DISS. ETH NO. XXX

DISTRIBUTED INTELLIGENCE IN REAL WORLD MOBILITY SIMULATIONS

A dissertation submitted to the

SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of

Doctor of Technical Sciences

presented by

CHRISTIAN DANIEL GLOOR

Eidg. Dipl. Inf. Ing. ETH Zürich

born December 27, 1975

citizen of
Seon, Aargau

accepted on the recommendation of
Prof. Dr. Kai Nagel, examiner
Prof. Dr. Willy Schmid, co-examiner

2005

To all creatures lucky enough
to live in a world that is modeled better.

Contents

I A Framework for Modular Multi Agent Simulations **1**

1 Introduction **2**

1.1 General context 2

1.2 Modeling Approach Overview 4

2 Framework **8**

2.1 Introduction 8

2.2 The Structure of the Framework 10

2.2.1 Physical Layer Modules 10

2.2.2 Mental Layer Modules 11

2.2.3 Other/Helper Modules 14

2.3 Discussion and Outlook 17

II The Modules of the Framework **19**

3 Mobility Simulation **20**

3.1 Existing approaches 21

3.1.1 General simulation techniques for mobility simulations 21

3.1.2 Microscopic simulation techniques for mobility simulations 21

3.1.3 Discretization 23

3.1.4 Desired velocity 24

3.2 Our Approach to the Desired Velocity Problem 25

3.2.1 Naive approach 25

3.2.2 A More Sophisticated Approach 25

3.2.3 Implementation details 29

3.3 Model Calibration 29
3.3.1 Experimental setup 31
3.3.2 Calibration of the path force 32
3.3.3 Calibration of pedestrian interaction 33
3.3.4 Fundamental diagrams, and the interaction strength A 34
3.4 Computational aspects 34
3.5 Movements inside Town or Buildings, or close to obstacles 39
3.5.1 Potential Field Model 40
3.5.2 Graph Model 42
3.5.3 Simulation of Zürich Main Station 44
3.6 Pedsim, A Lightweight Version for the Public 46
3.7 Cable Car Simulation 50
3.7.1 Introduction 50
3.7.2 Implementation 50
3.7.3 Coupling 51
3.8 Summary 51

4 Route Generator **53**
4.1 Introduction 53
4.2 Knowledge Representation 54
4.2.1 Temporal Representation of Knowledge 54
4.2.2 Individual Knowledge 56
4.2.3 Global Knowledge 57
4.3 Route Calculation 57
4.3.1 Shortest Path Algorithms 57
4.3.2 Generalized Costs 58
4.4 Listening to Events 58
4.4.1 Direct Spatial Events 59
4.4.2 Indirect Spatial Events 61
4.4.3 Personal Spatial Events 61
4.4.4 Non-Spatial Events 62
4.5 The Message Interface 62
4.6 Conclusion 64

5 Agent Database **66**

5.1 Introduction . . . 66

5.2 Agent Database in the Traffic Project . . . 66

5.3 The Simpler But Networked Agent Database . . . 67

5.3.1 Functionality . . . 67

5.3.2 Implementation(s) . . . 69

5.4 Summary . . . 70

6 Visualization **72**

6.1 Introduction . . . 72

6.2 The 2-dimensional Visualizer . . . 73

6.2.1 Output . . . 73

6.2.2 Input . . . 79

6.3 The 3-dimensional Visualizer . . . 81

6.4 The Offline Renderers . . . 82

6.4.1 PovRAY . . . 84

6.4.2 Renderman . . . 86

7 Communication **89**

7.1 Introduction . . . 89

7.2 Subroutine Calls . . . 90

7.2.1 Locally Linked . . . 90

7.2.2 Remote Procedure Call (RPC) and Common Object Request Broker Architecture (CORBA) . . . 90

7.3 Files . . . 91

7.4 Databases . . . 92

7.5 Message Passing: MPI/PVM . . . 92

7.6 Transmission Control Protocol (TCP) . . . 93

7.7 User Datagram Protocol (UDP) . . . 94

7.8 Multicasting . . . 95

7.9 TCP Multicast Daemon . . . 97

7.10 XML . . . 99

7.10.1 Extensible ASCII Messages . . . 99

7.10.2 Extensible Binary Protocol . . . 102

7.10.3 Sending raw binary data . . . 104
7.11 Conclusions . . . 104
7.12 Discussion . . . 104

8 Synchronization **106**
8.1 Introduction . . . 106
8.2 A Simple Mechanism: Suspend-On-Demand . . . 108
8.2.1 Description . . . 108
8.3 A Slightly More Sophisticated Mechanism: Advance-To . . . 110
8.3.1 Description . . . 110
8.4 Discussion . . . 112

