formel von Bedeutung. Ist nämlich nur der Funktionswert bekannt, kann bei einer Geraden, die weder parallel zur x-Achse noch zur y-Achse verläuft, ebenso der zugehörige x-Wert errechnet werden:

$$\begin{array}{rcll} y & = & m \cdot x + n & | - n \\ y - n & = & m \cdot x & | : m \\ \dfrac{y - n}{m} & = & x & \end{array}$$

4.4 Subpixelverhalten in Squeak

Der Hintergrund für die aufgeführten geometrischen Funktionen ist das Subpixelverhalten von Squeak. Dieser Begriff wurde innerhalb der ATEO Arbeitsgruppe geprägt und bezeichnet die Tatsache, dass Pixel in Squeak nicht als atomare Einheit verwendet werden. Squeak berechnet Positionen in Grafiken auch noch in Bruchteilen von Pixeln, sodass die durch die Analyse bestimmten Toleranzkorridore erhebliche Lücken aufweisen, da sie nur aus ganzzahligen Koordinatenwerten bestehen und dazwischen unendlich viele reelle Werte liegen. Der Versuch, diese zu ignorieren und geschickt die Koordinaten zu runden, wurde verworfen, denn dadurch wäre das Fahrverhalten des Trackingobjekts verändert worden. Statt der Maximalgeschwindigkeit von 20,48 Pixel pro Tick wären nur noch 20 und statt der maximalen horizontalen Reichweite von ca. 13,64 je Richtung sogar nur noch 13 Pixel pro Tick möglich gewesen. Diese Änderungen hätten dazu geführt, dass künftige Experimente nicht mehr vergleichbar gewesen wären. Daher musste eine Lösung für das Problem auf Subpixelebene gefunden werden, um die Vergleichbarkeit zu wahren und eine exakte Genauigkeit zu gewährleisten.

Erreicht wurde dies dadurch, dass aufeinander folgende Toleranzkorridor-Koordinaten bei Bedarf durch je eine virtuelle Gerade miteinder verbunden wurden. Diese Methode erlaubt auch mit reellen Werten zu arbeiten. Die Koordinaten

des Trackingobjekt-Mittelpunkts müssen nicht mehr gerundet und direkt mit den Koordinaten des Toleranzkorridors verglichen werden. Die genaue Positionsbestimmung (also ob eine bestimmte Position innerhalb oder außerhlab eines Toleranzkorridors liegt) erfolgt nun mit der *oberen und unteren Schranke*[18] einer beliebigen Position. Dieses generelle Prinzip ist in Abbildung 4.14 dargestellt. Jedes der großen Quadrate entspricht einem Pixel. Die umrundeten Eckpunkte der Pixel (a und b) markieren die Toleranzkorridor-Koordinaten welche durch eine virtuelle Gerade miteinander verbunden sind. Der schwarz gefüllte Kreis markiert die aktuell betrachtete Position des Trackingobjekt-Mittelpunkts, welche beliebig zwischen zwei Pixelzeilen beziehungsweise-spalten liegen kann. Die beiden Kreuze deuten die untere beziehungsweise obere Schranke (in diesem Fall als 'außerhalb' markiert) der Punktkoordinaten an.

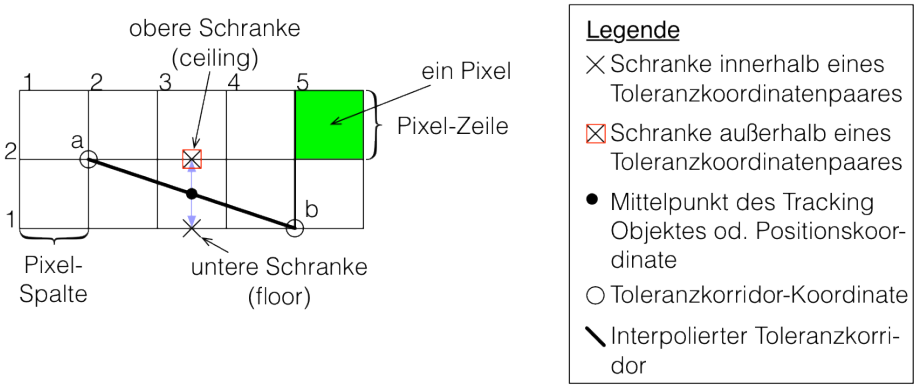


Abbildung 4.14: Starke Vergrößerung des Pixel-Grids einer Grafik.

Um die Antwort auf die Frage zu ermitteln, ob sich eine bestimmte Position des Trackingobjekts innerhalb oder außerhalb eines Toleranzkorridors befindet, wurde (beispielhaft bezogen auf die zuvor genannte Abbildung) folgender Algorithmus entwickelt:

1. Bilde die obere und untere Schranke der aktuellen Position.
2. Prüfe, ob die untere Schranke innerhalb des Toleranzkorridors (Zeile 1) liegt → Ergebnis ist eine Sammlung (Beschreibung siehe unten).
3. Prüfe, ob die obere Schranke innerhalb des Toleranzkorridors (Zeile 2) liegt → Ergebnis ist eine Sammlung der gleichen Form, wie für die untere Schranke.
4. Vergleiche die Ergebnisse:
   - Beide Ergebnisse liefern FALSE: Die Position befindet sich außerhalb des Toleranzkorridors: Antworte mit FALSE.
   - Beide Ergebnisse liefern TRUE: Die Position befindet sich innerhalb des Toleranzkorridors Antworte mit TRUE.

[18] Nur die y-Koordinate des Trackingobjekt-Mittelpunkts wird einmal auf- beziehungsweise abgerundet auf den jeweiligen größeren beziehungsweise kleineren Wert – Ergebnis sind ganzzahlige benachbarte Pixelzeilen.

- Die Ergebnisse sind unterschiedlich (TRUE/FALSE; FALSE/TRUE): Führe eine *Fallbestimmung* durch.

Beschreibung der Sammlung

Die Sammlung enthält drei Elemente. An ihrer erster Stelle den Wahrheitswert, ob sich die Schranke *innerhalb* oder *außerhalb* (dazu zählt auch *mittig*) von Toleranzkorridor-Koordinaten befindet. An zweiter Stelle folgt ein String der Länge zwei, wobei dieser String folgende Werte annehmen kann:

- 'IN' – INside, eine Schranke befindet sich zwischen einem Paar Toleranzkorridor-Koordinaten einer Zeile
- 'LI' – LeftInside, eine Schranke befindet sich zwischen dem linken Paar Toleranzkorridor-Koordinaten einer Zeile (Gabelung, linker Zweig)
- 'RI' – RightInside, analog zu links-innerhalb.
- 'LO' – LeftOutside, eine Schranke befindet sich links-außerhalb der Fahrbahn egal ob einspurig oder gegabelt
- 'RO' – RightOutside, analog zu links-außerhalb
- 'MI' – MIddle, zwischen der ersten rechten Koordinate und der zweiten linken Koordinate (also genau zwischen zwei Paaren Toleranzkorridor-Koordinaten) außerhalb der Fahrbahn im inneren Bereich einer Gabelung

An der dritten Position werden eine oder mehrere Toleranzkorridor-Koordinaten aus der Pixelzeile gespeichert, die auf der selben Höhe der Schranke liegen:

- wenn 'LO' → x-Wert der (ersten) linken Koordinate
- wenn 'RO' → x-Wert der (zweiten) rechten Koordinate
- wenn 'MI' → x-Wert der ersten rechten und zweiten linken Koordinate
- wenn 'LI' → Array mit der ersten linken und rechten Koordinate
- wenn 'RI' → Array mit der zweiten linken und rechten Koordinate
- wenn 'IN' → Array mit der linken und rechten Koordinate

Fallbestimmung

Für die Fallbestimmung sind einige zusätzliche Informationen erforderlich. Hierbei geht es darum festzustellen, wie die virtuelle Gerade verläuft, in welchem Bereich sich die untere beziehungsweise obere Schranke befinden und wie – relativ zur erwähnten Geraden – die zu überprüfende Position liegt.

Für die Bestimmung sind jedoch nicht alle Fälle interessant, da einige bereits damit abgegolten sind, dass zuvor geprüft wurde, ob beide Schranken komplett außerhalb beziehungsweise innerhalb eines Toleranzkorridors liegen (siehe Algorithmus-Schritt 4.a und 4.b). Somit sind die Fälle[19]

- Links-Außerhalb → Links-Außerhalb (a)

[19]Die Buchstaben in den Klammern am Ende jedes Eintrags der Liste dienen als Referenzen zu den jeweiligen Fällen in Abbildung 4.15.

- Rechts-Außerhalb → Rechts-Außerhalb (b)
- Innerhalb → Innerhalb (c)
- Innerhalb → Links-Innerhalb (Übergang von einspuriger Fahrbahn in den linken Zweig einer Gabelung) (d)
- Innerhalb → Rechts-Innerhalb (Übergang von einspuriger Fahrbahn in den rechten Zweig einer Gabelung) (e)
- Links-Innerhalb → Innerhalb (Übergang vom linken Zweig einer Gabelung in die einspurige Fahrbahn) (f)
- Rechts-Innerhalb → Innerhalb (Übergang vom rechten Zweig einer Gabelung in die einspurige Fahrbahn) (g)
- Mittig → Mittig (nur bei Gabelungen) (h)
- Links-Innerhalb → Links-Innerhalb (innerhalb des linkens Zweigs bei Gabelungen) (i)
- Rechts-Innerhalb → Rechts-Innerhalb (innerhalb des rechten Zweigs bei Gabelungen) (j)

nicht weiter zu beachten. Damit bleiben folgende Kombinationen[20] übrig, die noch überprüft werden müssen:

- Links-Außerhalb → Innerhalb (1)
- Innerhalb → Links-Außerhalb (2)
- Rechts-Außerhalb → Innerhalb (3)
- Innerhalb → Rechts-Außerhalb (4)

Zusätzlich die Übergänge in Gabelungen:

- Links-Außerhalb → Links-Innerhalb (5)
- Links-Innerhalb → Links-Außerhalb (6)
- Links-Innerhalb → Mittig (7)
- Mittig → Links-Innerhalb (8)
- diese vier Fälle analog für Rechts (9-12)

Und zwei spezielle Fälle an den Übergängen von einspuriger Fahrbahn in den mittleren Bereich einer Gabelung und aus dem mittleren Bereich einer Gabelung zurück auf die einspurige Fahrbahn:

- Innerhalb → Mittig (13)
- Mittig → Innerhalb (14)

[20] Die Zahlen in den Klammern am Ende jedes Eintrags der Listen dienen als Referenzen zu den jeweiligen Fällen in Abbildung 4.15.

In Abbildung 4.15 sind alle Fälle enthalten, die über eine gesamte Strecke gesehen auftreten können. Die Bedeutung der einzelnen Symbole und Zeichen entspricht der Legende in Abbildung 4.14. Die mit kleinen Buchstaben versehenen Fälle entsprechen den weiter oben erwähnten trivialen Fällen. Die mit Zahlen markierten und umrandeten Fälle müssen einer speziellen Prüfung unterzogen werden, um genau feststellen zu können, ob sich die jeweilige Position innerhalb oder außerhalb des Toleranzkorridors befindet. Wie diese Prüfung abläuft, wird weiter unten in diesem Abschnitt beschrieben. Zur genannten Grafik noch ein Hinweis: Auch wenn die Positionsmarkierungen (gefüllte schwarze Kreise) immer auf gleicher Höhe scheinen, können sie sich aber an jeder beliebigen Stelle zwischen zwei Pixelzeilen beziehungsweise-spalten befinden.

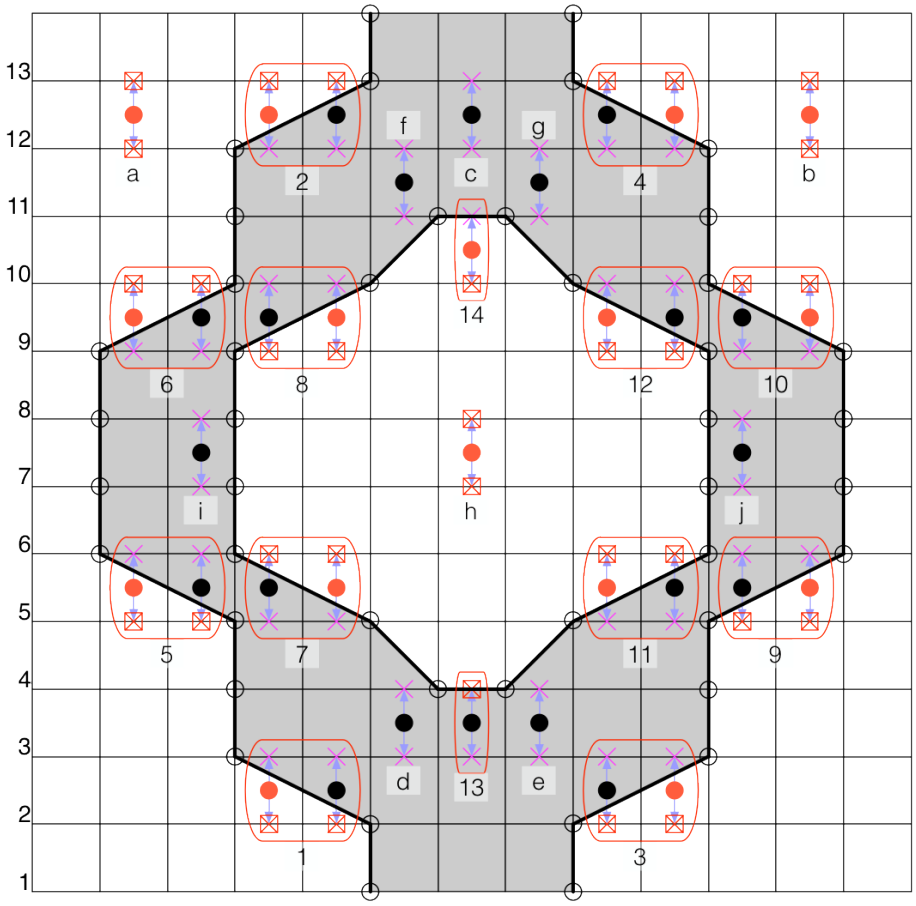


Abbildung 4.15: Eine schematische und stark vereinfachte Darstellung einer Gabelung mit einem kurzen Stück einspuriger Fahrbahn vor (Pixelzeilen 1 bis 3) und nach (Pixelzeilen 12 bis 14) den beiden Gabelungszweigen, um auch die Situationen auf einspurigen Fahrbahnstücken darstellen zu können.

Die 14 in Abbildung 4.15 mit Zahlen versehenen Übergänge werden noch einmal schematisch in Abbildung 4.16 gezeigt. Dabei sei angemerkt, dass manche Übergänge gleich aussehen und sie deshalb zusammengefasst dargestellt werden. Die

Kreise entsprechen den Toleranzkorridor-Koordinaten; Anfang und Ende eines jeden Pfeils stehen für die Verbindung zwischen unterer und oberer Schranke; die Geraden verbinden die zusammengehörenden Toleranzkorridor-Koordinaten von zwei benachbarten Pixelzeilen. Diese Übergänge bilden die Grundlage für die im Folgenden erklärten String-Signaturen.

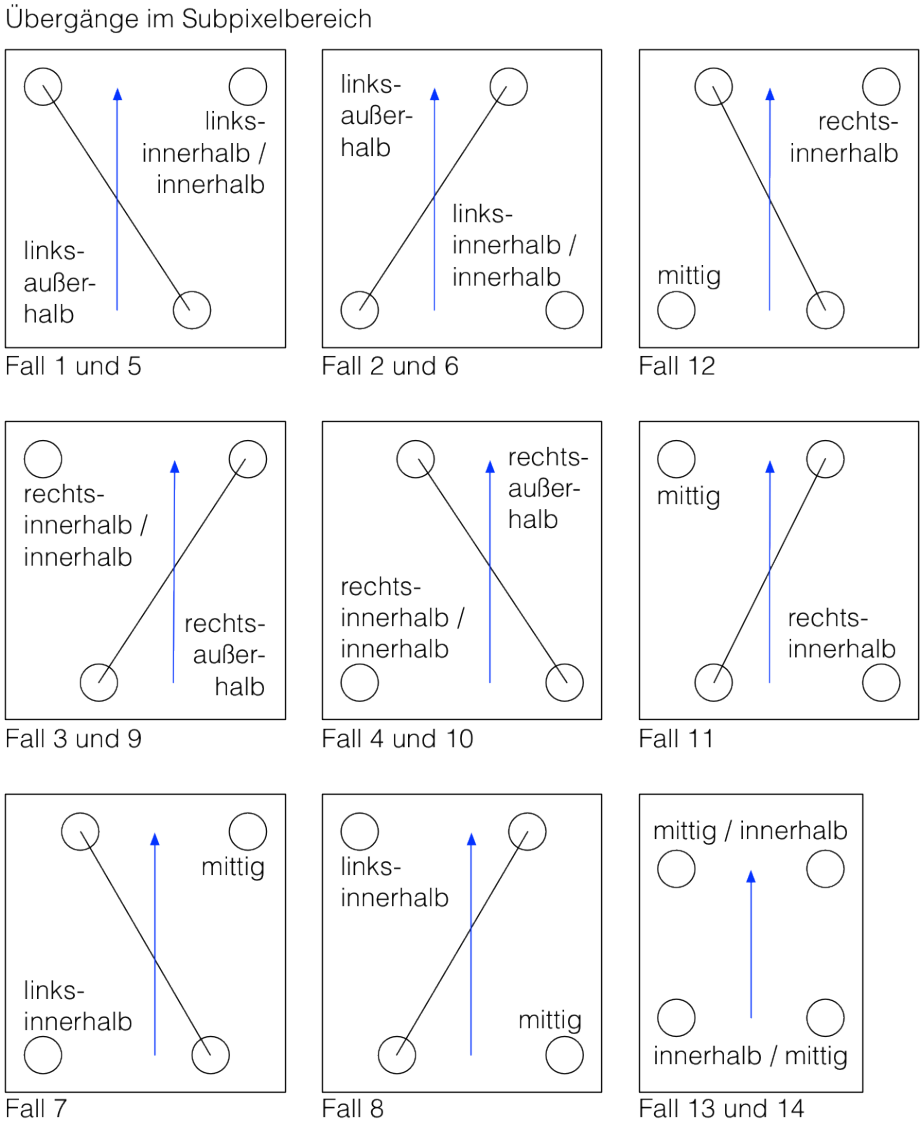


Abbildung 4.16: Alle 14 verschiedenen Übergangsvarianten im Subpixelbereich, die eine Fallbestimmung notwendig machen.

Die Überprüfung

Jeder der 14 Fälle hat eine eigene String-Signatur. Sie wird aus den Teilstrings der beiden anfangs erwähnten Sammlungen (Position 2) zusammengesetzt, die bei der Ermittlung der Wahrheitswerte für die beiden Schranken gefunden wur-

den. Ist die untere Schranke eines Punktes gerade *innerhalb* und die obere *links-außerhalb*, so wird daraus die Signatur *'INLO'* (Fall Nummer (2)) aus den Teilstrings *'IN'* und *'LO'* gebildet. Dies ist die Grundlage, auf der die Überprüfungsfunktion entscheidet, wie in welcher Situation die jeweilige Lage des Punktes und seiner Schranken zu interpretieren ist: Liegt der Punkt *unterhalb* der Geraden, die durch die benachbarten Toleranzkorridor-Koordinaten verläuft, so liegt der Punkt *innerhalb* des Toleranzkorridors, liegt er jedoch *oberhalb*, so befindet er sich *außerhalb*. Auf diese Weise wird für jeden Fall individuell geprüft, welche Schlussfolgerung sich für die gerade betrachtete Position ziehen lässt. Wie dann diese gewonnene Information weiterverwendet wird, ist dann Sache des Aufrufers (Agent oder andere Hilfsmethode).

