**Humboldt-Universität zu Berlin**

Institut für Informatik


# Erweiterung des ATEO-Systems zur Komplexitätserhöhung in SAM

Studienarbeit von Nicolas Niestroj

März 2008

Betreuer:

Prof. Dr. Klaus Bothe

Prof. Dr. Hartmut Wandke

Dipl.-Psych. Saskia Kain

Dipl.-Psych. Jens Nachtwei

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Kapitel 1

# Einleitung

Das ATEO-System ist ein Projekt der Psychologen von der Humboldt-Universität zu Berlin Institut für Psychologie, Lehrstuhl für Ingenieurpsychologie/Kognitive Ergonomie, Projekt ATEO und vom Graduiertenkolleg prometei (Prospektive Gestaltung von Mensch-Technik-Interaktion) an der Technischen Universität Berlin. ATEO steht für ArbeitsTeilung Entwickler-Operateur und untersucht die Funktionsteilung zwischen Mensch und Maschine. SAM ist die Socially Augmented Microworld, in der Versuchspersonen als Mikroweltbewohner eingesetzt werden. Hierdurch soll die Komplexität des Systems, welches ein Operateur oder eine Automatisierung steuern soll und ein Entwickler mit konzipierten Automatisierungen unterstützen soll, unvorhersehbar bleiben. Der Focus liegt also nur auf dem Operateur und dem Entwickler, die Mikroweltbewohner sind eine Komponente, die die Komplexität bestimmt. Das ATEO-Systems wurde bereits für eine erste Versuchsreihe konfiguriert und benutzt. Als eine Erkenntnis der ersten Versuchsreihe kann man sagen, dass das Konfliktpotenzial zwischen den Mikroweltbewohnern zu gering für einen Unterstützungsbedarf durch einen Operateur oder eine Automatisierung war. Die beiden haben die Strecke im Zuge des Experiments ohne größere Probleme abgefahren und durch die fehlende Komplexität der Strecke hatte auch der Operateur wenig Anlass unterstützend einzugreifen, d.h. die Komplexität des Systems war unzureichend. Der Entwickler ist direkt durch die niedrige Komplexität betroffen, da er wenig anspruchsvolle Systeme entwerfen muss, und das für ihn zur Routineaufgabe wird. Ein System, welches alleine fast fehlerfrei läuft, benötigt keine Assisstenzsysteme für die Mikroweltbewohner und niemanden, der steuernd eingreifen muss. Diese Konfliktsituationen für den Operateur und den Entwickler zu generieren, ist von zentraler Bedeutung für zukünftige Versuche. Zwar haben in der ersten Phase des Projektes der Operateur oder ein Assisstenzsystem keine Rolle gespielt, aber in Phase Zwei wird der Vergleich zwischen Operateur und durch Entwickler konzipierte Assisstenzsysteme angestrebt. Aus diesem Grund sollen einige Elemente zur Erhöhung der Komplexität hinzugefügt werden und der Einfluss dieser im Rahmen einer Voruntersuchung ermittelt werden. Im Einzelnen sind das statische und dynamische Hindernisse, ein Tachometer sowie die Möglichkeit, den Steuerungsinput der Mikroweltbewohner von 50-50 auf z.B. 40-50 zu verändern. Im Rahmen der Studienarbeit wurden diese Veränderungen der Software vorgenommen. Der Prozess, der zur letztend-

lichen Version des ATEO-Systems geführt hat sowie eine eingehende Beschreibung der Arbeiten an der Software sollen Inhalt dieses Dokumentes sein. Gleichzeitig werde ich auf die Modifizierbarkeit eingehen und in diesem Zusammenhang eine kurze Bewertung der Systemarchitektur geben. Zusätzlich wird ein Ausblick auf weitere Modifikationen auf dem Weg zu ATEO 2.0 gegeben.

# Kapitel 2

# Inhalt

In diesem Kapitel werde ich auf die einzelnen Erweiterungen eingehen und was der Hintergrund dieser ist. Ich werde auch versuchen, die Entstehung der Ideen nachzuzeichnen. Diese Ideen sind von den Psychologen insbesondere Saskia Kain und Jens Nachtwei entwickelt worden. Die Modifikationen sind im Laufe des Projektes verfeinert und angepasst worden.