III Application **113**

9 Sensitivity Studies **114**
9.1 Single Agent Scenarios . . . 114
9.1.1 Intraday Re-Planning . . . 114
9.1.2 Generalized Cost Function in Router . . . 117
9.1.3 Time Dependent Routes . . . 120
9.2 Multi Agent Scenarios . . . 121
9.2.1 Same Destination, Different Paths . . . 121
9.2.2 Two Destinations to Chose From . . . 125
9.2.3 The “Exploraition Scenario” . . . 128

10 General Use of the Framework **131**
10.1 Integration of 3rd Party Modules into the Existing Framework . . . 131
10.1.1 Adding Physical System Information into the Events Stream: Visual Analyzer . . . 131
10.1.2 Building Mental Maps for Individual Agents . . . 133
10.2 Using the Same Framework for a Completely Different Application: Value Added Services for Day Traders . . . 134
10.3 Summary . . . 137

11 Outlook, Summary and Conclusions **138**
11.1 Outlook . . . 138

11.2 Summary and Conclusions 140

A Algorithms **142**

A.1 Dijkstra's Algorithm 142

A.2 Time Dependent Dijkstra 143

A.3 Time Dependent Dijkstra Using Generalized Costs 144

B Network File **147**

B.1 Links (Streets) 147

B.2 Nodes (Streets) 148

B.3 Objects 148

C Communication **149**

C.1 Events Channel 149

C.2 Brain Comm Channel 149

D Module Usage **151**

D.1 Communication Channel 151

D.2 Mobility Simulation 151

D.3 PEDSIM 151

D.4 Cable-Car Simulation 152

D.5 Route Generator 152

D.6 2-dimensional Visualizer 152

D.7 Agent Database 152

List of Figures

1.1 The test site with its significant topography, a photo and its virtual representation. . . . 4

1.2 Concept of two layers in a typical simulation system 6

2.1 The complete simulation system, according to the modules used 9

2.2 Alternative approach to a full simulation system, where the multi-layer approach to the agent databases of Fig 2.1 is replaced by a single monolithic agent database 17

3.1 Traces of hikers in the first model, where they are all pulled toward the same waypoint 26

3.2 Example of the force toward the waypoint. It is at the same time an illustration of the use of cells for local force information. 26

3.3 Path-oriented coordinate system for the computation of the desired velocity and the path forces 27

3.4 Traces of pedestrians walking in the same direction according to the second model, where they stay on their side of the path, even at a bend 27

3.5 If the agent passes a virtual finishing line, it switches to the new link 29

3.6 Angle between the "force" caused by the desired velocity, and the path force, which is both a function of the path width, and the distance from the center of the path 30

3.7 Force $|\mathbf{f}_{path}(h)|$ in a 2 meters wide sidewalk ($B_1 = 0.5$) 30

3.8 Potential in a 2 meters wide sidewalk ($B_1 = 0.5$), from Mauron (2002). 31

3.9 Observed sidewalk at the Tannenstrasse 35

3.10 The measured quantities at the observed sidewalk 35

3.11 Real-world pedestrian distribution for non-interacting and interacting pedestrians . . . 36

3.12 Validation: Real-world and simulated single pedestrian distribution 37

3.13 Average speed $\overline{v}$ vs. pedestrian density for uni-directional flow and randomized boundary conditions ($B_{pp} = 1$, $L = 10$m, $W = 2$m) 37

3.14 The maximal velocity of walking pedestrians decreases if their density rises, and when oncoming traffice increases. $A_{pp} = 300, B_{pp} = 1, A_{path} = 6000$, random boundary conditions 38

3.15 Since block consume a lot of memory, they are loaded into memory as soon as an agent walks over, and are deleted after a while. 40

3.16 The scene is divided into multiple "blocks". Only forces from adjacent blocks have an influence to an agent. 40

3.17 A potential field generated for one destination in the center of the formation and for two destinations 41

3.18 For the computation of the potential field, different parameters influence the time needed 42

3.19 Possible walking directions $\mathbf{v}_i^0$ are limited by the number of neighbor cells looked at for a lower potential 42

3.20 The hybrid simulation technique. The forces are valid for the whole cell; a pedestrian's trajectory can follow arbitrary positions 43

3.21 Solutions to place nodes around an obstacle 44

3.22 An evacuation of Zürich Main Station using the potential field model. The pedestrians hurry to the closest exit available 45

3.23 Potential field and spanning tree for Zürich Main Station, generated for the 8 potential exits 46

3.24 For the computation of the potential field, different parameters influence the time needed: If we reduce the size of the ground cells, the time to precompute the potential field rises 47

3.25 The 2-dimensional visualizer, connected to the crowd simulation "pedsim". The trails of each pedestrian shows how it get diverted by other pedestrians. 47

4.1 Graphical comparison of the three variants of Dijkstra's algorithm used 55

4.2 The 2-dimensional visualizer is able to display `route` events sent by the router as an answer to a `route_request`, represented by tags with the node number 59