4.5 Funktionen für die Arbeitsunterstützung diverser Agenten

In diesem Abschnitt werden mehrere Funktionen vorgestellt, die von den Agenten für ihre Arbeit benötigt werden. Darunter sind einige komplexere, aber auch relativ einfache Methoden, die aber dennoch genannt werden, da sie für mehrere Agenten von zentraler Bedeutung sind und somit eine konzeptübergreifende Nutzung erfahren.

4.5.1 Berechnung der SAM-relativen Trackingobjekt-Position

Für viele Agenten und Methoden ist es grundsätzlich wichtig, die genaue SAM-relative Position zu kennen. Für diesen Zweck wurde die Klassenmethode
AAFControlSupport
calcSamRelativeObjectPos
implementiert. Sie berechnet die Koordinaten des Trackingobjekt-Mittelpunkts für den aktuellen Tick. Dieser bildet häufig die Ausgangslage für weitere Methoden, die bestimmte Einflüsse für den folgenden Tick berechnen sollen.

Die x-Koordinate ist die Summe aus totalCalculatedXAxisForBlitter und objectXOffset. Der erste Wert stammt aus dem SAMModelData-Objekt. Er wird von SAM berechnet und dient der korrekten Positionierung des Trackingobjekts bei der Darstellung auf dem Display. Der zweite Wert ist eine Konstante aus der Klasse AAFValues. Er steht für die Breite des Farb-Code-Streifens am linken Rand der Streckendarstellung.

Die y-Koordinate setzt sich ebenfalls aus zwei Werten zusammen: Der totalStepDistance – auch enthalten in SAMModelData – und der Konstanten objectYOffset, welche den Abstand des Trackingobjekt-Mittelpunkts vom unteren Streckenrand zu Beginn einer Fahrt markiert.

4.5.2 Berechnung der Folgeposition des Trackingobjekts auf drei Arten

Ebenso wichtig, wie die genaue Positionsermittlung des Trackingobjekt-Mittelpunkts, ist die Berechnung des Punktes, den das Trackingobjekt mit bestimmten Input-Werten von seiner IST-Position (der Standardparameter aller im Folgenden vorgestellten Methoden) aus ansteuert oder ansteuern würde. Hierfür stehen drei unterschiedliche Methoden zur Verfügung.

Variante 1: Berechnung der Folgeposition mit originalen Input-Werten

Die erste Variante berechnet auf die gleiche Weise wie SAM die Folgeposition des Trackingobjekt-Mittelpunkts. Zunächst wird überprüft, ob sich irgendwelche der Joystick-Input-Werte – die aus einem als Parameter übergebenen SAM-State stammen – im Bereich der Dead-Zone befinden – also weniger als +/- 30 Einheiten ausgelenkt sind. Anschließend werden die Änderungswerte (Deltas) berechnet. Dabei bezeichnet das x-Delta die Verschiebung des Trackingobjekt-Mittelpunkts in x-Richtung und das y-Delta analog.

Das x-Delta errechnet sich über folgende Formel:

$$\frac{(joy1XAxis \cdot inputMwi1) + (joy2XAxis \cdot inputMwi2))}{50 \cdot 1.5}$$

Dabei ist (jeweils analog für Mikroweltbewohner 2)

- joy1XAxis die Auslenkung des Joysticks von Mikroweltbewohner 1 in x-Richtung mit dem Wertebereich [-1024, 1023] (siehe [Bothe u. a. (2009)]),
- inputMwi1 der prozentuale Anteil des Einflusses mit dem Wertebereich [0,1], den Mikroweltbewohner 1 auf die Steuerung hat[21] und
- der Nenner des Bruchs der konstante Dämpfungsfaktor für den horizontalen Steuerinput (siehe [Bothe u. a. (2009)]).

Letztgenannter Dämpfungsfaktor hat historische Ursprünge. Er wurde für SAM 1.0 festgelegt und seit dem nicht geändert. Seine Wirkung besteht in einer Verringerung der Lenkempfindlichkeit, was das seitliche Steuern des Trackingobjekts vereinfacht. Dieser Faktor darf auch nicht geändert werden, damit die Vergleichbarkeit mit vorangegangenen Versuchen gewährleistet bleibt.

Das y-Delta wird mit der Formel

$$\frac{(((1024 - joy1YAxis) \cdot inputMwi1) + ((1024 - joy2YAxis) \cdot inputMwi2))}{100}$$

berechnet. Analog zur Berechnung des x-Deltas werden auch hier die Bezeichnungen verwendet. Der Dämpfungsfaktor beträgt in diesem Fall $\frac{1}{100}$. Auch dieser Wert hat seinen Ursprung in SAM 1.0.

[21] für Mikroweltbewohner 2: $inputMwi2 = 1 - inputMwi1$

Da auch für die Joystick-y-Achse ein Wertebereich von [-1024, 1023] vorgesehen ist (wobei der erste Wert die maximale Auslenkung des Joysticks nach vorne und der zweite Wert die maximale Auslenkung nach hinten ist), würde das Trackingobjekt nie zum Stillstand kommen[22]. Setzt man den Wert für die maximale Auslenkung nach hinten in die Formel ein, erhält man:

$$
\begin{aligned}
&\frac{(((1024-1023)\cdot inputMwi1)+((1024-1023)\cdot inputMwi2))}{100}\\
=&\frac{1\cdot inputMwi1+1\cdot inputMwi2}{100}\\
=&\frac{1}{100}, \text{ da } inputMwi1+inputMwi2=1.
\end{aligned}
$$

Das würde bedeuten, dass die geringste Geschwindigkeit für das Trackingobjekt nicht 0 ist, sondern 0,01. Außerdem haben Messungen gezeigt, dass ein Joystick trotz maximaler rückwärtiger Auslenkung nie den Wert 0,01 erreicht, sondern immer nur Werte zwischen 0,02 und 0,024 liefert. Daher wird in SAM jede Geschwindigkeit, die kleiner als 0,025 ist, als Wunsch zum Stillstand interpretiert und auf 0 gesetzt (einem Quelltext-Kommentar der Klassenmethode SAMControllerInput processInputData entnommen).

Die Koordinaten des gesuchten Punkts ergeben sich aus der Addition der Koordinaten der IST-Position des Trackingobjekts mit den entsprechenden Deltas:

$$
\begin{aligned}
P_x &= IST_x + xDelta\\
P_y &= IST_y + yDelta
\end{aligned}
$$

Variante 2: Berechnung der Folgeposition mit neuen Input-Werten

Die Berechnung der Deltas erfolgt hierbei nicht – wie bei Variante 1 – mit Werten aus einem SAMState, sondern mit neuen Joystick-Input-Werten, die als Parameter an die Funktion übergeben werden.

Variante 3: Berechnung der Folgeposition mit neuen Input- und Einfluss-Werten

Auch die letzte Variante berechnet die Folgeposition auf ähnliche Weise, nur dass diese sowohl individuelle Joystick-Input- als auch Einfluss-Werte für die beiden Mikroweltbewohner als Parameter empfängt. Dies ermöglicht eine vollständig flexible Bestimmung der Folgeposition mit jeder beliebigen Kombination aus Joystick-Input- und Einfluss-Vorgaben.

[22] maximale Auslenkung nach vorne: Maximalgeschwindigkeit; maximale Auslenkung nach hinten: Stillstand; Rückwärtsfahren ist nicht möglich

4.5.3 Input-Berechnung für das Manövrieren zu einer bestimmten Position

Das Gegenstück zur Berechnung der Folgeposition ist die Ermittlung der Eingabe-Parameter (für Joysticks und Steuerungseinfluss), um von der IST- zur Folgeposition zu gelangen.

Es sei noch einmal darauf hingewiesen, dass der Steuereinfluss beider Mikroweltbewohner zusammen 100 Prozent beträgt. Daher müssen nicht zwei unterschiedliche Joystick-Eingaben berechnet werden, sondern es genügt eine einzige. Werden dann die alten Werte mit den neuen überschrieben, so wird das Trackingobjekt zur gewünschten Position manövriert. Als Beispiel für die Berechnung der Bewegung in x-Richtung, wenn der Einfluss von Mikroweltbewohner 1 30 Prozent und der von Mikroweltbewohner 2 70 Prozent beträgt, dient folgende Gleichung (analog für den y-Input):

$$0.3(joyInputX) + 0.7(joyInputX) = joyInputX$$

Die Klassenmethode mit der Signatur
AAFControlSupport
calcJoystickRawSamFrom: objectsCurrentPosition
to: point.

bestimmt zunächst aus dem anzusteuernden Punkt POINT und der IST-Position des Trackingobjekts OBJECTSCURRENTPOSITION die Deltas für die Bewegung. Dies wird erreicht, indem die IST-Koordinaten von den Koordinaten des anzusteuernden Punktes subtrahiert werden.

Dazu wurden die Geleichungen, die schon aus Unterabschnitt 4.5.2 bekannt sind, umgestellt, um den Joystick-Input zu erhalten:

Für die Auslenkung in x-Richtung:

$$\frac{(joy1XAxis \cdot inputMwi1) + (joy2XAxis \cdot inputMwi2))}{\text{DämpfungsfaktorX}} = DeltaX$$

Vereinfachung:

$$joy1XAxis = joy2XAxis \text{ und } inputMwi1 + inputMwi2 = 1$$

$$\Rightarrow \frac{joy12XAxis}{\text{DämpfungsfaktorX}} = DeltaX| \cdot Dmpufngsfaktor X$$

und man erhält:

$$joy12XAxis = xDelta \cdot \text{DämpfungsfaktorX}$$

Für die Auslenkung in y-Richtung:

$$\frac{(((1024 - joy1Y) \cdot inputMwi1) + ((1024 - joy2Y) \cdot inputMwi2))}{\text{DämpfungsfaktorY}} = DeltaY$$

mit der gleichen Vereinfachung:

$$\frac{(1024 - joy12YAxis)}{\text{DämpfungsfaktorY}} = DeltaY| \cdot \text{DämpfungsfaktorY}| - 1024| \cdot (-1)$$

und man erhält:

$$joy12YAxis = 1024 - (DeltaY \cdot \text{DämpfungsfaktorY}).$$

Damit ist die Berechnung bereits abgeschlossen.Das Ergebnis wird als Punkt

$$(joy12XAxis|joy12YAxis)$$

an den Aufrufer zurückgegeben.

4.5.4 Suche nach optimaler Folgeposition innerhalb eines Toleranzkorridors

Eine Hauptkomponente bildet ein Algorithmus, der dazu dient – von einer IST-Position innerhalb eines Toleranzkorridors ausgehend – eine optimale nächste Position zu finden, die wieder innerhalb des Toleranzkorridors liegt. Dabei wird die Joystickeingabe der Mikroweltbewohner mit berücksichtigt. Es werden also keine Positionen bestimmt, die nicht innerhalb der vom Mikroweltbewohner vorgegebenen Parameter liegen.

Ablauf des Algorithmus

Voraussetzung 1: Die nächste, von den Mikroweltbewohnern angesteuerte Position P_{next} liegt außerhalb des Toleranzkorridors. Zur Feststellung der Gültigkeit einer Position – damit ist gemeint: Liegt der Punkt „gültig“ innerhalb eines bestimmten Toleranzkorridors– wurde die Methode mit der Signatur

AAFBoundTools
checkValidityFor: aPoint
in: aDataDict

entwickelt. Sie implementiert das Subpixel-Verhalten von Squeak (siehe Abschnitt 4.4).

Voraussetzung 2: Die Suchrichtung ist festgestellt worden (siehe Abschnitt 5.2 - Leitplanken). Je nachdem, in welche Richtung die nächstgelegene Racingline liegt, muss gesucht werden.

Die Suche nach der optimalen Position innerhalb des Toleranzkorridors erfolgt in drei hierarchischen Modulen. Sie wird abgebrochen, wenn die Position gefunden wurde, das heißt, unter Umständen werden nicht alle Module für die Suche benötigt.

1) Suche auf y-Niveau
Die Suche nach der nächsten Position innerhalb des Toleranzkorridors beginnt damit, dass auf dem aktuell durch die Mikroweltbewohner vorgegebenen *y-Niveau* nach einem Schnittpunkt mit dem Toleranzkorridor gesucht wird. Dazu werden die obere und untere Schranke von P_{next} gebildet und die linken (bei Suchrichtung links) bzw. rechten (bei Suchrichtung rechts) Toleranzkorridor-Koordinaten geholt. Als nächstes wird die Gerade bestimmt, die durch die beiden aquirierten Koordinaten verläuft. Anschließend ermittelt der Algorithmus den Punkt auf der Gerade, der auf dem y-Niveau liegt (nach x umgestellte Geradengleichung liefert den fehlenden Wert). Nun wird überprüft, ob der ermittelte Punkt auch erreichbar ist. Wenn er innerhalb der maximalen Lenkreichweite nach links bzw rechts (*leftmost-x* beziehungsweise *rightmost-x*) liegt[23], ist die gesuchte nächste Position gefunden worden. Liegt er außerhalb[24], so wird NIL zurückgegeben und die Suche muss unterhalb des y-Niveaus auf leftmost-x beziehungsweise rightmost-x fortgesetzt werden.

2) Suche auf dem linken oder rechten Rand des Bewegungsspielraums
Ausgehend vom y-Niveau wird auf dem jeweiligen Rand abwärts nach der nächsten Position gesucht, bis der y-Wert der IST-Position des Trackingobjekts unterschritten wurde; hierbei wird zunächst der Subpixel-Bruchteil der y-Auslenkung nicht berücksichtigt. Dieser wird im dritten Teil der Suche behandelt, wenn diese Suche erfolglos sein sollte. Um jede Pixelzeile zu untersuchen wird das y-Niveau in jedem Schritt um 1 dekrementiert. Die untere und obere Schranke zur aktuellen Position wird neu gebildet und die jeweils zugehörigen Toleranzkorridor-Koordinaten geholt. Liegt die untere Schranke erstmals im Bewegungsspielraum, ist die Pixelreihe gefunden, in der der gesuchte Schnittpunkt mit dem Toleranzkorridor liegt. Nun wird – wie schon zuvor – wieder eine Gerade durch die beiden Toleranzkorridor-Koordinaten gebildet. Dieses Mal wird aber der gesuchte Punkt ermittelt[25], in dem entweder leftmost-x oder rightmost-x (je nach Suchrichtung) in die Geradengleichung eingesetzt wird, sodass der fehlende y-Wert berechnet werden kann. Kommt die Suche zu einem Ergebnis, liefert sie den entsprechenden Punkt zurück; andernfalls – wie schon in 1) – ein NIL.

3) Suche auf y-IST-Niveau
Wurde bei der vorherigen Rand-Suche keine Position ermittelt, so muss auf Höhe des y-IST-Niveaus danach gesucht werden. Diese Suche muss separat durchgeführt werden, da der Subpixel-Bruchteil[26] der y-Auslenkung zuvor nicht mit berücksichtigt wurde. Zunächst wird wieder eine untere und obere Schranke gebildet; dieses Mal dient als Grundlage der y-Wert der IST-Position. Auch werden wieder die zugehörigen Toleranzkorridor-Koordinaten ermittelt und wie zuvor eine Gerade durch die Koordinaten gebildet. Ebenfalls wie zuvor wird nun der y-Wert auf der Geraden an der leftmost-x beziehungsweise rightmost-x Stelle gesucht[27]; das Ergebnis ist der gesuchte finale y-Wert auf der jeweiligen Seite. Zusammen mit leftmost-x beziehungsweise rightmost-x bildet der gefundene y-Wert den Punkt, der als Rückgabeobjekt an die Hauptmethode übergeben wird.

[23] vgl. Abbildung 4.17, Stellen b_{left} und b_{right}
[24] vgl. Abbildung 4.17, Stellen a_{left} und a_{right}
[25] vgl. Abbildung 4.17, Stellen c_{left} und c_{right}
[26] dargestellt in Abbildung 4.17 am unteren Rand
[27] vgl. Abbildung 4.17, Stellen d_{left} und d_{right}

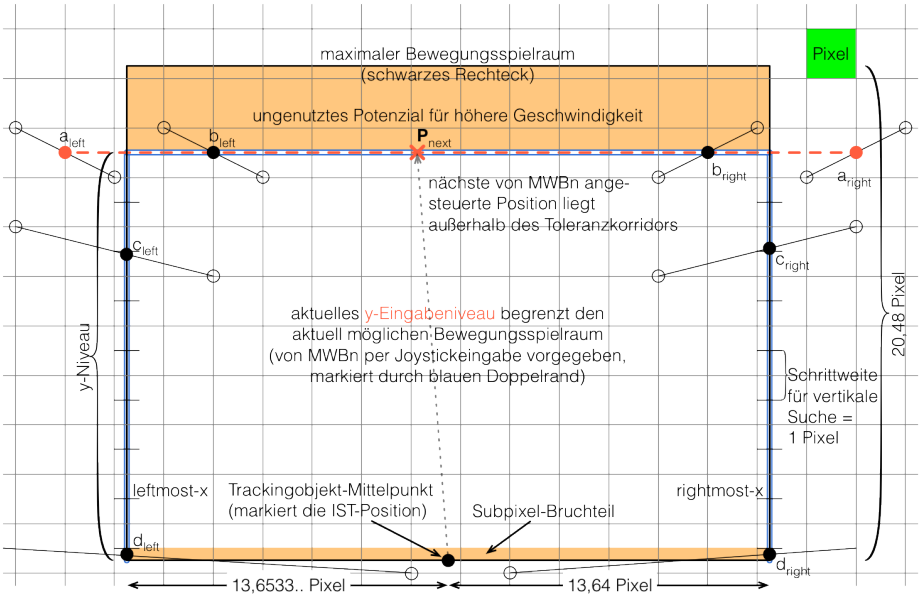


Abbildung 4.17: Die Suche nach einer Position innerhalb des Toleranzkorridor, wenn die von den Mikroweltbewohnern angesteuerte Position außerhalb liegt.

Die soeben vorgestellte Suche ist als Klassenmethode in der Klasse AAFBoundTools implementiert und besitzt folgende Signatur:

AAFBoundTools
discoverNextValidPointFrom: currentPosition
to: nextPoint
direction: aDirection
in: aDataDict

Hierbei ist

- **currentPosition** die IST-Position des Trackingobjekt-Mittelpunkts
- **nextPoint** die außerhalb des Toleranzkorridors liegende Position für den nächsten Tick, die korrigiert werden soll (siehe Voraussetzung 1)
- **aDirection** die Richtung, in die gesucht werden soll – gespeichert als Symbol #left beziehungsweise #right (siehe Voraussetzung 2)
- **aDataDict** der zugrunde liegende Toleranzkorridor (als Koordinatensammlung – siehe Abschnitt 4.2)

Zunächst werden die x-Werte der maximalen Lenkreichweiten leftmostX und rightmostX bestimmt:

$\text{leftmostX} = \text{currentPosition x} + (\frac{-1024}{1.5 \cdot 50})$

$\text{rightmostX} = \text{currentPosition x} + (\frac{1023}{1.5 \cdot 50})$

Die Besonderheit dabei ist, dass bei der Berechnung für rightmostX als maximaler Joystick-Ausschlag nur 1023 verwendet wird. Der Grund dafür ist, dass

der Ausschlag von 0 ebenfalls zu den gültigen Werten zählt, aber der Joystick nur 2048 verschiedene Werte liefern kann (1024 Werte für den Ausschlag nach links, 1023 Werte für den Ausschlag nach rechts und die Null-Position). Die Werte in den Klammern sind konstant: 13,64 für rechts und -13,6533̄ für links. Addiert man diese Werte zum x-Wert der gegenwärtigen Trackingobjekt-Position, so erhält man den links- beziehungsweise rechts-möglichsten x-Wert, den das Trackingobjekt ansteuern kann. Diese werden den einzelnen Such-Modulen als zusätzliche Parameter mit übergeben. Das Rückgabeobjekt der Methode ist der Punkt, der in einer der drei Such-Methoden ermittelt worden ist. Dieser Punkt

- liegt innerhalb des Toleranzkorridors und
- liegt von der IST-Position des Trackingobjekts aus innerhalb des Bewegungsspielraums für einen Tick.