## 2.1 Genese der Modifikationen

### 2.1.1 Tachometer

Der erste Schritt zur Erhöhung der Komplexität und somit des Konfliktpotenzials ist eine verstärkte Zielvorgabe. Die beiden Mikroweltbewohner waren in der ersten Projektphase von ATEO angewiesen worden, die Strecke möglichst schnell fehlerfrei zu fahren. Hierbei war die Mitte der Straße als Referenz gewählt und die Zeit, in der die Strecke absolviert wurde, wurde nicht explizit angezeigt. Das führte dazu, dass die Versuchspersonen zum Teil recht langsam gefahren sind, um nicht von der Ideallinie abzukommen, und damit die Zeitzielvorgaben weitestgehend ignoriert haben. Zeit und Fehlermaß sind beides wichtige Leistungskriterien zur Bewertung der Teams, da eine Anzeige der Zeit fehlte und die Genauigkeit durch Abweichen von der Mittellinie für die MWB jederzeit erkennbar war, wurde das Fehlermaß ungerechtigter Weise stärker betont. Das Einführen der Zeit soll nun mögliche Konflikte zwischen den Mikroweltbewohnern wahrscheinlicher machen und das fehlende Leistungsfeedback einführen. Die Darstellung der Zeit war am Anfang als einfache Uhr am unteren Rand des Bildschirms angedacht. Hierbei wurde auf jedwede Interpretation verzichtet und nur die abgelaufene Zeit angezeigt. Möglich gewesen wären auch ein Countdown, der von einer festgelegten Zeit herunter zählt, oder auch die Anzeige von abgelaufener Zeit in Bezug auf eine Referenzzeit. Dies kann aber zu einem völligen Ignorieren der Fahrgenauigkeit führen, da die Versuchspersonen nur noch versuchen die Strecke so schnell wie möglich zu absolvieren. Aus diesem Grund wurde die einfache Anzeige der aktuellen abgelaufenen Zeit gewählt. Die Anzeige der Zeit wurde aber im Laufe der Entwicklung durch eine Anzeige der Geschwindigkeit ausgetauscht. Die Geschwindigkeit kann sich in einem Intervall von 0 bis 100 bewegen. Nun ist es immer noch möglich,

den Versuchspersonen die Anweisung zu geben, die Strecke schnell zu absolvieren und sie bekommen auch ein Feedback darüber, wie schnell sie tatsächlich sind. Diesmal aber in der nahe liegenden Form eines Tachometers. Anfangs noch als String mit der aktuellen Geschwindigkeit wurde in der aktuellen Version ein dem Tachometer angelehnter sich füllender Kreis im Fahrobjekt selbst implementiert. Nun ist die gesamte Interpretation der Lage auf die Versuchspersonen geschoben und sie sind auch nicht mehr direkt in der Lage, festzustellen, ob sie gut in der Zeit liegen. Jeder kann nun aus der Geschwindigkeitsanzeige seine eigenen Schlüsse ziehen und für sich entscheiden schneller oder langsamer zu fahren. Dies ist eine individuelle Entscheidung und wird zu Konflikten führen, wenn der Versuchspartner andere Schlüsse zieht.

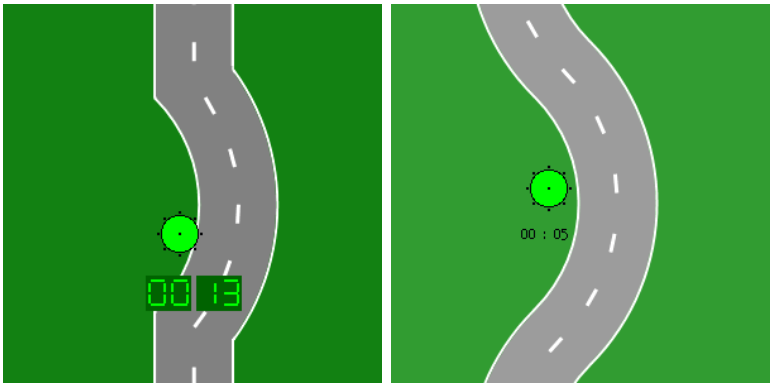


Abbildung 2.1: Zeitanzeige Version 1 und 2

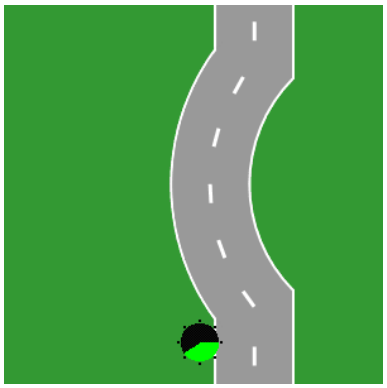

Abbildung 2.2: entgültige Version als Tachometer

### 2.1.2 Manipulation des Steuerungsinputs der Mikroweltbewohner

Durch die Manipulation des Steuerungsinputs soll die Leistungswahrnehmung der Mikroweltbewohner untereinander beeinflusst und das Konfliktpotenzial weiter erhöht werden. Hierbei ist wichtig, dass die Reduzierung des Steuerungsinputs nicht auf das System oder den Versuchsleiter zurückgeführt wird, sondern auf den Versuchspartner und seine schlechtere Fahrleistung bezogen werden. Aus diesem Grund soll es nicht zu einer abrupten und zu starken (aber auch nicht zu schwachen) Reduzierung kommen. Wie weit der Input eines Mikroweltbewohners reduziert wird, wird in einer Voruntersuchung ermittelt. Hierbei werden 4%, 8%, 12% und 16% getestet werden. Dies hat zur Folge, dass das Vorankommen auf der Strecke behindert wird, vor allem die Genauigkeit der Steuerung leidet hier aber auch die Gesamtgeschwindigkeit ist reduziert.