Dadurch, dass immer nur unterhalb des y-Niveaus gesucht wird, werden die Mikroweltbewohner nicht bevormunded – etwa durch eine Geschwindigkeitsvergrößerung, indem im maximalen Bewegungsspielraum nach dem optimalen nächsten Punkt gesucht wird, ohne die y-Vorgabe der Mikroweltbewohner zu berücksichtigen.

Kapitel 5

Entwicklerkonzept-Spezifikationen

In diesem Kapitel werden die Automatikfunktionen aus den Entwicklerkonzepten vorgestellt, die implementiert wurden. Dabei wird nicht auf Implementierungsdetails eingegangen, sondern nur die algorithmische Seite betrachtet und mit Illustrationen anschaulich dargestellt. Es werden im einzelnen die Anforderungen, Muss- und Wunschkriterien an jede Automatikfunktion beschrieben und untersucht, ob bereits existierende Funktionen komplett oder teilweise für ihre Umsetzung wiederverwendet werden können. Dabei wird unter anderem auch auf das Kapitel 4 Bezug genommen, da bei der Umsetzung der einzelnen Entwicklerkonzepte die Resultate der zuvor erstellten Hilfsmittel zum Einsatz kommen.

5.1 Anforderungen und Restriktionen

Sowohl die Entwickler der Konzepte (siehe Kapitel 3) als auch die Diplomanden der Informatik, welche die Bearbeitung der Konzepte übernommen haben, mussten verschiedene Vorgaben bei ihrer Arbeit berücksichtigen.

Die wichtigste Anforderung gebot die Unveränderbarkeit des Ursprungssystems. Die vorliegende Version von SAM durfte lediglich erweitert werden. Der Grund dafür war die Erhaltung der Vergleichbarkeit von zukünftigen Versuchsergebnissen mit denen von vergangenen Experimenten.

Zudem sollte kein Konzept eine Vollautomatisierung vorsehen, bei der die Mikroweltbewohner vollständig und dauerhaft bevormundet oder ignoriert werden. Auch die Studenten der ATEO Arbeitsgruppe mussten dies bei den Umsetzungsarbeiten berücksichtigen.

Ebenso galt es die Konzepte so strikt wie möglich umzusetzen und deren Vorgaben so wenig wie möglich zu verändern. Davon ausgenommen wurden notwendige Ergänzungen und Korrekturen, damit etwaige Unvollständigkeiten oder Fehler nicht die eigentliche Funktion des Konzepts beeinträchtigen. Ebenfalls

zulässig waren zwingende Ergänzungen, um die Konzepte in SAM beziehungsweise dem AAF einzubinden. Allerdings durften diese Ergänzungen nicht soweit reichen, dass sie das Konzept erheblich verbesserten oder dessen ursprünglich erdachte Funktion alterierten. Außerdem ist es erforderlich, entsprechende zusätzliche Methoden für das AAF GUI (siehe Abschnitt 2.5) zu implementieren, damit die Anbindung der Konzepte an das AAF gewährleistet wurde.

5.2 Leitplanken

Wie bereits in Abschnitt 3.6 erwähnt, dienen *Leitplanken* dazu, das Verlassen eines vorgegebenen Toleranzkorridors zu verhindern. Innerhalb des Toleranzkorridors haben die Mikroweltbewohner volle Steuergewalt und können sich darin – mit den üblichen Einschränkungen (nur vorwärts, rechts und links) – frei bewegen. Aber sowie das Trackingobjekt mit den Eingabesignalen der Mikroweltbewohner den Toleranzkorridor verlassen würde, nimmt der Leitplanken-Agent eine Korrektur vor. Dabei wird das Trackingobjekt von virtuellen – also nicht sichtbaren, sondern nur spürbaren – Leitplanken innerhalb des Toleranzkorridors gehalten. Es wird mit der von den Mikroweltbewohnern vorgegebenen vertikalen Geschwindigkeit entlang der Leitplanken geführt. Auch wenn die Mikroweltbewohner in einer Linkskurve nach rechts lenken, so bewegt sich das Trackingobjekt nach links – eine gewisse vertikale Geschwindigkeit vorausgesetzt; ansonsten würde es am Rand der Leitplanke verharren, bis wieder ein vertikaler Input erfolgt. Diese Funktionsweise wird als *flüssig (smooth)* bezeichnet. Ein zweiter Modus dieses Agenten heißt *klebrig (sticky)*. Hierbei wird das Trackingobjekt nicht automatisch entlang der Leitplanken gelenkt. Erreicht das Trackingobjekt mit dem Input der Mikroweltbewohner eine Leitplanke, so bleibt das Trackingobjekt solange an dieser kleben, bis die Mikroweltbewohner eine Eingabe tätigen, die das Trackingobjekt an eine Position innerhalb des Toleranzkorridors steuert. Erst dann bewegt sich das Trackingobjekt wieder und die Fahrt kann fortgesetzt werden. Diese Version wurde als Ergänzung zu der zuvor beschriebenen konzipiert, um gegebenenfalls einen Agenten einsetzen zu können, bei dem die Mikroweltbewohner stärker dazu gezwungen werden, sich an den Fahrbahnverlauf zu halten. Die Implementierung dieses Modus wurde jedoch noch nicht auf die neue Methode umgestellt, mit der nun festgestellt wird, ob sich das Trackingobjekt außerhalb oder innerhalb eines bestimmten Toleranzkorridors befindet[1]. Da er von keinem Konzept vorgesehen und vorerst keine Verwendung finden würde, konzentrierten sich die Entwicklungsarbeiten mehr auf andere Themen (siehe auch [Wickert (2012)] - Fehlersuche beziehungsweise Leistungsoptimierung).

5.2.1 Algorithmus

Im Algorithmus des Leitplanken-Agenten werden einige der Werkzeuge verwendet, die bereits in Kapitel 4 vorgestellt wurden. Grundvoraussetzung für Leitplanken-Agenten ist die Existenz eines Toleranzkorridors (siehe Abschnitt 4.1). Mit zwei Instanzvariablen des Agenten werden die Eigenschaften des Toleranz-

[1] vergleiche Abschnitt 4.4

korridors festgelegt, der für die bevorstehende Fahrt verwendet werden soll[2]. Diese sind:

- **propSideRailsOnOrOffTrack** – bestimmt, ob der Toleranzkorridor auf oder neben der Fahrbahn liegen soll.
- **propSideRailsRadius** – legt den Abstand des Trackingobjekt-Mittelpunkts vom Fahrbahnrand fest.

Beispiel: Wird die erste Eigenschaft auf #OFFTRACK gesetzt und ein Radius von 15 (Pixeln) gewählt, so kann sich das Trackingobjekt gerade soweit von der Fahrbahn herunter bewegen, dass es gerade noch mit dessen äußerem Rand die Fahrbahn berührt.

Eine weitere Eigenschaft legt den Leitplanken-Modus fest:

- **propSideRailsType** – schaltet zwischen *klebrigen* und *flüssigen* Leitplanken um.

Bei der Initialisierung des Agenten werden alle Eigenschaften mit Standardwerten initialisiert und ein entsprechender Toleranzkorridor geladen, sofern nicht andere Einstellungen aus dem zugehörigen GUI (siehe Unterabschnitt 5.2.2) vorliegen. Als Standard-Toleranzkorridor wird #ONTRACK mit 15px Randabstand und als Standard-Modus *flüssig* eingestellt.

Ablauf des Algorithmus im Flüssig-Modus beginnt mit zwei Schritten, die in jedem Fall ausgeführt werden:

1. Bestimmung der SAM-relativen Position des Trackingobjekt-Mittelpunkts (AAFCONTROLSUPPORT CALCSAMRELATIVEOBJECTPOS)
2. Überprüfung, ob diese Position innerhalb oder außerhalb des Toleranzkorridors liegt

Liegt die bestimmte Position *außerhalb* des Toleranzkorridors (dieser Fall tritt nur ein, wenn eine Kollision mit einem Hindernis stattgefunden hat und das Trackingobjekt neben die Fahrbahn strafversetzt wurde), so können die Mikroweltbewohner das Trackingobjekt so lange „querfeldein“ steuern, bis sie es zurück in den Toleranzkorridor manövrieren. Dann erst wird der Agent wieder tätig. Denkbar nach einer Strafversetzung wären auch automatische Rückführungen zur Strecke. Da die Konzepte aber derartige Funktionen nicht enthielten, würde dies eine Konzept-Verbesserung herbei führen, die aber durch die einzuhaltenden Restriktionen (siehe Abschnitt 5.1) nicht erlaubt ist. Daher wird in dieser Situation ohne bestimmte Vorgaben reagiert und den Mikroweltbewohner gestattet, das Trackingobjekt nach dem SAM-Standard zu steuern, bis es wieder den Toleranzkorridor erreicht hat.

Wurde bei der Überprüfung festgestellt, dass sich das Trackingobjekt *innerhalb* des Toleranzkorridors befindet, dann werden folgende Schritte ausgeführt:

[2] vorausgesetzt es hat eine Streckenanalyse für die gesetzten Parameter stattgefunden und die entsprechende Datenklasse steht zur Verfügung

3. Bestimmung der Position, die das Trackingobjekt mit den aktuellen Eingaben der Mikroweltbewohner ansteuert (AAFControlSupport getNextTickPoint: ... for: ...).
4. Überprüfung, ob die ermittelte Position innerhalb des Toleranzkorridors liegt:

4.1 Wenn sie innerhalb liegt, werden keine weiteren Schritte notwendig; das SAMState-Objekt kann unverändert vom Agenten an den Aufrufer zurückgegeben werden.

4.2 liegt sie außerhalb, so ist ein Eingriff des Agenten notwendig.

4.2.1 Aktuelle Racingline -Daten werden für das y-Niveau der Folgeposition geholt.

4.2.2 Handelt es sich um eine Gabelung, so wird ein Array zurück geliefert und die neue Folgeposition wird anhand der Methode
determinePoint: objectsNextPosition
withArrayOfXValues: xValue
bestimmt.

4.2.3 Wird ein einzelner Wert zurück geliefert, so findet die Bestimmung mit Hilfe der Methode
determinePoint: objectsNextPosition
withXValue: xValue
statt.

In den folgenden Schritten wird zunächst die Suchrichtung bestimmt. Von der (ungültigen) Folgeposition ausgehend, wird immer in Richtung des nächstliegenden Racingline -Zweiges gesucht. Im zweiten Fall, in dem nur ein einzelner Wert zurück geliefert wird, ist diese Bestimmung relativ einfach, da nur zwei Möglichkeiten existieren: Entweder die ungültige Folgeposition befindet sich links oder rechts von der Racingline. Innerhalb einer Gabelung, in der zwei Zweige vorhanden sind, wird durch einfache Größenvergleiche die Suchrichtung ermittelt:

- x-Wert der Folgeposition liegt links vom linken Zweig: Suchrichtung #right, da die Racingline rechts von der Position liegt.
- x-Wert der Folgeposition liegt rechts vom rechten Zweig: Suchrichtung #left, da die Racingline links von der Position liegt.
- x-Wert der Folgeposition liegt zwischen linkem und rechtem Zweig, der Abstand zum linken ist aber geringer: Suchrichtung #left.
- x-Wert der Folgeposition liegt zwischen linkem und rechtem Zweig, der Abstand zum rechten ist aber geringer: Suchrichtung #right.

Wenn die Suchrichtung fest steht, wird mit dieser als Parameter die Suche nach der optimalen Folgeposition (beschrieben in Unterabschnitt 4.5.4) gestartet. Diese liefert den (vorläufig) nächsten anzusteuernden Punkt. Nun werden die Eingabewerte für x- und y-Auslenkung bestimmt (siehe Unterabschnitt 4.5.3). Da es sein kann, dass der ermittelte Punkt an einer Stelle liegt, für dessen Erreichen nur sehr geringe Eingabewerte benötigt werden, überprüft ein Test, ob die neuen Eingabewerte in der Dead-Zone liegen. Wenn sie außerhalb der Dead-

Zone liegen, werden die neuen Werte in den SAMState geschrieben und der Agent beendet für diesen Tick seine Arbeit. Andernfalls muss noch eine Korrektur stattfinden, indem nach einer gültigen Position (also einer Position innerhalb des Toleranzkorridors) gesucht wird, die mit minimalen Eingabewerte außerhalb der Dead-Zone – also +/- 30 in x- beziehungsweise y-Richtung – erreicht werden kann. Die verwendeten Begriffe *minimaler* beziehungsweise *maximaler vertikaler Eingabewert* meinen in diesem Kontext nicht die minimal beziehungsweise maximal möglichen Eingabewerte, sondern:

- *Minimaler vertikaler Eingabewert* = minimale vertikale Auslenkung des Joysticks, sodass gerade die Dead-Zone in Rückwärtsrichtung verlassen wurde – also +30 Einheiten; resultierende Geschwindigkeit: Etwas weniger als 50 %.
- *Maximaler vertikaler Eingabewert* = *minimale* vertikale Auslenkung des Joysticks, sodass gerade die Dead-Zone in Vorwärtsrichtung verlassen wurde – also -30 Einheiten; resultierende Geschwindigkeit: Etwas mehr als 50 %.

Folgende Fälle könnten auftreten:

- x- und y-Achsen-Eingabewerte befinden sich in der Dead-Zone.
- Nur die Eingabe in x-Richtung ist in der Dead-Zone.
- Nur die Eingabe in y-Richtung ist in der Dead-Zone.

Für den ersten Fall werden diese Tests durchgeführt:

a) Kann eine gültige Position mit minimalem horizontalen Eingabewert und originalem vertikalen Eingabewert (effektiv 0 Einheiten) erreicht werden? Wenn ja, dann wird das Trackingobjekt minimal mehr zur Racingline hin im Toleranzkorridor positioniert.

b) Kann eine gültige Position mit maximalem vertikalen Eingabewert und originalem horizontalen Eingabewert (effektiv 0 Einheiten) erreicht werden? Wenn ja, wird das Trackingobjekt minimal schneller im Toleranzkorridor positioniert.

c) Kann eine gültige Position mit minimalem horizontalen und maximalem vertikalen Eingabewert erreicht werden? Wenn ja, werden diese Werte verwendet.

d) Kann eine gültige Position mit minimalem vertikalen und originalem horizontalen Eingabewert erreicht werden? Wenn ja, wird die orignale x-Achsen-Eingabe und die minimale vertikale Eingabe verwendet.

e) War keiner der obigen Tests erfolgreich, bleibt nur noch eine Möglichkeit, das Trackingobjekt mit Eingabewerten zu steuern, die außerhalb der Dead-Zone liegen, und die es gleichzeitig innerhalb des Toleranzkorridors positionieren: Minimale horizontale und minimale vertikale Eingabe.

Ähnlich aber etwas einfacher sind die Tests für die anderen beiden Fälle. Ist der (absolute) Eingabe-x-Wert kleiner als 30 Einheiten, so wird getestet, ob die tatsächliche Folgeposition dennoch innerhalb des Toleranzkorridors liegt. Falls ja, ist die Arbeit des Agent hier beendet. Ansonsten endet sie damit, dass als x-Eingabe die minimale horizontale Auslenkung – also -30 für links und +30 für rechts – verwendet wird. Diese sorgt nur dafür, dass das Trackingobjekt ein kleines Stück dichter hin zur Racingline gelenkt wird.

Im dritten Fall wird ebenfalls zunächst getestet, ob die tatsächliche Folgeposition gültig ist, obwohl der (absolute) y-Eingabewert kleiner als 30 Einheiten ist. Wenn nicht, wird geprüft, ob mit der maximalen vertikalen Auslenkung eine gültige Position angesteuert werden kann. Wenn auch das nicht erfolgreich ist, wird als y-Eingabe die minimale vertikale Auslenkung verwendet.

Nach diesen Korrekturen ist sichergestellt, dass SAM mit den Werten, die nun im SAMState gespeichert sind, tatsächlich eine Position innerhalb des Toleranzkorridors angesteuert wird.

5.2.2 Leitplanken-GUI

Zur Konfiguration eines Leitplanken-Agenten wurde ein GUI implementiert. Darüber lässt sich der Modus[3] und der Toleranzkorridor einstellen. Abbildung 5.1 zeit das Aussehen der GUI im Konfigurationstool.

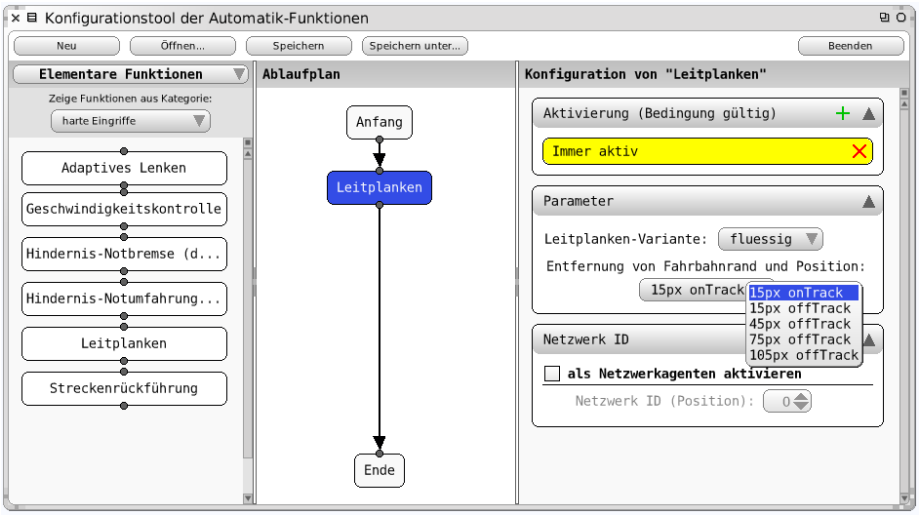


Abbildung 5.1: Konfiguration eines Leitplanken-Agenten.

[3] derzeit ist nur *flüssig* verfügbar

5.3 Rückführung

Die Aufgabe des Rückführungs-Agenten besteht darin, das Trackingobjekt auf optimalem Weg zurück in einen bestimmten Toleranzkorridor oder bis zur Racingline zu manövrieren. Da diese Funktionsklasse von mehreren Teams in leicht unterschiedlichen Versionen konzipiert wurde, sind drei Parameter identifiziert worden, mit denen all jene Konzepte abgedeckt werden können. Zunächst bedarf es eines *Aktivierungskorridors*. Da die Rückführung des Trackingobjekts in den einzelnen Konzepten an unterschiedlichen Stellen einsetzen soll, wurde über diesen Parameter die notwendige Flexibilität geschaffen. Gleiches gilt für den *Deaktivierungskorridor*. Bei ihm handelt es sich um die Grenze, an der der Rückführungsprozess endet. Außerdem ist es möglich, die Rückführung unterschiedlich stark in den SAM-Prozess einfließen zu lassen - von 0.0 (Prozent) bis zu 1.0 (= 100 Prozent).

In der Initialisierungsphase wird der Aktivierungskorridor geladen. Je nach verwendetem Deaktivierungsmodus wird zusätzlich der Deaktivierungskorridor oder die Racingline der verwendeten Strecke initialisiert. Außerdem wird eine Boolsche Instanz-Variable angelegt, die anzeigt, ob sich der Agent im Rückführungsmodus befindet oder nicht. Zu Beginn erhält diese Varialbe den Wert FALSE, da der Agent anfänglich deaktiviert sein soll.

5.3.1 Algorithmus

Die ersten Schritte des Agent bestehen darin, mit der wiederverwendbaren Klassenmethode **AAFControlSupport** calcSamRelativeObjectPos[4] die aktuelle Trackingobjekt-Position und die Folgeposition mit den originalen Eingabewerten der Mikroweltbewohner mit der Klassenmethode **AAFControlSupport getNextTickPoint:** samState **for:** objectsCurrentPosition.[5] Außerdem wird die Richtung festgestellt, in welcher der nächste Racingline -Zweig zu finden ist (**AAFControlSupport getDirectionToNearestRacingLineFrom:** objectsCurrentPosition). Das Ergebnis legt die Richtung fest, in welche die Rückführung erfolgen soll.

Um später die Eingabewerte der Mikroweltbewohner mit denen des Agenten kombinieren zu können, wird der gemeinsame Eingabewert der Mikroweltbewohner mit ihren Steuervorgaben bestimmt. Auch hierfür wird wieder eine AAFControlSupport-Klassenmethode wiederverwendet: **calcJoystickRawSamFrom:** objectsCurrentPosition **to:** ordinaryNextPosition[6]

Als nächstes wird überprüft, ob der Rückführungsagent bereits aktiv ist und aktuell eine Rückführung stattfindet. Ist dies *nicht* der Fall, so muss an dieser Stelle kontolliert werden, ob der Aktivierungskorridor im aktuellen Tick erreicht wurde und eine Rückführung eingeleitet werden muss. Liegt also die aktuelle Trackingobjekt-Position im Aktivierungskorridor, so wird der Rückführungsagent aktiviert, indem die Aktivitäts-Variable (siehe Initialisierung des Agenten) auf TRUE gesetzt wird.

[4] beschrieben in Unterabschnitt 4.5.1
[5] beschrieben in Unterabschnitt 4.5.2
[6] beschrieben in Unterabschnitt 4.5.3

Nun wird erneut geprüft, ob sich der Agent in einer Rückführung befindet (oder ob diese gestartet werden muss). Ist der Wert der Aktivitäts-Variablen nun TRUE, so wird – je nach Konfiguration des Rückführungsmodus – mit der entsprechenden Algorithmus-Variante fortgefahren.

Algorithmus-Variante: Mit Racingline

Wenn der Agent so konfiguriert wurde, dass er das Trackingobjekt bis zur Racingline zurück steuern soll, dann werden die Schritte dieser Variante durchgeführt. Für die optimale Route zurück zur Racingline steht in diesem Fall die Klasse **AAFOptimalStep** von Helmut Weidner-Kim zur Verfügung, die er in seiner Diplomarbeit [Weidner-Kim] ausführlich beschreibt. Für diesen Kontext genügt es zu wissen, dass nicht der direkte Weg mit „Vollgas“ und maximalem Lenkausschlag in Richtung Racingline auch der optimale Weg zurück zur Racingline ist oder sein muss. Die genannte Klasse stellt Instanz-Methoden für Gabelungen und einspurige Streckenabschnitte zur Verfügung, die jeweils die optimale Folgeposition von der aktuellen Trackingobjekt-Position aus zurückgeben.

Anhand der Folgeposition werden die optimalen Eingabewerte bestimmt (siehe Unterabschnitt 4.5.3). Diese werden mit den originalen Joystickeingaben der Mikroweltbewohner im konfigurierten Verhältnis (Einflusswert des Agenten) kombiniert. Erreicht wird dies über den Aufruf der in Unterabschnitt 4.5.2 beschriebenen AAFControlSupport-Klassenmethode

getNextTickPointInputForce1: (1 - propAgentInfluence)
inputForce2: propAgentInfluence
joy1: ordinaryRawInput
joy2: optimalRawInput
point: objectsCurrentPosition.

Nun werden mit der AAFControlSupport-Klassenmethode

getDirectionToNearestRacingLineFrom: aPoint

die *Richtungen* der (nächstliegenden) Racinglines für die optimale Folge- und die aktuelle Trackingobjekt-Position bestimmt. Diese Informationen sind von zentraler Bedeutung, da sie für die Feststellung notwendig sind, ob mit dem Steuerungsmanöver die Racingline gekreuzt wurde. Dabei wird für jede der beiden Positionen eine der folgenden Situationen ermittelt:

- SR → einspurige Racingline in Richtung rechts
  (**S**ingle-branch to the **R**ight)
- SL → einspurige Racingline in Richtung links
  (**S**ingle-branch to the **L**eft)
- LR → linker Zweig liegt rechts von der Position
  (**L**eft-branch to the **R**ight)
- LL → linker Zweig liegt links von der Position
  (**L**eft-branch to the **L**eft)
- RL → rechter Zweig liegt links von der Position
  (**R**ight-branch to the **L**eft)

- RR → rechter Zweig liegt rechts von der Position
  (**R**ight-branch to the **R**ight)

In Abbildung 5.2 sind alle Racingline-Übergänge dargestellt. Alle mit Rechtecken versehenen Übergänge stehen für die Fälle, in denen der Rückführungsagent seine Arbeit fortsetzen muss. In den übrigen Fällen ohne Rechteck wird die Racingline überquert, sodass die Abbruchbedingung „bis zur Racingline“ erfüllt ist und der Agent deaktiviert werden kann.

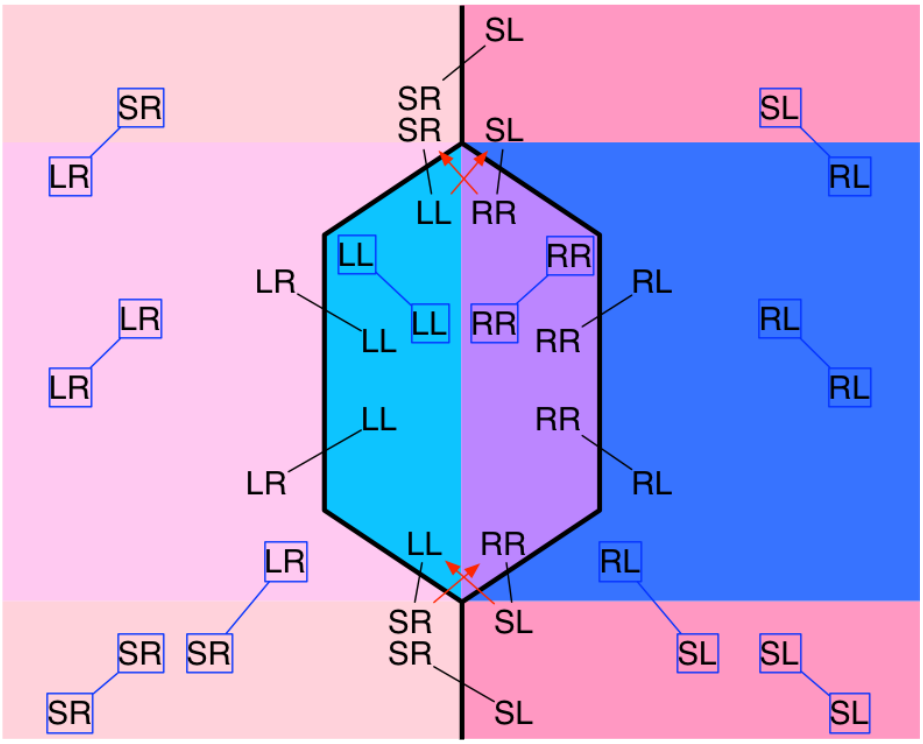


Abbildung 5.2: Alle möglichen Racingline-Übergänge. Die untere Position entspricht der aktuellen Trackingobjekt-Position und die obere der optimalen Folgeposition.

Algorithmus-Variante: Mit Toleranzkorridor

Optimal-Step ist hier nicht verwendbar, weil diese Methode nur bezüglich der Racingline-Rückführung korrekt arbeitet. Daher wird einfach das Steuern mit Höchstgeschwindigkeit und maximalem Lenkausschlag in Racingline-Richtung als optimal angenommen.

Zunächst wird die Richtung bestimmt, in die gefahren werden muss – bei einer Gabelung müssen gegebenenfalls die Abstände zu beiden Zweigen verglichen werden, wenn die aktuelle Position dazwischen liegt. Dann wird als optimale Joystickeingabe der jeweils maximale Lenk- und Geschwindigkeitsausschlag

gesetzt.

Nun werden die optimalen und die originalen Joystickeingaben im vorgegebenen Verhältnis – wie bereits im vorangegangenen Abschnitt beschrieben) miteinander kombiniert und die optimierte Folgeposition bestimmt.

Abschließend wird getestet, ob die optimierte Folgeposition innerhalb des Deaktivierungskorridors liegt. Wenn ja, dann ist der Eingriff des Agenten abgeschlossen und der Aktivitätswert kann auf FALSE zurückgesetzt werden. Andernfalls wird im folgenden Tick erneut vom Agenten versucht, das Trackingobjekt weiter in Richtung Deaktivierungskorridor zu steuern.

5.3.2 Rückführung-GUI

Die bereits eingangs dieses Abschnitts vorgestellten variablen Eigenschaften wurden für deren Konfiguration in einem individuellen GUI implementiert. Abbildung 5.3 zeigt ein Bildschirmfoto, auf dem alle einstellbaren Elemente mit ihren Standard-Werten zu sehen sind.

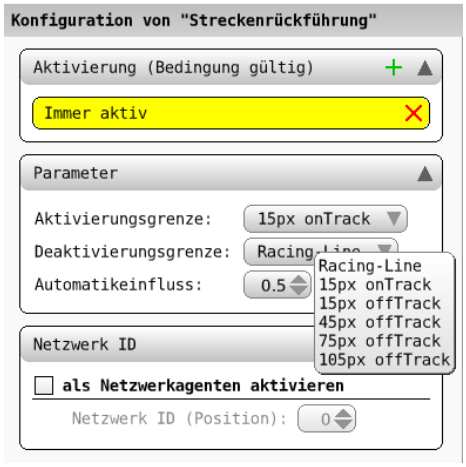


Abbildung 5.3: Über die dargestellten Elemente lässt sich der Rückführungsagent konfigurieren.

5.4 Geschwindigkeitskontrolle

Die Geschwindigkeitskontrolle soll dafür Sorge tragen, dass jeder Streckenabschnitt nur mit der jeweils zulässigen individuellen Höchstgeschwindigkeit durchfahren wird. Da jedoch einige Konzepte keine hundertprozentige Übernahme der Steuerung vorsehen, wurde ein Parameter eingebaut, mit dem man den Einfluss des Agenten von 0 bis 1.0 (= 100 Prozent) einstellen kann. Ebenso wurde die Möglichkeit zur Festlegung eines Toleranz-Intervalls geschaffen. Sie bietet dem Versuchsleiter die Option einen bestimmten Wert anzugeben, um den die aktuelle von der vorgegebenen Geschwindigkeit abweichen darf, bevor der Agent

eingreift. Das zugehörige GUI ist in Abbildung 5.4 dargestellt.

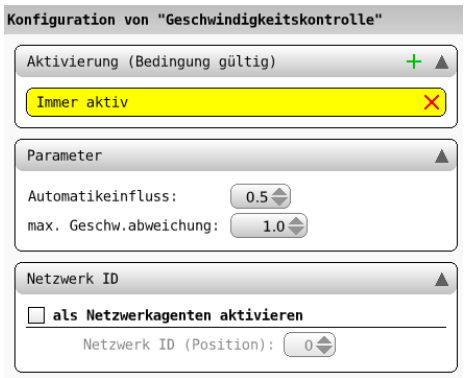


Abbildung 5.4: Dargestellt wird das GUI mit den beiden Parametern des Geschwindigkeitskontroll-Agenten mit deren Standard-Werten.

5.4.1 Algorithmus

Zunächst wird wieder die Methode zur IST-Positionsbestimmung[7] verwendet, um die aktuelle Position des Trackingobjekts in SAM-Relation zu erhalten.

Dann werden die von den realen Mikroweltbewohnern stammenden Eingabewerte zu einem kumulierten Wert zusammengefasst, genau so, als würde SAM die Joystick-Inputs verarbeiten (siehe Unterabschnitt 4.5.2).

Wenn festgestellt wird, dass sich die Geschwindigkeit des Trackingobjekts außerhalb des Toleranzbereichs befindet, wird zunächst die Folgeposition mit optimaler vertikaler Geschwindigkeit berechnet. Dazu wird die Methode von Aydan Seid (beschrieben in [Seid (2012a)])

AAFTrackOptimalSpeed calculateSpeedForY: yy
branch: bSymbol

aufgerufen. Sie liefert die Differenz der y-Werte von Folge- und IST-Position – also eine Geschwindigkeit. Damit lässt sich die optimale Folgeposition bestimmen und daraus wiederum die optimalen Eingabewerte, um das Trackingobjekt an diese Folgeposition zu steuern.

Da der Eingriff mit einer vorgegebenen Intensität erfolgen soll, wird als nächstes die Einflussverteilung zwischen Mikroweltbewohnern und Agent ermittelt. Danach wird die in Unterabschnitt 4.5.2 beschriebene AAFControlSupport-Klassenmethode mit der Signatur

getNextTickPointInputForce1: inputForceMWI1
inputForce2: inputForceMWI2
joy1: joystickRaw1
joy2: joystickRaw2
point: objectsPosOnTrack

[7] siehe Unterabschnitt 4.5.1

aufgerufen. Mit inputForceMWI1 und joystickRaw1 werden die kombinierten Eingaberesultate der beiden Mikroweltbewohner, und mit inputForceMWI2 und joystickRaw2 die zuvor bestimmten optimalen Eingabewerte des Agenten der Methode zur Bestimmung der endgültigen Folgeposition übergeben. Somit ist sichergestellt, dass schließlich die Eingaben der Mikroweltbewohner und die des Agenten im richtigen Verhältnis kombiniert werden. Das Trackingobjekt wird nun mit der eingestellten Intensität vom Agenten in die Richtung der optimalen Folgeposition bewegt.

5.5 Hindernisnotbremse

Die Aufgabe dieses Agenten ist, Kollisionen mit *dynamischen* Hindernissen zu verhindern. Um auch hier möglichst viele Konzepte abzudecken, die eine solche Notbremse vorsehen, wurden zwei Parameter für die Agenten-Konfiguration eingeführt. Zum einen sorgt die Typ-Auswahl für die Umschaltung zwischen zwei Modi: Eine Notbremsung mit absolutem Halt (kein Lenken möglich) und eine Notbremsung, bei der das Lenken auf horizontaler Ebene noch zugelassen wird, um den Gefahrenbereich beziehungsweise den Kollisionskurs schneller verlassen zu können. Zum anderen kann man über einen Entfernungswert die Distanz zum Hindernis festlegen, in der die Bremsung erfolgen soll. Der Wertebereich beginnt bei 15 (= Radius des Trackingobjekts, sodass unmittelbar vor dem Hindernis gebremst wird) und endet bei 800 Pixel (also dann, wenn das dynamische Hindernis auf dem Display erscheint. Das zugehörige GUI mit den Standard-Werten ist in Abbildung 5.5 zu sehen.

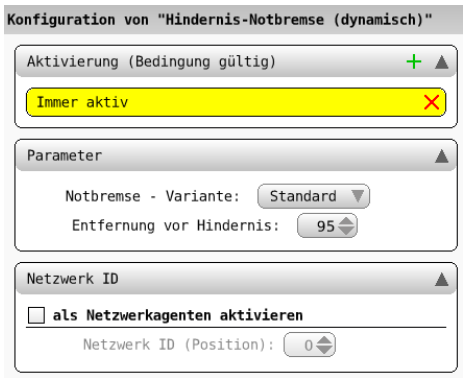


Abbildung 5.5: Das Konfigurations-GUI für die Notbremsung vor dynamischen Hindernissen.

5.5.1 Algorithmus

Der Ablauf des Standard-Agenten beginnt mit der Ermittlung der Ist-Position des Trackingobjekts unter Verwendung der Klassenmethode **AAFControlSupport** calcSamRelativeObjectPos.

Anschließend wird aus SAMs zentralem Datenobjekt mit der Methode **distanceDictionary at:** #nextDynamicObstacle die Entfernung zum nächsten dynamischen Hindernis geholt[8]. Ist diese Distanz kleiner, als die konfigurierte Entfernung zum Hindernis, dann befindet sich das Trackingobjekt in der Gefahrenzone.

Aber um festzustellen, ob tatsächlich eine Kollision droht, reicht diese Information alleine noch nicht aus. Es muss auch noch in Erfahrung gebracht werden, ob sich das Trackingobjekt auf unmittelbarem Kollisionskurs mit dem Hindernis befindet. Diese Frage beantwortet die Klassenmethode
collisionWithCurrentSpeedForState: samState
der Klasse
AAFSpeedHintsAtDynamicObstacles
von Nikolai Kosjar (beschrieben in [Kosjar (2012)]) mit einem Boolschen Wert.

Im letzten Schritt wird also geprüft, ob beide Bedingungen zutreffen: Das Trackingobjekt befindet sich in der Gefahrenzone und ist auf direktem Kollisionskurs. Und an dieser Stelle unterscheiden sich die beiden Modi. Für den Fall, dass der Stillstand-Modus (Standard) eingestellt ist, werden im SAMState schlicht die Joystick-Werte auf (0 @ 1023) gesetzt. Beim *Anti-Blockier-Modus* wird der originale x-Eingabewert statt der 0 verwendet, was dazu führt, dass das Trackingobjekt zwar noch auf der Horizontalen gelenkt werden kann, es sich dabei aber nicht vorwärts bewegt.

Mit der Rückgabe des SAMState an den Kontrollfluss ist die Arbeit des Agenten für diesen Tick getan. Nur wenn im nächsten Tick immer noch beide genannten Bedingungen erfüllt sind, verharrt das Trackingobjekt weiter an seiner Position beziehungsweise kann nur nach links oder rechts gesteuert werden. An sonsten wird die Fahrt unverändert fortgesetzt.

5.6 Hindernisumfahrung

Ein Agent dieser Klasse sorgt für die automatische Umfahrung von statischen Hindernissen. Er bedarf keiner weiteren Parameter und steuert das Trackingobjekt sicher an beliebigen statischen Hindernissen – also sowohl 25 % als auch 50 % Überdeckung – auf der Fahrbahn vorbei. Ähnlich wie bei den „flüssigen“ Leitplanken rutscht das Trackingobjekt an der unsichtbaren Notfallroute entlang, bis das Hindernis passiert wurde. Dabei sorgt die Route nur für eine horizontale Einschränkung; in vertikaler Richtung lässt sie stets die Maximalgeschwindigkeit zu. Da dieser Agent ohne zusätzliche Parameter auskommt, wurde in diesem Fall die Erstellung eines GUIs obsolet.

5.6.1 Initialisierungsphase

In der Initialisierungsphase des Agenten wird für alle vorgesehenen statischen Hindernisse des bevorstehenden Steps eine Notfallumfahrungsroute erstellt. Die-

[8] liefert der Aufruf nil zurück, so wird die Prozedur bereits an dieser Stelle abgebrochen, da keine weiteren dynamischen Hindernisse zwischen dem Trackingobjekt und der Ziellinie existieren

se besteht aus der *Notfall-Geraden (-gleichung)*, dem *Transit-Point* und dem *Point-of-no-Return*. Diese beiden Punkte (vergleiche auch Abbildung 5.6) lassen sich über zwei Parameter bestimmen: Der Hindernisreihenfolge (gelbe Zahlen in der Abbildung) und den entsprechenden x- beziehungsweise y-Werten der zugehörigen *Obstacle-No-Go-Area* (Ziffern 1 bis 4 in der Abbildung). Die Hindernis-No-Go-Area, das heißt die zugehörige Instanz-Methode
AAFStaticObstacleHelper
self avoidanceTipFor: anObstacle
liefert ein Array mit vier Werten[9]:

#1: linker x-Wert der No-Go-Area

#2: rechter x-Wert der No-Go-Area

#3: der untere (südliche) y-Wert der No-Go-Area

#4: der obere (nördliche) y-Wert der No-Go-Area

Beispiel: Der Point-of-No-Return von Hindernis 1 erhält als x-Wert das, was die Hindernis-No-Go-Area-Funktion an der Stelle 2 beinhaltet und als y-Wert das, was an Stelle 4 steht.