### 2.1.3 Hindernisse

Eine weitere Komponente sind Hindernisse, wobei hierbei zwischen statischen Hindernissen auf der Strecke und dynamischen Hindernissen, die die Strecke kreuzen, zu unterscheiden ist. Die Kollision wird als Fehler gesehen. Nach der Kollision wird das Fahrzeug neben die Fahrbahn gesetzt und die Geschwindigkeit für einen bestimmten Zeitraum auf Null reduziert.

#### statische Hindernisse

Die statischen Hindernisse werden nur auf geraden Strecken positioniert. Es gibt dabei zwei Möglichkeiten, eine 25%ige und eine 75%ige Abdeckung der Fahrbahn. Der Mikroweltbewohner wird hierbei bei 25%iger Abdeckung veranlasst die Geschwindigkeit zu reduzieren, um eine Kollision sicher zu verhindern. Ihm ist aber auch bei hoher Geschwindigkeit möglich, die Strecke nicht zu verlassen und trotzdem das Hindernis zu passieren. Bei 75%iger Abdeckung gibt es nur die Möglichkeit, die Strecke zu verlassen, um dem Hindernis auszuweichen. Hierbei entscheidet wahrscheinlich die Geschwindigkeit, wie viele der Sensoren die Strecke verlassen.

#### dynamische Hindernisse

Die dynamischen Hindernisse erscheinen am rechten oder linken Bildschirmrand und kreuzen auf der Waagerechten die Strecke. Hierbei soll die Geschwindigkeit des Hindernisses soweit angepasst werden, dass ohne Änderung des Fahrverhaltens der Mikroweltbewohner ein Zusammenstoß unausweichlich wird. Sie müssen sich also entscheiden, ob sie die Geschwindigkeit erhöhen, um vor dem Hindernis zu passieren, oder ob sie bremsen und das Hindernis erst passieren lassen. Die Geschwindigkeit des Hindernisses wird hierbei adaptiv ermittelt. Es wird in dem ersten kooperativem Trackingdurchlauf die Geschwindigkeit der beiden Mikroweltbewohner auf dem Streckenteil 26, wo dann auch später die Hindernisse platziert sein werden, gemessen und gemittelt. Diese Geschwindigkeit ist dann auch die des Hindernisses.

## 2.2 Dokumentation der Implementation

### 2.2.1 Tachometer

Die Geschwindigkeit, die vom Tachometer angezeigt wird, ist die interne Geschwindigkeit multipliziert mit dem Faktor 5, um die Anzeige auf ein gewohntes Intervall zu bringen. Da die interne Geschwindigkeit maximal 20.48 betragen kann, ist nach der Multiplikation mit 5 eine mögliche Höchstgeschwindigkeit von 102.4 erreichbar. Dies wird abgefragt und auf 100 korrigiert. Die Anzeige wurde anfangs auf folgendes Format gebracht: 20|100 und als String in einem Stringmorph angezeigt. Dies wird alles in der Methode 'step' von ATEOrealJoystickStepping eingefügt, da hier die Geschwindigkeit, gespeichert in der Variable schritt, berechnet wird. In der nun aktuellen Version des Tachometers wurde die Klasse PCPiePortionMorph missbraucht. Eigentlich ist die Klasse zum Anzeigen von Pie Charts gedacht. Ich habe ein Objekt dieser Klasse erzeugt und in das Fahrobjekt integriert. Das Tortendiagramm besteht hier aus nur zwei Teilen, die dynamisch an die Geschwindigkeit gekoppelt in jedem Step aktualisiert angezeigt werden. Die Aktualisierung wird deswegen auch in der steps Methode von ATEOrealJoysticksStepping implementiert. Die Definition des neuen Fahrobjekts wird wie auch schon zuvor in ATEOCarNoFb→trackerbauen gehandhabt. Hierbei wird pie mit den Anfangswerten 0, 100 initialisiert und in carName integriert. Damit beide Morphe übereinanderliegen, muss noch die Position genau festgelegt werden und pie auf eine bestimmte Grösse gebracht werden. Eine zusätzliche Zugriffsmethode wurde der Klasse hinzugefügt, um Manipulationen am Objekt auch außerhalb von ATEOCarNoFb vornehmen zu können. Dies geschieht z.B. in ATEOrealJoysticksStepping→step.