Die Steigungen der Notfall-Geraden ist bekannt: Sie errechnet sich aus der jeweiligen maximalen Lenkreichweite (in den folgenden Formeln jeweils im Nenner) und der Maximalgeschwindigkeit (jeweils im Zähler):

$$m_{links} = \frac{20.48}{13.64} = 1.5$$
$$m_{rechts} = \frac{20.48}{-13,6533} \approx -1.5$$

Der geringfügige Unterschied ergibt sich aus dem bereits mehrfach erwähnten Wertebereich eines Joysticks (-1024 bis +1023). Eine Notfall-Gerade verläuft durch den zugehörigen Transit-Point. Transit-Punkte markieren die Stellen, an denen das Trackingobjekt gefahrlos das Hindernis auf der Fahrbahnseite passieren kann, sofern weiterhin das Lenken in Hindernis-Richtung unterbunden wird. Mit Steigung und Punkt auf der Geraden lässt sich der y-Achsenabschnitt bestimmen (siehe Abschnitt 4.3). Damit sind alle Komponenten der Geraden ermittelt und ein AAFStraight-Objekt kann erzeugt werden.

Der Point-of-No-Return markiert die Stelle, ab der eine Kollision mit dem Hindernis ausgeschlossen ist. Das Trackingobjekt hat – wenn es diesen Punkt passiert – das Hindernis vollständig umfahren.

5.6.2 Algorithmus

Zunächst werden wieder die Methoden zur Bestimmung der aktuellen Trackingobjekt-Position (siehe Unterabschnitt 4.5.1) und der Folgeposition (siehe Unterabschnitt 4.5.2) wiederverwendet. Anschließend wird über alle vorhandenen Notfall-Routen iteriert und für jede die folgenden Schritte ausgeführt:

[9] Detaillierte Beschreibung dieser Methode ist in [Weidner-Kim] zu finden.

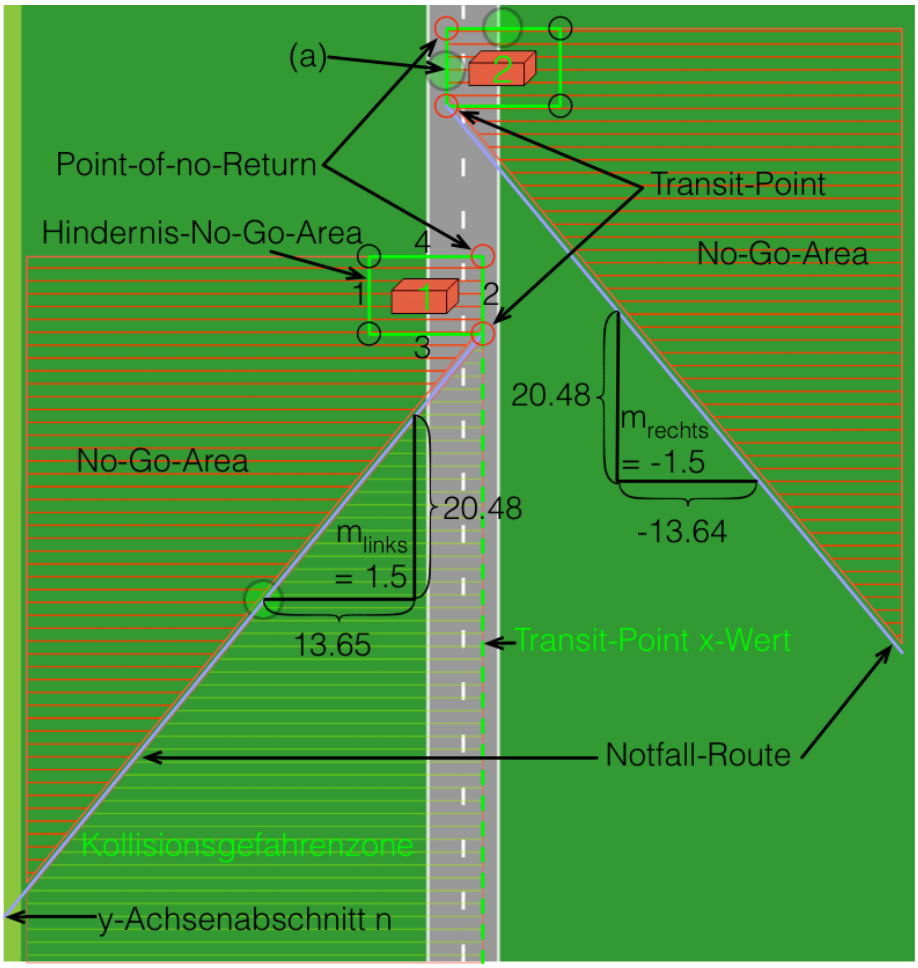


Abbildung 5.6: Dargestellt sind die Komponenten, die zur Umfahrung von statischen Hindernissen benötigt werden.
Hinweis: Die Einheiten der Steigungsdreiecke sind absichtlich nicht angegeben, da sie nicht maßstabsgetreu sind. Auch die Proportionen stimmen nicht ganz.

Als erstes wird geprüft, ob eine Notfall-Route überhaupt noch zu beachten ist, oder ob bereits ihr Point-of-no-Return passiert wurde und mit der nächsten Route fortgefahren werden kann (Schleife beginnt von vorne).

Wurde eine Route von Belang gefunden, dann wird geprüft, ob sich die Folgeposition bezüglich der Notfall-Geraden *oberhalb* befindet[10]. Wenn ja, dann liegt diese Position potentiell in der No-Go-Area. Daher muss der nächste Test entscheiden, ob eingegriffen werden muss, oder nicht. Andernfalls befindet sich die Folgeposition höchstens in der *Kollisionsgefahrenzone* (siehe Abbildung 5.6) aber nicht unmittelbar auf Kollisionskurs, sodass die Fahrt – vorerst – ohne Eingriff fortgesetzt werden kann.

Der Test überprüft, ob sich der Folgeposition-x-Wert (im Falle eines linken Hindernisses) links vom Transit-Point-x-Wert beziehungsweise (im Falle eines rechten Hindernisses) rechts davon und gleichzeitig zwischen dem y-Niveau des Transit-Points und dem des Point-of-no-Return befindet. Dann würde die Folgeposition innerhalb der No-Go-Area liegen und es muss eine horizontale Korrektur auf eben den x-Wert des Transit-Points erfolgen (vergleiche Stelle (a) in Abbildung 5.6).

Sollte der Test fehlgeschlagen sein, ist sichergestellt, dass das y-Niveau der Folgeposition unterhalb der y-Niveaus von Transit-Point und Point-of-no-Return liegt. Außerdem liegt die Folgeposition (wie bereits weiter oben festgestellt) oberhalb der Notfall-Geraden – es muss also wieder eine horizontale Korrektur erfolgen. Dieses Mal wird als neue Folgeposition ein Punkt auf der Notfall-Geraden auf dem y-Niveau der ursprünglichen Folgeposition bestimmt. Dies geschieht unter Verwendung der Instanz-Methode getPointOnStraightAtY: yValue des AAFStraight-Objektes der Notfall-Geraden.

Im letzten Schritt werden noch unter Wiederverwendung der Eingabe-Berechnungsmethode (siehe Unterabschnitt 4.5.3) die neuen Joystick-Auslenkungswerte bestimmt und anschließend im SAMState gespeichert. Dieser wird dann an den Aufrufer der compute-Methode zurück gegeben.

5.7 Situationsbedingte Einflussänderungen

Ein sehr spezieller und komplexer Agent ist der Einflussmanagement-Agent. Er bringt eine ganze Reihe Parameter mit sich, über welche sich der Einfluss von Mikroweltbewohner 1 auf sämtlichen Fahrbahnverläufen und auch bei bevorstehenden Hindernissen individuell festlegen lässt. Ein weiter Parameter legt die Anpassungsrate fest. Der Betrag der Anpassungsrate gibt an, um wie viel Prozent sich der Einfluss pro Tick ändern soll, wenn eine Differenz zwischen Ist- und Soll-Einfluss existiert. Die Standard-Konfiguration ist in Abbildung 5.7 dargestellt und entspricht den Vorgaben von Konzept 36.

Der Agent legt aber nicht nur die Einflussverteilung für Streckenabschnitte und Hindernisse fest. Er sorgt auch dafür, dass der Mikroweltbewohner mit der Prämisse *Genauigkeit* mehr Einfluss erhält, je weiter sich das Trackingobjekt von der Fahrbahnmitte – also der Racingline– entfernt. Das Konzept von Team 36,

[10] unter Verwendung der Methode zur Bestimmung der relativen Lage eines Punktes bezüglich einer Geraden (siehe Abschnitt 4.3)

Abbildung 5.7: Die verfügbaren Parameter, über die sich ein Einflussmanagement-Agent konfigurieren lässt.

welches mit diesem Agenten umgesetzt wurde, sieht eine stufenweise Umschaltung der Einflussverteilung vor. Da aber nicht genau spezifiziert wurde, wie groß diese Stufen sein sollen, wurde von der ATEO-Arbeitsgruppe beschlossen, den Durchmesser des Trackingobjekts als Grundlage zu verwenden. Das heißt, die erste Abstufung erfolgt, sowie das Trackingobjekt die Fahrbahn komplett verlassen hat und der Abstand zwischen Objektrand und Fahrbahnrand 0 Pixel beträgt; die zweite erfolgt, wenn der Objektrand 30 Pixel von der Fahrbahn entfernt ist (1 × Objektdurchmesser); die dritte erfolgt, wenn der Objektrand 60 Pixel von der Fahrbahn entfernt ist (2 × Objektdurchmesser); die vierte und letzte erfolgt, wenn der Objektrand 90 Pixel von der Fahrbahn entfernt ist (3 × Objektdurchmesser). Mit jeder Abstufung erhält der genauere Mikroweltbewohner 10 Prozent mehr Einfluss, sodass er schließlich mit der letzten Abstufung 90 Prozent Steuereinfluss erhält. Diese Abstufung wurde wieder mit Toleranzkorridoren realisiert. Jede Abstufung entspricht einem Toleranzkorridor. Über die vorhandenen Methoden lässt sich genau ermitteln, in welchem der verschiedenen Toleranzkorridore sich das Trackingobjekt aktuell befindet und welche Einflussverteilung demnach als Soll-Wert festgelegt werden müsste.

Nun wurden zwei möglicherweise unterschiedliche Einflussverteilungen ermittelt:

1. Die Verteilung für das aktuelle Strecken-Event (Gerade, Kurve, Gabelung, Hindernis).
2. Die Verteilung für die Trackingobjekt-Position bezüglich der Fahrbahn (auf der Fahrbahn, innerhalb von Toleranzkorridor 1, . . .).

Daher muss der Agent beide Verteilungen miteinander vergleichen, um dann die restriktivste Einflussverteilung als Soll-Wert in der zugehörigen Instanz-Variablen zu speichern.

Im letzten Schritt wird die Ist- (ebenfalls in einer Instanz-Variablen gespeichert) mit der Soll-Verteilung verglichen. Die Verteilung muss nur für einen Mikroweltbewohner angepasst werden, da sich der Wert für den zweiten automatisch daraus ergibt (siehe u. a. Unterabschnitt 4.5.3). Daher wird die Verteilung hier nur aus Sicht von Mikroweltbewohner 1 betrachtet. Wenn die Ist-Verteilung für Mikroweltbewohner 1 kleiner ist, als seine Soll-Verteilung, so wird sie mit der festgelegten Anpassungsrate erhöht; analog wird sie verringert, wenn sie größer ist, als die Soll-Verteilung. Dieser Schritt wird auch ausgeführt, wenn sich an der Soll-Verteilung nichts geändert hat, die Ist-Verteilung aber trotzdem noch nicht den richtigen Wert besitzt und vom Soll-Wert abweicht.

5.8 Adaptives Lenken

Dieser Agent schränkt die horizontale Lenkfähigkeit in Abhängigkeit von der vertikalen Geschwindigkeit ein. Um den meisten Konzepten gerecht zu werden, wurden hier mehrere Parameter zur Verfügung gestellt, die die Wirkungsweise sehr weitreichend beeinflussen können. Es gibt die Möglichkeit, eine Geschwindigkeit festzulegen, ab welcher der Agent beginnen soll, die Lenkung einzuschränken. Dieser Wert bezieht sich auf die *minimale* beziehungsweise *maximale Auslenkung des Joysticks in y-Richtung*. Daraus ergibt sich ein Wertebereich von -1024 (Höchstgeschwindigkeit) bis 1023 (Stillstand). In Abbildung 5.8 werden die Lenkeinschränkungen durch die rot-markierten, dreieckigen Flächen dargestellt. Es sind die Bereiche, die nicht mehr „verfügbar“ sind, wenn der Joystick über die vertikale untere Schranke hinaus bewegt wird.

Außerdem lassen sich separat voneinander die Lenkeinschränkungen für links bzw rechts einstellen. Dabei sind für links Werte von -1024 bis 0 und für rechts von 0 bis 1023 zugelassen. Somit ist ausgeschlossen, dass sich diese Grenzen überschneiden (linke Grenze liegt rechts von der rechten Grenze), wodurch undefiniertes Verhalten hervorgerufen werden könnte. Da in den Konzepten keine weiteren Angaben über die Art der Einschränkung gemacht wurden, erfolgt sie auf lineare Weise. Das zugehörige Konfigurations-GUI ist in Abbildung 5.9 als Bildschirmfoto dargestellt. Sie sieht vor, dass man auch andere Varianten wählen kann, als die zur Zeit einzig implementierte lineare. Außerdem lassen sich die maximale linke und die rechte Lenkeinschränkung individuell festlegen. Die Geschwindigkeit für die Aktivierung gibt an, ab welcher Joystick-Auslenkung in y-Richtung die Lenkung eingeschränkt werden soll.

5.8.1 Initialisierung

Das Besondere bei der Initialisierung eines Adaptive-Steering-Agenten ist die Erzeugung der beiden Dämpfungsgeraden. Die eine Gerade verläuft durch die Punkte (a) und (b), die andere durch (c) und (d) – wie auf Abbildung 5.8 dargestellt. Hierfür wird die Zwei-Punkte-Form der Geradengleichung – erläutert in Abschnitt 4.3 – wiederverwendet.

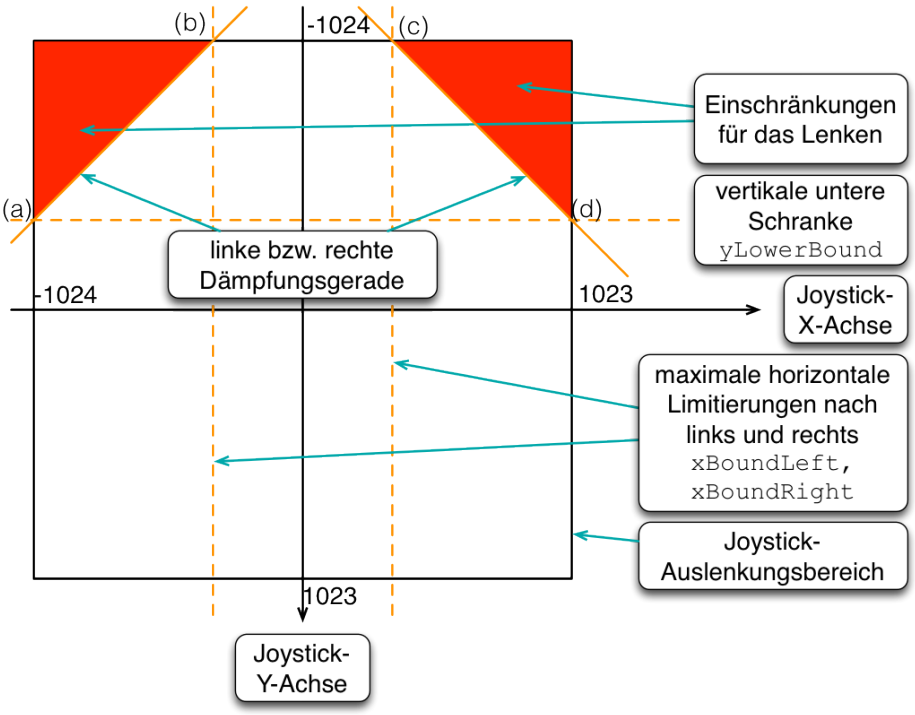


Abbildung 5.8: Die Aktivierungsgrenze (vertikale untere Schranke: yLowerBound) und die horizontalen Einschränkungswerte (xBoundLeft, xBoundRight) im Eingabebereich eines Joysticks.

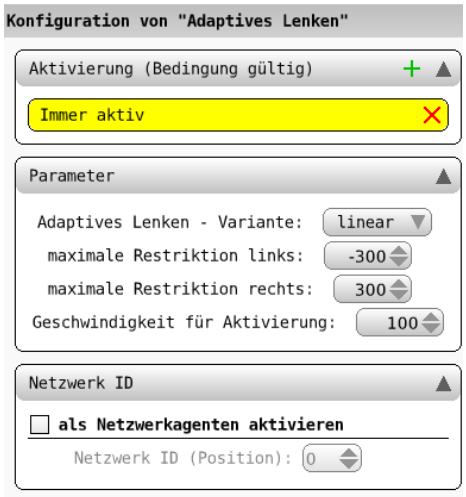


Abbildung 5.9: Die Konfigurations-GUI für adaptiv-lenkende Agenten.

5.8.2 Algorithmus

Im ersten Schritt werden die aktuellen Joystick-y-Eingaben aus dem SAMState extrahiert. Nun werden für beide Mikroweltbewohner separat die jeweiligen maximal zulässigen Lenkausschläge in horizontaler Richtung bestimmt, die bei der individuell anliegenden Geschwindigkeit (y-Auslenkung des Joysticks) zulässig sind. Hierbei kommt die herkömmliche Punkt-Steigungsform der Geradengleichung (ebenfalls in Abschnitt 4.3 beschrieben) zum Einsatz. Anschließend werden die Ergebnisse normalisiert, das heißt auf den maximal möglichen Lenkausschlag gesetzt, sollten die bestimmten Werte größer beziehungsweise kleiner als 1023 beziehungsweise-1024 sein.

Nach diesen Vorbereitungen erfolgt nun die Überprüfung, ob der Agent eingreifen muss. Dazu wird für beide Mikroweltbewohner separat verglichen, ob die jeweiligen x-Eingaben eine der folgenden Bedingungen erfüllen:

- Joystick-x-Wert ist kleiner als zulässiger maximaler Lenkausschlag nach links → Korrektur der x-Eingabe im SAMState auf maximal zulässigen Lenkausschlag nach links.
- Joystick-x-Wert ist größer als zulässiger maximaler Lenkausschlag nach rechts → Korrektur der x-Eingabe im SAMState auf maximal zulässigen Lenkausschlag nach rechts.

Wenn keine dieser Bedingungen erfüllt ist, lenken beide Mikroweltbewohner innerhalb der zulässigen Grenzwerte, sodass die Lenkung nicht durch den Agenten eingeschränkt werden muss.

Kapitel 6

Tests

Um die Korrektheit der Agenten nachweisen zu können, wurden Tests unter Zuhilfenahme des Test-Frameworks von Nikolai Kosjar (detailliert beschrieben in Kapitel 7 seiner Diplomarbeit [Kosjar (2012)]) durchgeführt.

Nachdem in Abschnitt 6.1 das Test-Framework und seine für diese Arbeit verwendeten Funktionen kurz erläutert wurden, werden in Abschnitt 6.2 die durchgeführten Agenten-Tests und die dadurch entdeckten Schwachstellen beschrieben.