### 2.2.2 Manipulation des Steuerungsinputs der Mikroweltbewohner

Die Inputmanipulation ist in drei Klassen implementiert worden. Die Konfiguration ist in ATEOVuSt::start als zwei OrderedCollections RedMB und AnteilMBRed initialisiert. Hierbei ist in AnteilMBRed die Reihenfolge der vier verschiedenen Reduzierungsstärken angegeben. Die ersten und letzten beiden Einträge sind 0.0 für beide Mikroweltbewohner, d.h. es findet keine Reduzierung statt. In RedMB wird festgelegt, welcher der beiden Mikroweltbewohner bei einem Step als erstes manipuliert wird. Es werden zwei Teams pro Reihengolge für die Untersuchung gebildet (bestehend aus zwei MWBs). Step 1 und 6 ohne Inputmanipulation in der Tabelle 2.1 wurden bei der Implementierung im Array eingespart. In Step 2 wird für Team 1 MWB1 im ersten Streckenteil eine Inputmanipulation von 4% erfahren und MWB2 im zweiten Teil. Team 2 erfährt die genaue Umkehrung der Manipulation. Dies ist nochmal in den beiden Tabellen 2.1 und 2.2 zusammengefasst.

| Teams | Reihenfolge | Step 1 | Step 2 | Step 3 | Step 4 | Step 5 | Step 6 |
|---|---|---|---|---|---|---|---|
| 1,2 | 1 | 0 | 4 | 8 | 12 | 16 | 0 |
| 3,4 | 2 | 0 | 16 | 4 | 8 | 12 | 0 |
| 5,6 | 3 | 0 | 12 | 16 | 4 | 8 | 0 |
| 7,8 | 4 | 0 | 8 | 12 | 16 | 4 | 0 |

Tabelle 2.1: Inputmanipulation

| Team | Step 2 | Step 3 | Step 4 | Step 5 |
|---|---|---|---|---|
| 1 | 1,2 | 2,1 | 1,2 | 2,1 |
| 2 | 2,1 | 1,2 | 2,1 | 1,2 |

Tabelle 2.2: Reihenfolge der Manipulation

Welche Reihenfolge und welches Team genutzt werden soll, kann vor dem Versuch im Menu ausgewählt werden. Dafür werden in temporären Variablen die entsprechenden Werte gespeichert, um dann entsprechend zu verzweigen und AnteilRedMB und RedMB zu initialisieren.

In der Klasse ATEOPar wird als nächstes in der Methode complexity, die von der Methode bewegen bei jedem Morphstep aufgerufen wird, geprüft, ob in seqColl der dummy 100 vorkommt. Sollte dies der Fall sein wird eine Instanzvariable reduce auf 0 gesetzt. Dies ist der Startpunkt. Nun wird in der Methode bewegen bei reduce = 0 und dem richtigen Farbcode im Object Ctrl die Inputmanipulation durch das Setzen der Variable anteilReduzieren ausgelöst.

In der Methode ATEOAblaufsteuerung::makeItComplex wird, wenn anteilReduzieren auf true gesetzt ist, in RedMB nachgeschaut, bei wem der Input reduziert werden soll. Möglich sind hierbei 0, 1 und 2. Anschließend wird mit Hilfe von AnteilRedMB der gewünschte Prozentsatz in ein prozentigen Schritten reduziert oder im Falle von 0 gar nicht reduziert. Am Ende jeden Falles wird reduce auf 1 gesetzt.

Zurück in ATEOPar::bewegen wird auf reduce = 1 reagiert, wenn der Farbcode für das Ende der Inputmanipulation ermittelt wird. Im Ctrl Objekt wird nun anteilReduzieren auf false und anteilErhöhen auf true gesetzt. Darauf wird in ATEOAblaufsteuerung::makeItComplex reagiert, indem der Inputanteil des entsprechenden Mikroweltbewohners wieder auf 50% erhöht wird. Am Ende jeden Falles wird reduce auf 2 gesetzt und der Zugriffsindex von RedMB um eins erhöht, damit der nächste Mikroweltbewohner manipuliert werden kann. In ATEOPar::bewegen bewirkt reduce=2, dass in Ctrl die Variable anteilErhöhen wieder auf false gesetzt wird. Damit ist ein Durchlauf beendet, alle Zustandvariablen sind wieder auf false und der Automat wird bei dem nächsten Vorkommen des dummys 100 in den Zustand 0 zurückgesetzt. Dies wird in Bild 2.2 noch einmal grafisch verdeutlicht.

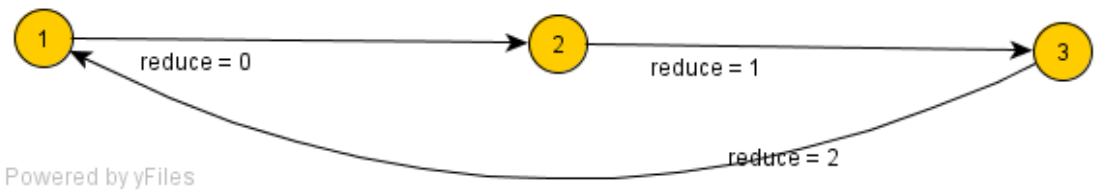


Abbildung 2.3: Graph der Manipulationsabfolge

### 2.2.3 Hindernisse

Die beiden verschiedenen Varianten von Hindernissen werden beide in der Klasse ATEOAblaufsteuerung implementiert. Dies war am Anfang auch noch sinnvoll, aber im Laufe der Arbeiten wurden die Hindernisse immer komplexer, so dass nun eine Implementierung als eigene Klasse die beste Lösung ist. Dies wird im Anschluss an diese Arbeit umgesetzt. In eine Strecke können die Hindernisse durch einen Dummy in den Streckenkonfigurationsdateien eingefügt werden. Die Verarbeitung geschieht in ATEOPar::complexity. In folgender Tabelle sind alle möglichen Dummy-Einträge gelistet.

| Dummy | Erweiterung |
|---|---|
| 100 | Inputmanipulation |
| 1100 | statisches Hindernis mit 25%iger Fahrbahnabdeckung |
| 1101 | statisches Hindernis mit 75%iger Fahrbahnabdeckung |
| 111 | dynamisches Hindernis |
| 201 | adaptive Geschwindigkeitserkennung |

Tabelle 2.3: Bedeutung der Dummies

In ATEOPar::complexity wird auf jeden einzelnen Dummy reagiert, indem spezielle Variablen gesetzt werden, die dann in ATEOAblaufsteuerung::makeItComplex zu Aktionen führen. Welche dies im Einzelnen sind, wird an den entsprechenden Stellen erläutert.

#### statische Hindernisse

Die Erstellung des statischen Hindernisses wird durch zwei Methoden abgedeckt, und zwar place25 und place75. Beide Methoden unterscheiden sich nur im Detail, zur leichteren Handhabung und zur Erhöhung der Lesbarkeit des Codes aber wurden zwei Methoden weiteren Parametern vorgezogen. Der Unterschied liegt in den unterschiedlichen x- und y-Koordinaten. Diese werden in den Instanzvariablen xPos und yPos gespeichert, damit sie im Logfile jederzeit abgelegt werden können. Die Koordinaten als Parameter für eine einzige Methode zu benutzen, wurde verworfen, da dann die Methode makeItComplex immer länger und die place Methode immer kürzer geworden wäre. Die beiden Koordinaten wurden durch Verschieben der Morphe genau bestimmt und sind nur für die Positionierung auf einer im rechten Drittel der Strecke verlaufenden Geraden gedacht. In der Tabelle 2.4 sind die zwei weiteren möglichen Platzierungen notiert. Dabei wird immer von der selben Geraden 26 ausgegangen.

| Koordinate | Gerade links | Gerade mittig | Gerade rechts |
|---|---|---|---|
| x (25% Abdeckung) | 116 | 418 | 718 |
| y (25% Abdeckung) | -924 | -924 | -924 |
| x (75% Abdeckung) | 87 | 387 | 687 |
| y (75% Abdeckung) | -1500 | -1500 | -1500 |

Tabelle 2.4: Koordinaten für statische Hindernisse

In ATEOPar::complexity werden zum Starten der Erweiterung die Variablen statHindernis auf true und statBarrierPlacing auf 25 bzw. 75 gesetzt. Beide bewirken in ATEOAblaufsteuerung::makeItComplex, dass das richtige Hindernis an der richtigen Stelle zum richtigen Zeitpunkt erstellt wird. Nach der Erstellung wird es dem Streckenbildmorph hinzugefügt und somit zusammen mit dem Bild zerstört. Welchem Morph es hinzugefügt wird, ist jedes Mal neu zu bestimmen, da zwei Bilder gleichzeitig in den Variablen bild1 und bild2 (beide in ATEOPar) gehalten werden. Sollte also das Hindernis auf einem anderen Streckenbild als im Moment auf der 26 erscheinen, muss hier evtl. eine Anpassung vorgenommen werden.