6.1 Test-Framework

Das Test-Framework wurde von Nikolai Kosjar im Rahmen seiner Diplomarbeit [Kosjar (2012)] entwickelt. Es basiert auf dem Smalltalk-Framework für Unit-Tests *SUnits*[1]. Die drei wichtigsten Klassen, die für Agenten-Tests notwendig sind, lauten:

- AAFEventTestCase aus der Klassenbibliothek AAF,
- AAFTestAgent aus der Entwickler-Klassenbibliothek AAF-Agents-Dev, und
- AAFInputProviderAgent, ebenfalls der Entwickler-Klassenbibliothek zugehörig.

Die erstgenannte Klasse bildet die Schnittstelle zum SUnit-Framework. Das bedeutet, dass jeder Test-Agent (also jede Test-Klasse) von dieser Klasse erben muss, damit alle benötigten Test-Methoden (Schreiben, Konfigurieren und Ausführen von Tests) zur Verfügung stehen. Die wichtigsten Methoden zur Konfiguration sind

- configureStepAgentConfig - zur Konfiguration des zu testenden Agenten (Einstellung seiner Parameter)

[1] weitere Informationen dazu sind der Webseite http://sunit.sourceforge.net zu entnehmen

- configureStepEndAtDistance - an welchem y-Wert der Test abgebrochen werden soll
- configureStepMwiControl - 1, 2 oder 12, also ob nur Mikroweltbewohner 1, nur Mikroweltbewohner 2 oder beide zusammen für die Steuerung berücksichtigt werden sollen
- configureStepObstacleConfig - Hindernis-Konfiguration
- configureStepStartAtDistance - die Stelle in einem Step, an welcher der Test beginnen soll (zum Beispiel wenn eine explizite Stelle wie eine Gabelung getestet werden soll)
- configureStepTickDuration - eine individuelle Einstellung der Tick-Dauer, wenn 39 ms zu schnell oder zu langsam für eine bestimmte Stelle sind
- configureStepTrack - auf welcher Strecke getestet werden soll

Die Methode runStep startet den konfigurierten Step.

Die zweite Klasse stellt Ausgaben und erweiterte Parameter-Anzeigen für Tests zur Verfügung. Ein Test-Agent erhält aber auch in erster Linie zuvor konfigurierte Agenten, die es zu testen gilt – in der Regel nur einen einzelnen.

Die letzte Klasse ermöglicht sowohl manuelle Tests, sodass man wahlweise mit einer Maus oder den herkömmlichen Joysticks das Trackingobjekt steuern kann, als auch automatische Tests mit ganzen Sätzen vorgegebener Joystick-Eingabewerte. Letztere ermöglichen eine exakte Wiederholung eines Tests, sodass bestimmte Situationen immer mit den selben Voraussetzungen erneut herbeigeführt werden können. Dies ist besonders dann sinnvoll, wenn Fehler oder Fehlverhalten aufgetreten ist, und man diese Stelle besonders genau untersuchen möchte.

Wird ein vollständig konfigurierter Test gestartet, dann beginnt ein SAM Step ab der vorgegebenen Start-Position und endet an der entsprechenen End-Position (oder mit überfahren der Ziellinie, wenn diese Option nicht genutzt wird). Es wird also eine „ganz normale“ Fahrt simuliert, sodass man das Verhalten eines Agenten sehr gut in Echtzeit, Zeitraffer oder in Zeitlupe (je nachdem, wie die Tick-Dauer eingestellt wurde) beobachten kann.

6.2 Agenten-Tests

Da die Änderung der Eingabewerte eine zentrale Rolle bei den meisten Agenten spielt, wurden für die Tests drei spezielle Eingabewert-Serien generiert und als statische Methode im AAFInputProviderAgent zur Verfügung gestellt. Eine Serie enthält Eingabewerte, die für beide Mikroweltbewohner in jedem Tick konstant den Wert (0 @ 0) vorgibt, welcher der neutralen Joystick-Stellung entspricht. Die zweite Serie gibt für beide Mikroweltbewohner die maximale Joystick-Auslenkung nach vorne und neutrale seitliche Auslenkung vor, sodass das Trackingobjekt ohne Agenteneinfluss lediglich geradlinig und mit Höchstgeschwindigkeit vom Start zum Ziel manövriert wird. Die dritte Serie ist etwas umfangreicher geworden. Sie bietet eine lineare Steigerung der Geschwindigkeit

und alterierende maximale Lenkausschläge nach links und rechts. Diese Eingabewert-Serien genügen, um bei den entwickelten Agenten das gewünschte Verhalten beobachten zu können. Es gab zwei weitere Gründe, weshalb diese beiden Serien benötigt wurden. Zum einen setzt Squeak ohne angeschlossene Joysticks die Eingabewerte selbstständig auf (-1024 @ -1024). Das hat zur Folge, dass das Trackingobjekt immer mit Höchstgeschwindigkeit und maximalem Lenken nach links bewegt wird. Dadurch können Tests nicht ausreichend interpretiert werden; diverse Fehler konnten nur mit anderen Eingabewerten gefunden und beseitigt werden. Zum anderen entstand damit die Möglichkeit, einen Test in exakt gleicher Weise zu wiederholen. Ohne den InputProviderAgent wäre dies nicht möglich, da sonst immer manuell gesteuert werden müsste beziehungsweise nur die (-1024 @ -1024)-Eingabe als konstantestes Mittel zur Verfügung stünde.

Leitplanken-Tests

Für die Tests mit dem Leitplanken-Agenten kam die Eingabewert-Serie ohne Lenkausschlag aber mit Höchstgeschwindigkeit zum Einsatz. Dadurch konnten viele Schwachstellen gefunden und beseitigt werden. Unter anderem wurde auf diese Weise das Subpixelverhalten von Squeak untersucht (siehe Abschnitt 4.4) und daraufhin eine umfangreiche Überarbeitung der Leitplanken-Funktionsweise durchgeführt, um das Subpixelverhalten korrekt im Algorithmus einzubinden. In diesem Rahmen entstanden auch die Implementationen der *geometrischen Funktionen* (vergleiche Abschnitt 4.3. Weiterhin wurde in diesem Zusammenhang die *Suche nach der optimalen Folgeposition innerhalb eines Toleranzkorridors* entwickelt.

Einflussänderungs-Tests

Die Verwendung der Serie mit neutralen Eingabewerten ist insbesondere bei den Tests von Agenten mit Einflussänderung oder teilweiser Beteiligung eines Agenten an der Steuerung zum Einsatz gekommen. Hier konnte anhand der Kreisfüllung (welche die Trackingobjekt-Geschwindigkeit anzeigt) die korrekte Funktionsweise der jeweiligen Agenten beobachtet und nachgewiesen werden.

Tests für Rückführung, Geschwindigkeitskontrolle und Einflussmanagement

Die Tests für Einflussmanagement, Geschwindigkeitskontrolle und Rückführung wurden ebenfalls mit der neutralen Eingabewert-Serie durchgeführt. Zusätzlich wurde für die Rückführung und das Einflussmanagement die Strecke mit den jeweils verwendeten Toleranzkorridoren eingefärbt (vergleiche Abbildung 4.8), sodass – mit etwas verlangsamter Tick-Dauer – auch mit bloßem Auge erkennbar wurde, ob die jeweiligen Agenten an den richtigen Stellen gemäß den Vorgaben reagierten. Für den Geschwindigkeitskontroll-Agenten wurden zusätzlich die Streckensektionen sichtbar gemacht (vergleiche Abbildungen 6.5 und 6.6 auf den Seiten 67 und 68 in [Kosjar (2012)]). Dadurch konnte mitverfolgt werden, ob die Geschwindigkeit für die jeweiligen Sektionen vom Agenten angepasst wurde.

Adaptives Lenken - Tests

Für den adaptiv lenkenden Test-Agenten wurde die oben genannte dritte Serie mit Eingabewerten verwendet. Hier wurden mehrere Durchläufe mit unterschiedlichen Agent-Konfigurationen durchgeführt. Dabei wurden alle extremen Einstellungen getestet. Am Aussagekräftigsten war ein Test, bei dem die Lenk-

einschränkung bereits wirkte, sowie sich das Trackingobjekt in Vorwärtsbewegung setzte. Bei Höchstgeschwindigkeit herrschte in diesem Testfall dann eine Lenkeinschränkung von 100 Prozent – es war also nur noch die Fahrt geradeaus möglich. Hier wurde stichprobenartig an beliebigen Stellen der Ablauf unterbrochen und per Hand für die vorgegebenen Parameter (Geschwindigkeit, Lenkausschlag, Aktivierungsgrenze, Einschränkung links bzw. rechts) berechnet, welche resultierenden, effektiven Werte im Agenten ermittelt werden müssten. Diese Werte stimmten mit den tatsächlichen überein, womit der Agent als einsatzbereit markiert werden konnte (Variable READYTOUSE = TRUE).

Hindernisumfahrungs-Tests

Um die Hindernisumfahrung zu testen kamen sowohl die neutralen als auch die Hochgeschwindigkeits-Eingabewerte zum Einsatz. Außerdem wurden viele Testläufe mit manuellen Eingaben durchgeführt und zusätzlich eine speziell auf diesen Agenten zugeschnittene Serie, die abwechselnd die linke und rechte Notumfahrung testen sollte. Dazu wurden die drei folgenden Schritte wiederholt, bis die Hindernisse passiert waren:

1. 60 Ticks maximal nach links ohne Vorwärtsbewegung (entspricht einer Seitwärtsbewegung von etwas mehr als 800 Pixel, um sicher zu gehen, dass die Streckenseite gewechselt worden ist)
2. 1 Tick mit Höchstgeschwindigkeit geradeaus (y-Distanz $= 1 \times 20.48px$)
3. 60 Ticks maximal nach rechts ohne Vorwärtsbewegung

Auch hier wurde durch manuelle Überprüfung die korrekte Funktionsweise des Agenten nachgewiesen. Es fanden Kontrollen statt, ob die Positionen, an denen das Trackingobjekt zum Stillstand kam, den Vorgaben entsprachen und ob es sich bei Geradeausfahrt an der Notfall-Geraden entlang bewegte.

Fehlersuche und Ursachenforschung durch Tests

Erst die Einführung des Test-Frameworks brachte ganz neue Möglichkeiten mit sich, um Ungenauigkeiten und Fehler besser untersuchen zu können. Dadurch, dass nun der SAM Prozess durch eine Anpassung der Tick-Dauer verlangsamt werden konnte, sind Schwachstellen identifiziert worden. Es ließen sich diverse schwer auffindbare Fehler feststellen und korrigieren. Unter anderem der Effekt, dass ein Agent zwar die Folgeposition korrekt bestimmt hat, das Trackingobjekt jedoch zu einer ganz anderen Stelle manövriert wurde, konnte auf diese Weise entdeckt und beseitigt werden (siehe Unterabschnitte 4.5.2 und 5.2.1).

GUI-Tests

Für die Agenten, die über konfigurierbare Parameter verfügen, wurden Konfigurations-GUIs (Konfigurationsdialoge) implementiert. Diese müssen gewährleisten, dass die eingestellten Werte auch tatsächlich im Agenten gesetzt und – im Umkehrschluss – die im Agenten gesetzten Werte im Konfigurationsdialog korrekt angezeigt werden. Alle Dialoge und die zu übertragenden Parameter sind erfolgreich getestet worden.

Testergebnisse

Die Tests haben gezeigt, dass alle Agenten augenscheinlich und durch Stichprobenkontrolle der expliziten Werte von einem Tick zum nächsten fehlerfrei und

den Vorgaben entsprechend funktionieren und einsatzbereit sind. Das gleiche gilt für alle erstellten Konfigurationsdialoge, die in vollem Umfang ihre Aufgaben korrekt umsetzen.

6.3 Parameter-Empfehlungen für die Agentenkonfiguration

Die zuvor angesprochenen Parameter der jeweiligen Agenten lassen sich mitunter so einstellen, dass die Wirkung des Agenten darunter leiden kann oder die Mikroweltbewohner durch den ungünstig konfigurierten Agenten eine erhöhte Anstrengung erfahren. Daher sollen hier für alle konfigurierbaren Agenten Empfehlungen oder Richtlinien gegeben werden, inwieweit bestimmte Parameter auf die Wirkungsweise des Agenten Einfluss nehmen und welche Einstellungen sinnvoll erscheinen.

Leitplanken

Es wird empfohlen für diesen Agenten den Parameter *Entfernung vom Fahrbahnrand* auf den Wert 15px onTrack zu stellen. Diese Einstellung bewirkt, dass das Trackingobjekt die grau-weiße Fahrbahn nicht verlässt und somit der geringste Flächenfehler bei optimaler Geschwindigkeit entsteht.

Rückführung

Hierbei kommt es auf die Wünsche des Experimentleiters an. Je eher der Agent aktiviert wird (*Aktivierungsgrenze*) und zurück zur Fahrbahn, Racingline oder einem anderen Toleranzkorridor (*Deaktivierungsgrenze*) geführt wird, desto kleiner ist der entstehende Fehler. Außerdem sollte der *Einflusswert des Agenten* mehr als 0.5 (also mehr als die Hälfte des Gesamteinflusses) betragen, damit eine aktive Rückführung überhaupt gewährleistet wird. Beträgt der Wert weniger oder gleich 0.5, so ist bei gegensätzlichem Lenken der Mikroweltbewohner maximal ein Stillstand beziehungsweise Geradeausfahren möglich.

Geschwindigkeitskontrolle

Auch hier sollte der *Einfluss des Agenten* möglichst groß sein, damit eine entsprechende Wirkung erzielt wird. Die *Abweichung von der Soll-Geschwindigkeit* ist eher klein zu wählen, da durch große Werte die Aktivierung des Agenten erst spät oder gar nicht (wenn der Maximalwert von 20.48 gewählt wurde) eintritt. Es muss eine individuelle geeignete Einstellung gefunden werden. Tests haben gezeigt, dass Werte unter 2.0 dazu führen, dass der Agent nicht immer aktiv ist, aber in brenzligen Situationen auch die notwendigen Anpassungen vornimmt, wenn der Einfluss entsprechend hoch gewählt wurde.

Hindernisnotbremse

In den durchgeführten Tests haben sich Werte zwischen 20 und 100 Pixel für den *Abstand zum Hindernis* als geeignet erwiesen. Die Notbremse ist in den meisten Fällen nicht länger als ein bis drei Ticks aktiv. In dieser Zeit hat das Hindernis eine ausreichende Strecke zurückgelegt, dass das Trackingobjekt das Hindernis gefahrlos passieren kann. Werte jenseits der 100 wirken subjektiv so, als würde zu früh gebremst, da noch ausreichend Platz zum Hindernis ist,

um problemlos einen Kurs um das Hindernis herum zu wählen. Zudem ist die *Variante* „ABS“ besser geeignet, da sie noch ein horizontales Lenken zulässt, und somit die Möglichkeit besteht, das Trackingobjekt schneller vom Kollisionskurs zu manövrieren und das Hindernis noch frühzeitiger zu passieren.

Einflussmanagement oder situationsbedingte Einflussnahme

Mit den Parametern dieses Agenten wird der Einfluss des ersten Mikroweltbewohners für die jeweiligen Ereignisse geregelt. Hierbei ist es ratsam, die Einschränkungen mit zunehmendem Schwierigkeitsgrad des Ereignisses zu steigern, sodass eine Kontinuität entsteht. Die Standardeinstellung zeigt dies sehr schön. Mikroweltbewohner 1 hat die Prämisse möglichst schnell zu fahren. Daher soll ihm in Kurven und bei Hindernissen weniger Einfluss gegeben werden, um die Genauigkeit zu verbessern, was im Interesse von Mikroweltbewohner 2 ist. Daher wird Mikroweltbewohner 1 für

- 30er Kurven 40 Prozent Einfluss,
- 90er Kurven 30 Prozent Einfluss,
- 150er Kurven 20 Prozent Einfluss und bei
- 300er Kurven und Hindernissen 10 Prozent Einfluss

gewährt. Im Gegensatz dazu bekommt Mikroweltbewohner 1 auf Geraden 90 Prozent, da dies die Hochgeschwindigkeits-Stellen sind und dort besonders schnell gefahren werden kann.

Die *Anpassungsrate* kann zwar bis zu 1.0 groß werden, damit können aber sämtliche Veränderungen innerhalb eines Ticks vorgenommen werden. Das würde eine abrupte Einflussveränderung von *„100 Prozent für den ersten Mikroweltbewohner “* zu *„100 Prozent für den zweiten Mikroweltbewohner “* ermöglichen. Sinnvoll sind jedoch Einstellungen unter 0.1 (also maximal 10 Prozent je Tick) – dadurch wird ein fließender Übergang der Einflussänderung erreicht. Zu klein sollte die Rate aber auch nicht gewählt werden, da sonst gar nicht die zur Zeit gültige Verteilung erreicht wird, bevor schon die nächste Verteilung angestrebt werden muss. Somit wird empfohlen, den Parameter nicht unter 0.25 zu senken, da sonst die Anpassung zu langsam vonstattengeht.

Adaptives Lenken

Einstellbar sind hier die *maximalen Restriktionen nach links* beziehungsweise *rechts* und die *Geschwindigkeit für die Aktivierung*. In den meisten Fällen ist es sinnvoll, die Restriktionen nach links und rechts auf den gleichen Wert einzustellen. Für die Tests wurden verschiedene Einstellungen ausprobiert. Ein spürbarer Effekt trat ab etwa +/-950 ein. Je dichter die Werte an 0 heran kommen, desto stärker wird der Effekt. Werden beide Werte auf 0 gesetzt, ist ein Lenken bei maximaler vertikaler Geschwindigkeit nicht mehr möglich. Daher sollte abgewogen werden, wie viel man das Lenken einschränken möchte. Auch ab welcher Geschwindigkeit der Effekt eintreten soll, ist Geschmacksache. Hier noch einmal eine Erklärung der einstellbaren Werte: 1023 entspricht dem maximal nach hinten ausgelenkten Joystick; -1024 entspricht dem maximal nach vorne ausgelenkten Joystick (maximale vertikale Geschwindigkeit). Es macht also wenig Sinn, die Aktivitätsgrenze bei -950 bis -1024 anzusetzen, da der Effekt sonst –

aus Sicht der Mikroweltbewohner– als sehr abrupt empfunden wird beziehungsweise überhaupt nicht erst eintritt. Hier sollte eine Einstellung von maximal 80 (etwa -800) bis 90 Prozent (etwa -900) der Maximalgeschwindigkeit vorgenommen werden. Wird der Wert 1023 gewählt, beginnt die Einschränkung der Lenkung bereits ab der geringsten Vorwärtsbewegung.

Kapitel 7

Zusammenfassung, Diskussion und Ausblick

7.1 Zusammenfassung

Wie in Abschnitt 1.3 ausgeführt, hat diese Arbeit als Zielstellung den Anspruch erhoben, alle Entwicklerkonzepte, die einflussverändernde oder eingreifende Automatikfunktionen enthielten, zu realisieren. Von den Automatikfunktionen, die in Abschnitt 3.6 beschrieben wurden, konnten – mit Ausnahme der Einflussänderung *bei fehlerhaftem Verhalten* – alle Konzepte umgesetzt werden. Die genannte Ausnahme ist zwar generell umsetzbar und es lässt sich ein geeignetes Konzept für die Bewertung der Mikroweltbewohner-Steuerungsgüte finden, jedoch würden die Bearbeitung sehr zeitaufwändig und komplex werden. Außerdem lieferten die Konzepte der Entwicklerteams aus Wirtschaft und Forschung keine näheren Details zur Umsetzung, was eine komplette Neuentwicklung dieser Grundidee der Gütebestimmung nach sich gezogen hätte. Daher entschied man sich, diese Konzepte vorerst auszusparen, da diese Aufgabe den Rahmen einer Diplomarbeit sicher alleine füllen würde. Einige Konzepte wurden in dieser Arbeit nicht spezifiziert, da es sich dabei schlicht um Kombinationen aus den hier aufgeführten Agenten handelt (wie zum Beispiel das adaptive Lenken mit Toleranzkorridor-Bezug).