**dynamische Hindernisse**

Dynamische Hindernisse unterscheiden sich durch ein wichtiges Detail von ihren statischen Verwandten. Sie müssen bewegt werden. Deshalb ist das erzeugen in ATEOAblaufsteuerung::createDynamicBarrier auch nur in der Positionierung verschieden von ATEOAblaufsteuerung::place25. Positioniert wird das Hindernis im zum Zeitpunkt der Erzeugung nicht sichtbaren Bereich 0@-2200 und verharrt dort, bis es gesehen werden kann. ATEOAblaufsteuerung::moveDynamicBarrier ist auch einfach gehalten. Das Hindernis wird um $speed/2$@0 pixel nach rechts bewegt. Wenn es rechts aus dem Bild verschwindet, wird es zerstört. Die Berechnung von speed ist dagegen komplizierter und erstreckt sich über zwei Klassen, nämlich ATEOPar und ATEOAblaufsteuerung. Die Berechnung beginnt in der Methode ATEOAblaufsteuerung::step:punkt:, wo der Parameter schritt, errechnet in ATEOrealJoysticksStepping::step, in der Instanzvariable schrittVar gespeichert und ATEOPar::bewegen übergeben wird. Auf den Streckenteil 26, welches im kooperativem Tracking dreimal vorkommt, wird die Geschwindigkeit des zu steuernden Objektes gemessen. Ausgelöst und beendet wird die Messung von bestimmten Farbcodes im Rand der Streckenbilder. Die eigentliche Berechnung findet dann wieder in der Klasse ATEOAblaufsteuerung statt, und zwar in makeItComplex. Wenn die Messung durch die Variable speedCount aktiviert ist, wird die Geschwindigkeit schritt in spd aufaddiert und in count die Anzahl der Additionen gespeichert. Wenn die Messung beendet ist, ruft ATEOPar::bewegen die Methode ATEOAblaufsteuerung::calculateSpeedOnce:. Diese Methode berechnet nun die Durchschnittsgeschwindigkeit auf dem Streckenteil 26, indem die aufaddierte Geschwindigkeit durch die Anzahl der Messungen dividiert wird. Der Instanzvariable spdC, welche die so berechneten Streckenteilgeschwindigkeiten speichert, wird der in vorherigen Messungen gespeicherte Wert aufaddiert. spd und count werden auf null gesetzt und spdC zurückgegeben. In ATEOPar::bewegen wird der Rückgabewert in cumulativeSpeed gespeichert und nach drei Messungen, wenn das Ende der Strecke erreicht ist, wird dieser Wert durch 3 dividiert, um den Mittelwert aller drei Messungen zu ermitteln. Durch eine get-Methode kann nun ATEOAblaufsteuerung::moveDynamicBarrier auf die adaptiv ermittelte Geschwindigkeit zugreifen.

Ausgelöst wird die Erzeugung eines dynamischen Hindernisses in ATEOPar::complexity mit dem dummy-Code 111 und der Variable dynHindernis.

**Kollision**

Aufgerufen wird die Kollision in ATEOCarNoFb::feedback. Hier wurde in der Schleife zur Bestimmung der sich außerhalb der Strecke befindlichen Sensoren eine weitere Abfrage eingebaut. Wenn ein Sensor nun ein Hindernis streift, wird ATEOAblaufsteuerung::collision aufgerufen. Der Algorithmus für die Kollision ist recht einfach gehalten: Das Objekt wird in einer while-Schleife solange auf der x-Achse um einen Pixel nach links gefahren, bis die beiden Sensoren auf der linken und rechten Seite grün (Color r: 0.199 g: 0.599 b: 0.199) melden. Im Anschluss wird das Auto durch eine for-Schleife am selben Platz gehalten, indem es einen Pixel nach links und dann einen Pixel nach rechts bewegt wird. Durch die hohe Geschwindigkeit der Abarbeitung und der geringen Bewegung um einen Pixel scheint das Objekt sich nicht von der Stelle zu rühren. Damit es nicht im Anschluss durch ATEOCarNoFb::startPos auf seine alte Position korrigiert wird, wird die nach der Verschiebung aktuelle Position des Objektes in der privaten Instanzvariable collisionKoordinate gespeichert, um später als Parameter für ATEOCarNoFb::startPos zu dienen. Die letzte Zeile zerstört das Hindernis.

### 2.2.4 Das neue Interface

Wegen einer Portierung auf die Squeak Version 3.7, musste das Konfigurationsinterface neu geschrieben werden. Das alte Interface wurde noch mit E-Toys erstellt und war deswegen schwer zu warten. Aus diesem Grund wurde die Gelegenheit genutzt, um dieses Interface durch Smalltalk zu erzeugen. Den Anfang bildet hierbei ein neuer Button 'begin', welcher das Menu erzeugt. Es wird die Methode ATEOConfig::start aufgerufen, welches dann das Interface erzeugt und in der globalen Variable Configure speichert. Die Erzeugung wird dann komplett in ATEOConfig::initialize abgehandelt. Es werden hierfür acht Einstellungsfelder erzeugt, wobei fünf davon direkt vom alten Menu übernommen wurden. Das Einstellungsmenu Operateur erlaubt es, wie bei der Teamnummer, die unterschiedlichen Operateure im Logfile eindeutig zu identifizieren. Team und Reihenfolge der Manipulation sind für die Einstellung der Imputmanipulation, die Einstellungsmöglichkeiten sind in Tabelle 2.1 und 2.2 erfasst. Die beiden Buttons Los und Joystick/Tastatur sind auch aus dem alten Konfigurationsmenu übernommen, die Funktionalität ist die selbe geblieben. Wenn nun auf den Button Los gedrückt wird, wird der Versuch mit der Funktion ATEOConfig::go konfiguriert und gestartet. Hierbei ist die Methode ATEOConfig::go der alten player-Methode nachempfunden und die Funktionalität lediglich erweitert. Im Einzelnen werden globale Variablen mit Werten aus dem Menu oder Konstanten initialisiert. Am Ende wird die Versuchssteuerung VuID erzeugt und der ATEOVuSt::start gesendet, nachdem die Konfiguration versteckt wurde. Die Klasse hat noch einige Funktionen zum erfragen von privaten Instanzvariablen. Diese werden vorrangig für das erweiterte Logfile benötigt. Leider ist die ATEOConfig::initialize sehr umfangreich geworden, eine Codeoptimierung ist demnach vorgesehen, damit doppelter Code und die große Anzahl von Instanzvariablen minimiert werden kann. Optisch sieht das Interface auch nicht sehr ansprechend aus, dies ist der fehlenden Erfahrung im Interfacedesign mit Smalltalk geschuldet. Auch gibt es keine Dokumentation oder Toolunterstützung wie Java Netbeans etc.