Die Konzeptanalyse in Kapitel 3 lieferte eine Reihe von Attributen, sodass viele Schnittmengen der einzelnen Konzepte erkannt wurden. Durch die Einführung von Parametern für die meisten Automatikfunktionen konnten gleich mehrere Konzepte gleichzeitig abgedeckt werden (zum Beispiel Leitplanken in unterschiedlichen Entfernungen zur Fahrbahn).

Von großem Vorteil für die Konzept-Umsetzungen erwies sich der modulare Aufbau wiederverwendbarer Software-Komponenten. Es ist gelungen, atomare Funktionen zu identifizieren, die häufig benötigt und durch die Auslagerung in Hilfsklassen an vielen Stellen eingesetzt werden konnten (siehe Kapitel 4). Besonders zu erwähnen ist die Lösung, die für das Subpixelverhalten von Squeak gefunden wurde. Erst durch diese wurde eine exakte Steuerung des Tracking-

objekts und eine exakte Positionsanalyse in den unterschiedlichsten Situationen möglich. Von ebenfalls großem Wert erwies sich die Streckenanalyse, die die notwendigen Arbeiten zur Ermittlung unterschiedlicher Toleranzkorridore erledigte. Die Toleranzkorridore kamen schließlich in vielen Konzepten zum Einsatz und erfüllten ihren Zweck in unterschiedlichen Kontexten.

In Kapitel 5 wurde der Versuch unternommen, die teils ungenauen und undetaillierten Konzept-Formulierungen in soweit zu spezifizieren, dass daraus funktionierende Agenten resultieren. Es wurden Klassen von Automatikfunktionen entwickelt, die über veränderbare Parameter verfügen, wodurch sie auf die jeweiligen Konzepte anpassbar wurden.

Die durchgeführten manuellen und automatischen Tests (siehe Kapitel 6) sind alle positiv abgeschlossen worden, sodass dem Einsatz der Agenten nichts widerspricht.

7.2 Diskussion und Ausblick

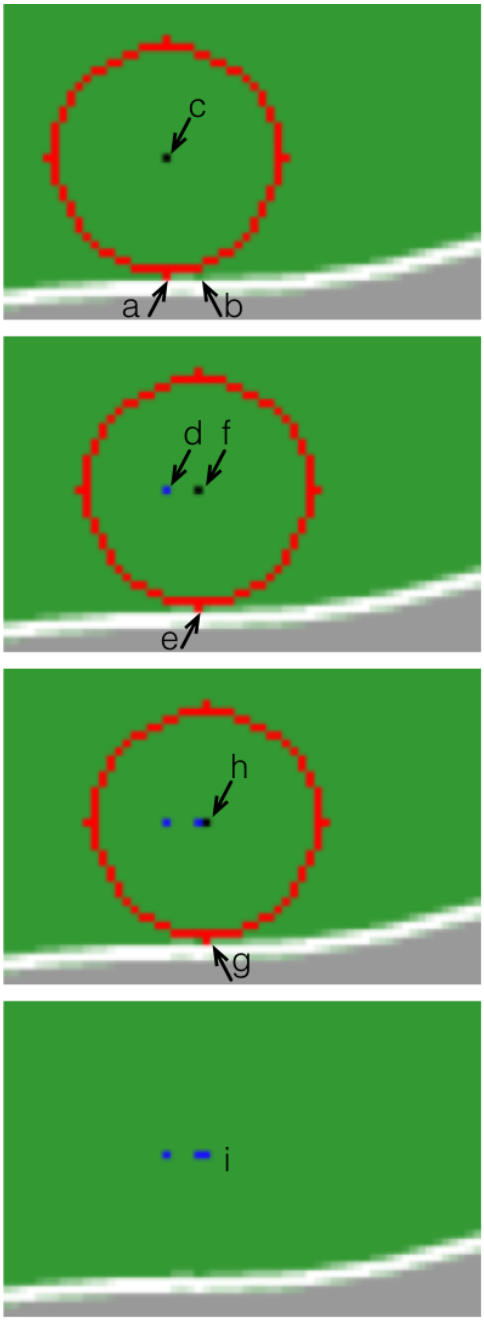
Unter anderem das Testen von Agenten und „Herumexperimentieren“ mit deren Parametern hat jedoch einige Probleme zu Tage befördert und es wurden verbesserungswürdige Stellen erkannt. Diese werden im Folgenden erläutert.

Streckenanalyse erzeugt ungewollte Artefakte

Die in Abschnitt 4.1 vorgestellte Methode zur Toleranzkorridor-Koordinaten Gewinnung erzeugte an ganz wenigen Stellen Artefakt-Koordinaten. Dabei handelte es sich um Koordinatenpaare, die entweder den selben Wert besaßen oder sich um 1 oder zwei Pixel unterschieden. Diese entstanden in der Nähe von 300er Kurven (siehe Übersicht der Kurvenarten in Abbildung 2.3 im Unterabschnitt 2.4.2). Die Untersuchung dieses Phänomens ergab, dass es sich dabei nicht um einen Fehler des Analyse-Werkzeugs handelte, sondern der unsaubere Rand der Fahrbahn dafür verantwortlich war. Der Suchkreis hatte an seinem südlichsten Punkt eine einzelne Koordinate – sein Rand bestand an dieser Stelle also nur aus einem Pixel. Mit diesem Pixel geriet der Suchkreis in eine Lücke von einem Pixel Breite im Fahrbahnrand, wodurch der Suchkreis nach Fahrbahnkontakt erstmals wieder komplett neben der Fahrbahn war. Dieses Ereignis erzeugte die erste Koordinate. Der Suchkreis wird ein Pixel weiter geschoben. Nun ist der südlichste Punkt auf dem Suchkreisrand wieder auf der Fahrbahn und die vorherige Mittelpunktkoordinate wird erneut gespeichert. Diese Artefakte wurden per Hand aus der Sammlung entfernt, da es mehr Zeit gekostet hätte, das Werkzeug so anzupassen, dass es auch diesen einen speziellen Fall erkennt und die Artefakte automatisch entfernt. Abbildung 7.1 zeigt den zuvor beschriebenen Ablauf.

Verbesserung des Streckenanalyse-Algorithmus

Derzeit verfolgt der Streckenanalyse-Algorithmus eine *brute-force-Strategie*, da die *gesamte* Umgebung jedes einzelnen Pixels untersucht wird. Es ist aber nicht notwendig, immer die komplette Umgebung abzusuchen. Je nach Analyse-Variante reicht es auch aus, nur die linke beziehungsweise rechte Umkreishälfte für die Erst-Erkennung des Fahrbahnrandes zu benutzen, und für die Erkennung der zweiten Koordinate die jeweils andere Hälfte. Da für die Analyse aber



a) Die südlichste Koordinate des Suchkreises stößt auf eine Fahrbahnfarbe.
b) Eine grüne Lücke in der Fahrbahn.
c) Die Position wird gespeichert.

d) Die korrekte Koordinate wurde gespeichert.
e) Die südlichste Koordinate stößt auf die Lücke.
f) Daher wird die falsche Koordinate gespeichert.

g) Die südlichste Suchkreiskoordinate ist wieder über Fahrbahnfarbe.
h) Daher wird die zweite falsche Koordinate gespeichert.

i) Drei Koordinaten wurden gefunden. Nur die ganz linke ist die gewünschte Grenze. Die anderen beiden sind Artefakte.

Abbildung 7.1: Zu sehen ist der Ablauf des Streckenanalyse-Werkzeugs, bei dem Artefakte erzeugt werden. Der Suchkreis wird von links nach rechts pixelweise weiter verschoben.

„unendlich“ viel Zeit zur Verfügung stand, weil sie außerhalb der Objektsteuerungsaufgabe durchgeführt wurde, ist auf eine Optimierung des Algorithmus auf diese Weise verzichtet worden.

Streckenanalyse kann mit Codegenerator gekoppelt werden

Der Vorgang der Streckenanalyse erzeugt zunächst eine CSV-Datei, die sämtliche Toleranzkorridor-Koordinaten enthält. Diese müssen dann mit dem Codegenerator (siehe Unterabschnitt 4.2.2) erst noch in die Datenstruktur eingebunden werden. Diese Lücke kann noch automatisiert werden, sodass der Codegenerator direkt ausgeführt wird, sowie eine Analyse beendet wurde. Außerdem lassen sich noch weitere Schritte, die bislang manuell auszuführen sind, mit in den Codegenerierungsprozess einbinden. Denkbar wäre unter anderem ein automatischer Import der erzeugten Klassendatei und das aktualisieren der Konfigurationsdialoge, die Toleranzkorridore zur Auswahl bieten – hier müsste eine Aktualisierung durchgeführt werden, damit auch der neu hinzugefügte Toleranzkorridor für diese Agenten verfügbar wird.

Verbesserung der Konfigurationsdialoge

Einige der erstellten Dialoge benutzen als Vorgabe originale Werte aus SAM. Das hat zwar den Vorteil, dass keine weitere Konvertierung der Werte für deren Benutzung notwendig ist, aber der große Nachteil ist, dass der Konfigurierende sehr viel implizites Wissen über SAM und teilweise auch über den zu konfigurierenden Agenten benötigt. Dem wurde zwar mit relativ aussagekräftigen Balloon-Tipps entgegengewirkt; aber diese geraten schnell an ihre Grenzen und drohen häufig mit Informationen überladen zu werden. Es könnten aber zusätzliche, visuelle Erklärungen oder Beispiele das Verständnis der jeweiligen Parameter verbessern und erleichtern. Noch geeigneter wäre die Verwendung von relativen Werten, etwa die Angabe von Lenkausschlägen oder Geschwindigkeiten in Prozent. Besonders beim adaptiven Lenken fällt dies auf; die Angabe der Werte in Prozent würde hier den Dialog deutlicher und viel leichter verständlich machen. Optimal wäre die Erstellung eines Screencasts oder eines Wizards, in dem erklärt wird, wie das jeweilige GUI zu bedienen ist und welche Auswirkungen die Veränderungen der jeweiligen Parameter haben.

Verbesserung des Konfigurationsdialogs für Rückführungsagenten

In der derzeitigen Version könnte die Deaktivierungsgrenze außerhalb der Aktivierungsgrenze liegen, was dazu führt, dass die Rückführung nicht beendet wird, weil die Deaktivierungsgrenze nie erreicht werden kann. Eine Plausibilitätsprüfung im Dialog würde hier Abhilfe schaffen.

Verbesserungen für den Leitplanken-Agenten

Sind Leitplanen aktiv und kollidiert das Trackingobjekt mit einem Hindernis, so wird es neben die Fahrbahn strafversetzt. Somit kann es sein, dass sich das Trackingobjekt nun außerhalb des eingestellten Toleranzkorridors befindet. Da kein Konzept diesen Fall berücksichtigt, wurde davon abgesehen, diese Situation gesondert zu behandeln. Daher kann das Trackingobjekt außerhalb des Toleranzkorridors von den Mikroweltbewohnern völlig frei bewegt werden. Erst wenn sie es zurück in den Toleranzkorridor steuern, „rastet“ es wieder zwischen den Leitplanken ein und kann diese nicht mehr verlassen. Es wäre aber denkbar, dass im Fall einer Hinderniskollision und der damit verbundenen Strafversetzung auch

eine Rückkehr in den Toleranzkorridor erzwungen wird. Die Art und Weise, wie diese Rückkehr stattfindet und mit welcher Intensität (Einfluss des Agenten), könnte dann über Parameter im GUI einstellbar sein. Hier müssten geeignete Konzepte erstellt und umgesetzt werden, was aber genauso wenig Ziel dieser Arbeit war wie die Lösung der Subpixel-Problematik für den zusätzlichen *klebrigen* Modus des Leitplanken-Agenten.

Einflussmanagement: Zusätzliche Parameter definieren

Der Einflussmanagement-Agent derzeit noch unkonfigurierbare Toleranzkorridore, die angeben, ab welchem Abstand zur Fahrbahn der Mikroweltbewohner mit der Genauigkeitsprämisse mehr Einfluss erhält. Die Korridore sind also gegenwärtig statisch und könnten in Zukunft variabel werden.

Automatikkonfiguration kann die Gesamtwirkung verfälschen

Eingabewert-verändernde Agenten können sich gegenseitig beeinflussen. Da eine parallele Verarbeitung in vielen Fällen nicht sinnvoll ist und sogar Fehlverhalten hervorrufen kann (siehe auch Kapitel 2.2, [Wickert (2012)]), wird in jedem Fall geraten, eingreifende Agenten seriell im Graph anzuordnen. Außerdem kann ungewolltes Verhalten hervorgerufen werden, indem der Konfigurierende eine Automatik entwirft, in der eingreifende Agenten sich gegenseitig beeinflussen, sodass die resultierende Automatik dadurch unbrauchbar wird. Voraussetzung für eine korrekte Funktionsweise von kombinierten Eingriffs-Agenten ist ein relativ genaues und detailliertes Verständnis der einzelnen Agenten und deren Wirkungsbereich. Nur wenn das Wissen vorhanden ist, auf welche Stellen in einem SAMState ein Agent zugreift, können die Agenten sinnvoll arrangiert und konfiguriert werden. Hier besteht für zukünftige Generationen von ATEO SAM noch Verbesserungspotential, indem bestimmte Prüfungen im AAFGT stattfinden und der Benutzer – eventuell über einen geeigneten Wizard – gewarnt wird, wenn mehrere der verwendeten Agenten auf die selben Ressourcen im SAMState zugreifen.

Literaturverzeichnis

[ATEO 2008] ATEO: *ATEO - ein Beitrag zum Graduiertenkolleg prometei*. 2008. – URL http://www.psychologie.hu-berlin.de/prof/ingpsy/forschung/ateo. – Zugriffsdatum: 2012-07-01

[Bothe u. a. 2009] Bothe, Klaus ; Hildebrandt, Michael ; Niestroj, Nicolas: *ATEO-System - Komponente SAMs*. Version 2.0/2.2. Institut für Informatik, Humboldt-Universität zu Berlin (Veranst.), 2009

[Bothe u. a. 2007–2013] Bothe, Klaus ; Wandke, Hartmut ; Hildebrandt, Michael ; Niestroj, Nicolas ; Meyer, Charlotte: *Mensch-Technik-Interaktion in Echtzeit - Projektseminar*. 2007-2013. – URL https://www2.informatik.hu-berlin.de/swt/lehre/MTI/seminars/. – Zugriffsdatum: 2011-11-26

[Fuhrmann 2010] Fuhrmann, Esther: *Entwicklung eines GUI für die Konfiguration der Software-Komponente zur Systemprozessüberwachung und -kontrolle in einer psychologischen Versuchsumgebung*, Humboldt-Universität zu Berlin, Diplomarbeit, 2010

[Goldberg und Robson 1983] Goldberg, Adele ; Robson, David: *Smalltalk-80: The Language and its Implementation*. BlueBook. Addison Wesley, 1983

[Gross und Sacklowski 2007] Gross, Barbara ; Sacklowski, Uli: *ATEO-System Nutzerdokumentation*. Version 1.1. Humboldt-Universität zu Berlin (Veranst.), 2007

[Hasselmann 2013] Hasselmann, Michael: *Titel unbekannt - Thema: ATEO Automation Framework*, Humboldt-Universität zu Berlin, Diplomarbeit in Vorbereitung, 2013

[Hildebrandt 2009] Hildebrandt, Michael: *Analyse und Evaluierung der Architektur des ATEO-Systems*, Institut für Informatik, Humboldt-Universität zu Berlin, Studienarbeit, 2009

[Kain] Kain, Saskia: *Entwickler in komplexen Mensch-Maschine-Systemen: Analyse des Einflusses von Entwicklerressourcen auf den Entwicklungsprozess und das -ergebnis*, Dissertation in Vorbereitung

[Kesselring 2009] Kesselring, Kai: *Entwicklung einer Softwarekomponente zur Systemprozessüberwachung und -kontrolle in einer psychologischen Versuchsumgebung*, Institut für Informatik, Humboldt-Universität zu Berlin, Diplomarbeit, 2009

[Kosjar 2011] Kosjar, Nikolai: *Die Gebrauchstauglichkeit des Automatiken-GUI im Projekt Arbeitsteilung Entwickler Operateur (ATEO)*, Humboldt-Universität zu Berlin, Studienarbeit, 2011

[Kosjar 2012] Kosjar, Nikolai: *Ein Ereignis-System für das ATEO Automation Framework sowie die Implementierung und Testung von auditiven und visuellen Hinweisen*, Humboldt-Universität zu Berlin, Diplomarbeit, 2012

[Nachtwei 2010] Nachtwei, Jens: *Design and Evaluation of a Supervisory Control Lab System for Automation Research - A Theoretical and Empirical Contribution to the Discussion on Function Allocation* -, Humboldt-Universität zu Berlin, Dissertation, 2010

[Niestroj 2008] Niestroj, Nicolas: *Erweiterung des ATEO-Systems zur Komplexitätserhöhung von SAM*, Humboldt-Universität zu Berlin, Studienarbeit, Mai 2008

[Niestroj 2009] Niestroj, Nicolas: *Vernetzung im ATEO Projekt aus inhaltlicher und technischer Sicht*, Humboldt-Universität zu Berlin, Diplomarbeit, 2009

[prometei 2006–2012] prometei, FSP8: *Graduiertenkolleg prometei; Forschungsschwerpunkt 8: Prof. Dr. Hartmut Wandke*. 2006-2012. – URL http://www.prometei.de/forschungsschwerpunkte/fsp-8.html. – Zugriffsdatum: 2012-11-11

[Schwarz 2009] Schwarz, Hermann: *Fenster zum Prozess: Ein Operateursarbeitsplatz zur Überwachung und Kontrolle von kooperativem Tracking*, Humboldt-Universität zu Berlin, Diplomarbeit, 2009

[Seid 2012a] Seid, Aydan: *Entwicklung und Konfiguration von visuellen Anzeigen im Projekt ArbeitsTeilung Entwickler-Oprateur (ATEO)*, Humboldt-Universität zu Berlin, Diplomarbeit, 2012

[Seid 2012b] Seid, Aydan: *Erweitern der Log-Datei und der Analyse von Log-Dateien im ATEO-Projekt*, Humboldt-Universität zu Berlin, Studienarbeit, 2012

[Sharp 1997] Sharp, Alec: *Smalltalk By Example*. 1997

[Sommerville 2011] Sommerville, Ian: *Software Engineering*. 9. Pearson, 2011

[Stritzinger 1999] Stritzinger, Alois: *Komponentenbasierte Softwareentwicklung*. Addison Wesley Longman Verlag GmbH, 1999

[Wandke und Nachtwei 2008] Wandke, Hartmut ; Nachtwei, Jens: The different Human Factor in Automation: The Developer behind versus the Operator in action. In: Waard, Dick de (Hrsg.) ; Flemisch, Frank (Hrsg.) ; Lorenz, Bernd (Hrsg.) ; Oberheid, Hendrik (Hrsg.) ; Brookhuis, Karel (Hrsg.): *Human Factors for assistance and automation*. Maastricht, the Netherlands: Shaker Publishing, 2008, S. 1—10

[Weidner-Kim] Weidner-Kim, Helmut: *Ermittlung des optimalen Weges für das Trackingobjekt in SAM zurück zur Streckenmitte*, Humboldt-Universität zu Berlin, Diplomarbeit in Vorbereitung

[Wickert 2012] Wickert, Andreas: *Fehler und deren Vermeidung bei der Programmierung und Konfiguration von Automatiken im ATEO-Projekt*, Humboldt-Universität zu Berlin, Studienarbeit, 2012

Anhang A

Schnittstellenspezifikation eines Agenten

Jeder Agent muss von der Klasse AAFAgent abgeleitet werden. Diese Super-Klasse gibt dann das Interface vor, welches in jedem Agenten individuell implementiert werden muss.