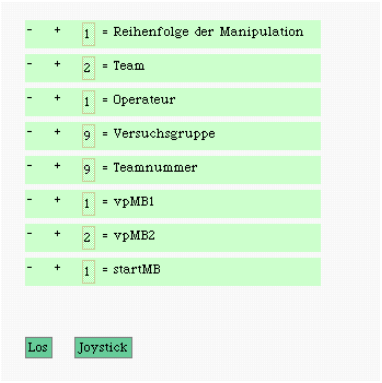


Abbildung 2.4: das neue Interface

## 2.3 Zusammenfassung und Statistik der Implementation

| Klassenname | zusätzliche Methoden | modifizierte Methoden |
|---|---|---|
| ATEOAblaufsteuerung | 16 | 2 |
| ATEOCarNoFb | 2 | 3 |
| ATEOConfig | 8 | 0 |
| ATEOLog | 0 | 0 |
| ATEOOperateur | 0 | 0 |
| ATEOPar | 6 | 1 |
| ATEOTastaturabfragen | 0 | 1 |
| ATEOVuSt | 0 | 3 |
| ATEOrealJoysticksStepping | 0 | 1 |

Tabelle 2.5: Statistik

## 2.4 Performance Problem

Je mehr Erweiterungen implementiert wurden, desto schlechter wurde die Performance des Systems. Als Richtlinie wird hierbei die Zeitdifferenz zwischen zwei Logfileeinträgen genutzt. Das Optimum ist auf 40 ms gesetzt. Je mehr Befehle innerhalb eines Steps abgearbeitet werden, desto mehr wird das Schreiben des Logs hinausgezögert. Es kommt auf dem alten Rechner dabei zu Verzögerungen von bis zu 100 ms. Ein Engpass konnte ich hierbei explizit ausmachen. In der Methode feeback der Klasse ATEPCarNoFb wird über die OrderedCollection, die die Sensoren des zu steuernden Objektes beinhaltet, iteriert und festgestellt, wieviele dieser Sensoren nicht mehr auf dem Grau der Straße fahren, sondern Grün unter sich finden. Diese Methode wird bei jedem Step aufgerufen und geht demnach jedes Mal alle Sensoren durch. Dies führt zu beträchtlichen Verzögerungen. Wenn nur ein Sensor überprüft wird und es nur zu einem indizierten Zugriff kommt, können ohne

Probleme mindestens 30 ms gespart werden. Diese Funktion wird sicher nur ein Engpass darstellen, aber ein schwerwiegender ohne Frage. Leider kann auf diese Methode nicht verzichtet werden und im Rahmen der Hinderniskollision wurde diese noch um Code erweitert. Es wurde überlegt, ob man Sensoren sparen könne, aber alle Sensoren werden benötigt, weshalb nun das Programm auf leistungsstärkeren Rechnern getestet wird. Ein weiteres Problem ist die lange Ladezeit beim Start. Dies wird mit hoher Wahrscheinlichkeit durch das Laden der Streckenbilder verursacht. Zwei Lösungswege wären denkbar. Einmal könnte man die Streckenbilder als Morphe abspeichern und diese dann vor oder während des Veruschsdurchlaufes laden. Eine andere Möglichkeit ist, die Strecke mit Hilfe von Bezierkurven dynamisch zu erzeugen. Dies hat den Vorteil, die Strecke leichter manipulieren zu können. Welchen Einfluss die Berechnung auf die Gesamtperformance hat, ist ungewiss. Meine Einschätzung ist, dass das Problem nur von einem Punkt zum anderen verschoben wird. Das Programm wird sehr schnell laden, aber die Zeitdifferenz wird weiter steigen. Über die tatsächlichen Auswirkungen kann man natürlich nur spekulieren. Da das Problem der langen Wartezeit vor dem Versuch im Moment unwichtig erscheint, wurden noch keine Untersuchungen in diese Richtung gemacht. Die Implementierung der Strecke mit Hilfe der Bezierkurven bedeutet einen großen Eingriff in die Architektur des Systems. Der Aufwand ist also im Moment ebenfalls nicht gerechtfertigt.