Ein Agent muss über bestimmte Instanz- und Klassen-Methoden verfügen, damit er korrekt in das AAF eingebunden werden kann und das Konfigurations-GUI die gewünschten Einstellung am Agenten speichert und anzeigt. Diese werden in den folgenden Abschnitten erläutert.

Um die Benutzung des Agenten mit dem Event-System (siehe [Kosjar (2012)]) zu ermöglichen, ist folgender Quelltext am Anfang der compute-Methode einzufügen:

(self event isValid: aSamState) ifFalse: [^aSamState].

Für das *Logging* sind umfangreichere Ergänzungen durchzuführen, die in der Studienarbeit [Seid (2012b)] nachzulesen sind.

A.1 Instanz-Methoden

- **initialize** – die Initialisierungs-Methode legt Standardwerte für die Agenten-Eigenschaften fest und initialisiert gegebenenfalls vorhandene Instanz- bzw. Klassenvariablen
- **Property-Getter und -Setter** – für jede Eigenschaft-Variable (property) muss es einen Getter beziehungsweise Setter geben.
- **setAllProps** – diese Methode überträgt die Informationen aus dem Agenten-GUI in die entsprechenden Eigenschaft-Variablen, indem sie die zugehörigen Setter mit den vorliegenden Daten aufruft.
- **getAllProps** – die Methode holt die eingestellten Eigenschaften aus den Instanz-Variablen und sendet sie an die Agenten-GUI, damit diese die aktuell konfigurierten Werte anzeigt.

- **compute** – die Compute-Methode ist das Kernstück des Agenten. Sie wird in jedem Durchlauf (Tick) von SAM aufgerufen, wenn die Agenten ausgeführt werden und beinhaltet die eigentliche Funktion des Agenten. Beim Aufruf erhält sie die Instanz eines SAMState und gibt diese (ggf. in veränderter Form) auch wieder zurück an den Aufrufer.

A.2 Klassen-Methoden

- **plaintextName** – diese Methode gibt einen String zurück, der den Namen des Agenten enthält. Dieser wird für die Anzeige (Benennung) im GUI verwendet.
- **readyForUse** – diese Methode gibt ein TRUE zurück, wenn der Agent fertig implementiert und einsatzbereit ist.
- **shortDescription** – der String, der von dieser Methode zurückgegeben wird, enthält eine kurze Beschreibung, die im GUI als Tooltip angezeigt wird.
- **tags** – gibt eine Menge von String-Symbolen zurück, die für die Kategorisierung im GUI dienen.

A.3 Einrichtung eines GUI

Ein Konfigurationsdialog muss von der Klasse AAFAgentDialog abgeleitet werden. Die Name eines Dialogs hat eine feste Syntax. Es ist zwingend erforderlich, dass der Dialog den selben Namen, wie der zugehörige Agent, trägt, an den zusätzlich das Postfix *Dialog* angehängt wird. Desweiteren müssen in einem Dialog Instanz-Variablen definiert werden, in denen die Eigenschaften des entsprechenden Agenten gespeichert werden können. Sinnvoll ist hier eine ähnliche Benennung, wie im Agenten.

Zudem sind zwei Instanz-Methoden erforderlich:

- **addSpecialElements** – hier werden die Anzeigeobjekte definiert und konfiguriert[1].
- **updateFromDelegate** – diese Methode sorgt für das korrekte Anzeigen der konfigurierten Eigenschaften des Agenten. Ohne sie würden nicht die im Agenten eingestellten Werte angezeigt werden.

Optional können weitere Hilfsmethoden – wie zum Beispiel userChose: aProperty, die beispielsweise die Übertragung der Einstellungen an den jeweiligen Agenten übernehmen könnte – implementiert werden, um eine bessere Übersichtlichkeit zu gewährleisten.

Die Einstellungen des Nutzers werden mit der geerbten Methode DELEGATE <propSetterName>: <value> an den jeweiligen Agenten übertragen.

[1] zur Auswahl stehende Objekte können über AAFGTWidgetGallery betrachtet werden

Anhang B

Die Klassen AAFTrackInfo, AAFTrackTools und AAFBoundTools

B.1 AAFTrackInfo Klasse

AAFTrackTools
loadTileNames

Diese Methode speichert alle verwendeten Kachelnamen in der Klassenvariablen vom Typ *OrderedCollection* ClassTileNames. Werden neue Kacheln hinzugefügt oder vorhandene entfernt, dann muss diese Methode angepasst werden.

Mit der Methode getTileNames wird diese OrderedCollection zurückgegeben.

AAFTrackTools
loadTileHeights

Die zweite Methode stellt die jeweiligen Höhen aller Kacheln in einem *Dictionary* zur Verfügung. Diese Höhen wurden hart-codiert, da sie sich nicht ändern – es wurde also auf eine dynamische Bestimmung der Kachelhöhen verzichtet. Dieses Dictionary kann mit der Methode getTileHeights geholt werden. Anwendung findet dieses in der Methode loadTrackDataFor: ... (siehe Unterabschnitt B.2.3), in der die Sammlung von Toleranzkorridor-Koordinaten für die gesamte aktuelle Strecke zusammen gestellt wird.

B.2 AAFTrackTools Klasse

In diesem Abschnitt werden die Unterstützungswerkzeuge vorgestellt, die die Klasse AAFTrackTools bereitstellt. Diese ist in der Kategorie AAF-Track enthalten.

Bei allen Methoden handelt es sich grundsätzlich um Klassenmethoden, die es

ermöglichen auch ohne eine Objektinstanz ausgeführt zu werden. Das hat den Vorteil, dass man für statische Aufrufe kein Objekt definieren und instanziieren muss, wodurch der ohnehin schon vorhandene hohe Speicherbedarf, der hauptsächlich durch große Bilddaten verursacht wird, nicht weiter erhöht wird.

Die Methoden sind zudem in Kategorien geordnet, die der Hauptwirkungsweise der jeweils enthaltenen Methoden entsprechen:

- AnalyzingTool: enthält die beiden Streckenanalyse-Varianten (siehe Abschnitt 4.1), zusätzliche Hilfsmethoden für den Analyseprozess und eine Stapel-Verarbeitungsmethode, die eine konfigurierte Analyse automatisch für alle Kacheln durchführt.
- CodeGenerator: Enthält den Codegenerator (beschrieben in Abschnitt 4.2.2).
- checker: Überprüfungsmethoden (siehe Anhang B.2.1)
- colorSupport: Methoden, die auf bestimmte Farben oder Farbbereiche testen, eine Farbanalyse auf allen Kacheln durchführt beziehungsweise die Farbcodes für spezielle Streckenbereiche bestimmt (siehe Anhang B.2.2).
- loader: Lademethoden für den klasseninternen und -externen Gebrauch (siehe Anhang B.2.3)
- painter: Färbemethoden (siehe Anhang B.2.4)
- pathinformation: Pfadinformationsmethoden, die Pfade zu bestimmten Ordnern liefern, in denen zum Beispiel Bild- oder CSV-Dateien enthalten sind.

B.2.1 Checker-Methoden

Diese Kategorie enthält drei Methoden, die sich in ihrer Wirkungsweise ähneln:
AAFTrackTools
checkSurroundingAt: point
offTrack: sourceImage
withRadialCoordinates: coordinates

Diese Methode sucht den Suchkreis (gegeben durch die coordinates-Sammlung) des aktuellen Punktes (point) auf einer vorgegebenen Kachel (sourceImage) nach Weiß- (isWhite; isWhitish) beziehungsweise Grauwerten (isGrey; isGreyish) ab. Wird unter einer der Koordinaten ein solcher Wert gefunden, gibt die Methode false zurück, da der Suchkreis Fahrbahn-Farbwerte enthält. Wird keiner dieser Werte gefunden, ist der Rückgabewert true.
AAFTrackTools
checkSurroundingAt: point
onTrack: sourceImage
withRadialCoordinates: coordinates

Diese Methode sucht den Suchkreis (gegeben durch die coordinates-Sammlung) des aktuellen Punktes (point) auf einer vorgegebenen Kachel (sourceImage) nach Grünwerten (isGreen) ab. Wird unter einer der Koordinaten ein solcher Wert gefunden, gibt die Methode FALSE zurück, da der Suchkreis „Gras"=Farbwerte enthält; er befindet sich also nicht mehr vollständig auf der Fahrbahn. Wird kein Grünwert gefunden, ist der Rückgabewert TRUE, da sich der Suchkreis noch vollständig auf der Fahrbahn befindet.

AAFTrackTools
checkSurroundingForObstacleAt: point
onTrack: sourceImage
withRadialCoordinates: coordinates

Diese Methode sucht den Suchkreis (gegeben durch die coordinates-Sammlung) des aktuellen Punktes (point) in einem Bild mit einem Hindernis (sourceImage) nach Farbwerten des Hindernisses (isBlackish; isRedish) ab. Wird unter einer der Koordinaten ein solcher Wert gefunden, gibt die Methode FALSE zurück, da der Suchkreis mit dem Hindernis überlappt. Andernfalls TRUE. Diese Methode wird für die Ermittlung der Koordinaten rund um ein Hindernis in einem zuvor definierten Abstand verwendet.

B.2.2 Color-Checker-Methoden

In dieser Kategorie befinden sich einige Methoden, die alle in irgendeiner Weise mit Farbwerten zu tun haben. Teilweise dienen sie nur speziellen Entwicklungszwecken, andere sollen feststellen, ob die als Parameter übergebene Farbe einem speziellen Bereich angehört (also die einzelnen Teilfarben Rot, Grün und Blau innerhalb eines vorgegebenen Toleranz-Intervalls liegen) oder exakt einer speziellen Farbe entspricht. In der Regel liefern sie boolsche Werte zurück. Es gibt aber auch einige Ausnahmen, die im Folgenden etwas detaillierter beschrieben werden.

Um eine Übersicht aller verwendeten Farben auf allen verwendeten Kacheln zu erhalten, wurde die Methode BUILDCOLORMAP entwickelt. Sie speichert alle verwendeten Farben und deren Namen in der CSV-Datei *colorMap.csv*. Außerdem nutzt sie die Methode RETRIEVECOLORNAME (siehe unten), um den Namen der aktuellen Farbe zu erhalten (oder *UNKNOWN*, wenn sie noch keinen Namen besitzt). Wurde eine unbekannte Farbe entdeckt, wird zusätzlich noch deren Position auf der jeweiligen Kachel mit in die Color-Map eingetragen.

FINDCOLORCODES dient der Feststellung, aller Farbcodes, die für die Erkennung bestimmter Streckenabschnitte (Gabelung, Kurvenart, etc.) verwendet werden und speichert diese in der CSV-Datei *colorCodes.csv*.

Die Methode RETRIEVECOLORNAME dient ausschließlich Entwicklungszwecken. Sie gibt den Namen der übergebenen Farbe oder den String 'UNKNOWN' zurück, wenn die Farbe nicht einem bestimmten Vorgabewert entspricht.

B.2.3 Loader-Methoden

AAFTrackTools

getSensoryArray

Diese Methode gibt ein Array mit Koordinaten zurück. Diese Koordinaten sind die Positionen der *Sensoren* des Trackingobjekts relativ zu dessen Mittelpunkt und werden über die Methode LOADSENSORARRAY (siehe unten) bereit gestellt.

AAFTrackTools
loadBoundsWithRadius: *aRadius*
onOrOffTrack: *aSymbol*

Diese Methode lädt *indirekt* die Toleranzkorridor-Koordinaten für die aktuelle Strecke. Sie ermittelt den aktuellen Streckennamen und ruft die Methode LOADTRACKDATAFOR: ... (siehe unten) auf.

AAFTrackTools
loadColorCodes

Die Aufgabe dieser Methode ist die Bereitstellung aller Farbcodes, die als Markierung für spezielle Streckenbereiche (Gabelung, Kurvenart, etc.) dienen. Sie werden als *OrderedCollection* abgelegt.

AAFTrackTools
loadCsvDataFor: *aTileName*
from: *aCsvFolder*

Die Methode wird ausschließlich von den Painter-Methoden (siehe Anhang B.2.4) verwendet und dient dazu, die von der Streckenanalyse ermittelten und in CSV-Dateien gespeicherten Koordinaten für die jeweiligen Painter bereit zu stellen.

AAFTrackTools
loadSensoryArray

Die Methode lädt alle Sensoren-Koordinaten in die Klassenvariable **CLASSSENSORARRAY** welche vom Typ *OrderedCollection* ist.

AAFTrackTools
loadToleratedColors

AAFTrackTools
loadTrackDataFor: *aTrackFileName*
onOrOffTrack: *aSymbol*
with: *aRadius*

Diese Methode erstellt das *Dictionary*, in welchem die Toleranzkorridor-Koordinaten der gesamten Strecke gespeichert werden und gibt dieses zurück.

B.2.4 Painter-Methoden

AAFTrackTools
paintCoordinatesFrom: *aCsvFileName*
in: *aCsvFolder*

Diese Methode markiert die in einer CSV-Datei gespeicherten Toleranzkorridor-Koordinaten auf der entsprechenden Kachel mit blauer Farbe und speichert die veränderte Kachel im vorgegebenen Ausgabeverzeichnis.

AAFTrackTools
paintCoordinatesFrom: *aCollection*
from: *aTileName*
to: *anOutputFolder*

Diese Methode erhält statt einer CSV-Datei die bereits in der Sammlung vorhandenen Toleranzkorridor-Koordinaten und markiert diese auf den entsprechenden Kacheln.

B.3 AAFBoundTools Klasse

Die Methoden der Klasse AAFBoundTools wurden auf drei Kategorien aufgeteilt:

- *access:* Methoden, um die Zugehörigkeit einer Position zu einem Toleranzkorridor zu prüfen (checkValidityFor:. . . , Abschnitt 4.4) oder die nächste gültige Folgeposition in einem Toleranzkorridor zu bestimmen (discoverNextValidPointFrom:. . . , siehe Unterabschnitt 4.5.4).
- *finder:* Methoden zur Bestimmung der nächsten links- beziehungsweise rechts-liegenden Toleranzkorridor-Koordinate:
  findBoundToTheLeftFrom: aPoint in: aDataDict
  beziehungsweise
  findBoundToTheRightFrom: aPoint in aDataDict.
- *supporter:* Methoden, die einen Übergang vom einen in einen anderen Gültigkeitsbereich feststellen (siehe Abschnitt 4.4).

Anhang C

Bedienungsanleitungen

C.1 Anleitung für das Streckenanalyse-Werkzeug

Zunächst muss festgelegt werden, ob die Analyse *„onTrack“* oder *„offTrack“* erfolgen soll. Dann muss entschieden werden, ob die Analyse nur auf einer Kachel oder für alle Kacheln auf einmal durchgeführt werden soll. Wenn die Entscheidung für *alle Kacheln* erfolgt ist, wird nur noch der *Radius des Suchkreises* erforderlich. Im Fall der Einzelanalyse muss zusätzlich der *Name der zu analysierenden Kachel*, der *Pfad zur Quellbilddatei* und der *Ausgabepfad* angegeben werden.

Gestartet wird jede Suche über einen Squeak-Workspace. Beispielhafte Kommandos könnten etwa so aussehen:

- Für eine Analyse aller Kacheln auf der Fahrbahn mit einem Radius von 15 Pixeln:

AAFTrackTools
processAllTilesWithRadius: 15
analyzer: #onTrack.

- Für die Analyse der Kachel *ELl.bmp* neben der Fahrbahn mit einem Radius von 75 Pixeln:

AAFTrackTools
locateBoundsOffTrack: 'ELl'
in: ''
to: 'outputDir'
with: 75.

C.2 Anleitung für den Codegenerator

Wie bereits in Unterabschnitt 4.2.2 beschrieben, entsteht bei der Streckenanalyse eine Menge von CSV-Dateien. Um diese in die Datenstruktur zu integrie-

ren, wurde der Codegenerator entwickelt. Er erstellt aus den CSV-Dateien eine Squeak-Klasse, die zunächst nur als *.st-Datei auf der Festplatte gespeichert wird. Dieser Vorgang wird mit folgender Codezeile in einem Squeak-Workspace angestoßen:

```
AAFTrackTools
    codeGeneratorForTrackData: <trackTypeSymbol>
    with: <aRadius>.
```

C.3 Anleitung zur Erweiterung der Datenstruktur

Wenn neue Datenklassen dem Bestand hinzugefügt werden sollen, sind aufgrund der extra dafür ausgelegten Datenstruktur nur wenige Handgriffe notwendig. Zunächst muss der Codegenerator (siehe Abschnitt 4.2.2) aus den neu gewonnenen Daten der Streckenanalyse Squeak-Klassen erzeugen. Diese müssen nun ins Squeak-Image eingebunden werden (zum Beispiel mit *filein*; sie werden automatisch in die Kategorie AAF-TrackData einsortiert). Anschließend müssen ein paar neue Zeilen Quelltext in den folgenden drei Methoden der Hauptklasse AAFTrackData eingefügt werden:

- die Methode loadBoundDataFor:in: ergänzen mit:
  ```
  switch at: '<newClassName>' put:
      [<newClassName> loadIndividualDataFor: aTileName.].
  ```
- die Methode loadClassNames ergänzen mit:
  ```
  add: '<newClassName>';
  ```
- die Methode loadCompleteBoundData ergänzen mit:
  ```
  switch at: '<newClassName>' put:
      [<newClassName> loadAllData.].
  ```

Damit ist das Einbinden der neuen Datenklasse abgeschlossen. Sie ist damit bereit für die Verwendung und kann auf Anfragen von außen ihre Daten bereitstellen.

Gegebenenfalls müssen auch die Konfigurations-GUIs aller Agenten angepasst beziehungsweise ergänzt werden, die die neue Datenklasse verwenden (können) sollen; wenigstens aber das GUI des Agenten, für den die Datenklasse bestimmt ist. Je nachdem, wie das GUI arbeitet, könnte so eine Stelle beispielsweise so aussehen (Beispiel aus AAFSideRailsDialog):

```
string = '105px offTrack'
    ifTrue:
        [
        delegate propSideRailsRadius: 105.
        delegate propSideRailsOnOrOffTrack: #offTrack.
        ].
```

Anhang D

Inhalt der DVD

1. Abbildungen
   grobe Skizzen, OmniGraffle-Projekt-Datei, zusätzliche Entwürfe
2. Diplomarbeit
   als PDF und das zugehörige LaTeX-Projekt
3. Dokumente
   Pseudo-Code und Grobentwürfe von Algorithmen
4. Quellcode
   Ausschließlich die in dieser Arbeit erzeugten Quellcodes
5. Studienarbeit
   als PDF und das zugehörige LaTeX-Projekt
6. Squeak
   Stand der Entwicklung inkl. Squeak-Image und aktueller Quellen
7. Tabellen
   u. a. Tabelle mit den Entwicklerkonzepten (auch mit Bewertung der Automatik-Funktionen)
8. Transkripte
   der Videomitschnitte bei der Konzeptentwicklung
9. Vortragsfolien
   einige Vorträge, die im Rahmen der Arbeit gehalten wurden

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit mit dem Titel *Klassifizierung und Entwicklung von Automatiken für Eingriffe in der Socially Augmented Microworld (SAM)* selbstständig und nur unter Zuhilfenahme der verwendeten und als solche kenntlich gemachten Quellen verfasst habe. Weiterhin erkläre ich, dass ich erstmalig eine Diplomarbeit aus dem Studiengebiet *Informatik* einreiche.

Berlin, 7. August 2013

Andreas Wickert

Ich bin damit einverstanden, dass die vorliegende Arbeit mit dem Titel *Klassifizierung und Entwicklung von Automatiken für Eingriffe in der Socially Augmented Microworld (SAM)* in der Bibliothek ausgelegt wird.

Berlin, 7. August 2013

Andreas Wickert