## 2.5 Auswertung

Die Geschwindigkeitsanzeige im Fahrzeug war einfach zu integrieren. Auch die Streckenerweiterungen wie Hindernisse und Inputmanipulation waren anfangs einfach zu integrieren. Am Ende war es aber sehr anstrengend, diese dann für die Versuchsreihe zu konfigurieren. Man musste immer an zwei Klassen arbeiten und manchmal an der ATEOConfig Elemente hinzufügen und damit die ganze Initialisierung von Versuchsparametern in einer nächsten Klasse ATEOVuSt vornehmen. Der Umstand, dass man sehr viel in den Klassen springen musste, spricht für eine komplizierte Architektur. Einiges kann man nicht verhindern, die Konfiguration und die Versuchsteuerung können nur voneinander abhängen, hier kann man aber eventuell einiges von der ATEOVuSt in die ATEOConfig übernehmen. Generell hat man viel mit den Entscheidungen, die getroffen wurden, zu kämpfen. Vieles wüsste ich auch nicht anders zu machen. In diesem Fall wäre eine komplett neue Strategie am besten. Eine weitere Gefahr besteht darin, dass man den vorgegebenen Stil weiterführt. An vielen Stellen wurde prozedural programmiert. Eine Erweiterung ist dann nur eine weitere If Anweisung.

## 2.6 Ausblick

Im Zuge der Folgearbeiten, möchte ich die Dinge beenden, die ich aus Zeitgründen nicht geschafft habe. Dies wären zum einen eine objektorientierte Umstrukturierung des vorhandenen Codes. Hierbei ist nicht nur der alte Code betroffen, sondern auch der neue Code zu den Erweiterungen. Im Laufe der Programmierung sind die Klassen durch die Erweiterungen zu groß geworden und es lohnt sich jetzt für Dinge wie die Hindernisse

eigene Klassen zu schreiben. Anfangs bestanden diese nur aus einer oder zwei Methoden, der Rest des Codes wurde in vorhandene Methoden geschrieben. Diese Entkopplung nicht schon am Anfang gemacht zu haben, führt nun zu unübersichtlicheren Klassen und auch zu langen Methoden. Laut [Sha97] sollte eine gut programmierte Methode in Smalltalk in einem Fenster der Entwicklungsumgebung vollständig lesbar sein. Dies gilt sicher für alle objektorientiert programmierten Methoden, in ATEO gibt es Methoden, die das vorgebende um mehrere Bildschirme übersteigt. Dies ein wenig aufzulösen und objekt orientierter zu gestalten ist ein Ziel. Wenn es möglich ist, möchte ich auch die globalen Variablen, die im Global Dictionary Smalltalk stehen, durch Instanzvariablen oder Pool Dictionaries ersetzen. Dies wird zu tiefgreifenden Änderungen im System führen. Das Gelingen des Unterfangens hängt dann von der Systemarchitektur ab. Nötig für eine Performanzsteigerung ist dies wahrscheinlich nicht, aber zur Lesbarkeit und Erweiterbarkeit des Quellcodes wird es entscheidend beitragen.

Der zweite und größere Teil der Folgearbeiten wird die Vernetzung zwischen dem im Moment bestehenden ATEO System und dem Operateursarbeitsplatz sein. Diese Arbeiten werden durch eine verbesserte Struktur des ATEO Systems erleichtert, weshalb sich die Vorarbeiten und die Bemühungen um mehr Objekte im System lohnen. Welche Netzarchitektur dem zu Grunde liegen soll, wird genauso Teil der Arbeit sein, wie die Definition der Funktionalität, die benötigt wird, um einen Arbeitsplatz für den Operateur effektiv und anpassungsfähig zu gestalten. Die bestehende Implementation ist, so weit es sich jetzt abschätzen lässt, nur mit einem gewissen Aufwand als verteilte Anwendung umsetzbar. Das ATEO System könnte als Server die verschiedenen Clients mit Informationen versorgen, das Interface muss aber erst geschaffen werden. Den Operateursarbeitsplatz zu entwerfen, wird Aufgabe von Hermann Schwarz sein. Für die Arbeiten an der Klasse ATEOLogfile und die darauf aufbauende Auswertung ist Sascha Hanse verantwortlich. Michael Hildebrandt wird die Systemarchitektur untersuchen und eine neue verbesserte Architektur entwerfen.

# Literaturverzeichnis

[Sha97] Alec Sharp. *Smalltalk by Example - A Developer's Guide.* McGraw-Hill, 1997.