4.3 A route from the hotel to the mountain top of rellerli after it started to rain in the area where the shortest route was before 60

4.4 The routers main task is to answer `route_requests` for the Agent Database. In order to build its view of the virtual world, it receives events from the simulation and adjacent modules 60

5.1 The agent database is a central part in the framework. It connects the modules in a layer (left, rigt) and adjacent layers as well (up, down). 67

5.2 Flowchart of an agent database which polls the route generator to determine which activity might be better. The agents hike up and down, each time they chose the activity that fits their preferences better 71

6.1 The 2-dimensional visualizer displays information related to the street network. It shows link and node IDs, and the street name, if given in the network file 74

6.2 The 2-dimensional visualizer shows lots of details if configured to do so, like in this example of the street network of Zürich 75

6.3 The 2-dimensional visualizer is capable to show agent related data, for example the agent's IDs, or to plot a trail of the last known positions of an agent. 76

6.4 The 2-dimensional visualizer is capable to show the events sent to the event streams . . 77

6.5 The 2-dimensional visualizer is able to display networks used for traffic simulations as well, like the street network of switzerland 79

6.6 A visualizer does display the events received from the events stream. In order to maintain a sound design of the framework, input should be achieved through independent modules 80

6.7 Since the code to display a 2-dimensional representation of the simulated area and react to user input was already here, this 2-dimensional visualizer now has the ability to send `rain` events, and lets the user interact directly with the system 81

6.8 The 3-dimensional visualizer by Duncan Cavens. In this picture, a early version can be seen. Agents walk on a trail to the *Hugeligrat* (Rellerli). 82

6.9 The 3-dimensional visualizer by Duncan Cavens. A view of the village of *Schönried*, with *Rellerli* and *Hugeligrat* in the background 83

6.10 The offline renderers allow to generate a photo-realistic image of the scenario. However, to produce this image of high quality, a decent computer has to calculate the traces of light rays for about 10 minutes 84

6.11 The same view as shown in Figure 6.9 and Section 9.2.3 created by using the offline renderer and PovRAY: a view toward *Rellerli* and *Hugeligrat* 85

6.12 The 2-dimensional visualizer, connected to the crowd simulation "pedsim". The trails of each pedestrian shows how it get diverted by other pedestrians. This is part of the scene rendered for Figure 6.13. 87

6.13 A frame of a "pedsim" scene rendered by the off-line renderer using Povray. This is the same scene as displayed in the 2-dimensional visualizer in Figure 6.12 88

7.1 UDP packets sent at different rates using an 1 Gbit/s network, plotted as sent vs. received messages 95

7.2 Multicast uses one data connection, even if there are multiple receivers subscribed to the multicast channel. 96

7.3 The TCP Multicaster is a communication module which accepts incoming TCP connections, and allows to mix different kind of XML event streams in a module 98

7.4 Comparison of parsing speed: Expat, XML Subset Parser, Extensible Binary Protocol vs. raw binary messages. 101

7.5 Number of messages the receiving module is able to parse. The more CPU cycles are used by the parser, the more messages are lost in the network 102

7.6 Our largest street network scenario, Greater Zürich, consists of 166'000 nodes and 205'000 links. On a busy rush-hour simulation, there are up to 100'000 agents moving on these links. 105

8.1 A problem with the Suspend-On-Demand algorithm is that even if the agent database module decides infinitely fast that a suspend is needed, the mobility simulation has plenty of time to run further 109

8.2 Depending on when each module proposes a new advanceto time to the controller, it is possible that one modules runs without discontinuation. 111

8.3 The problem with the Suspend-On-Demand algorithm is less serious with the Advance-To algorithm: even if a module runs very fast, it will wait after it has reached the next advanceto time. 112

9.1 After an agent was hit by a raindrop, it is heading home on the shortest path immediately, but avoids areas where rain was detected. 116

9.2 It is possible to use other criteria than just the fastest path in the route generator. E.g. rain and sun can be liked or not. 119

9.3 This experiment was done in order to demonstrate that not only the link travel times are stored in multiple time bins, but the other attributes (e.g. sun, rain) as well 122

9.4 Same destination, different kind of hikers. In the 1st iteration, every agent walks on the same path, although they have different preferences 124

9.5 Two different activities to chose from, two groups of hikers. In the beginning, they chose randomly 126

9.6 It is assumed that all agents leave in the morning from the hotel and hike to the same mountain peak. They start to explore the area in hope to find a faster route. 129

10.1 The events sent by the mobility simulation are received by the mental layer modules. The view analyzer module listens to these events as well 132

10.2 The program by D. Kistler includes not only the mental map module, but also functionality of the agent database in the second mental layer 133

10.3 The view analyzer module reports how much forest is in the field of view of an agent. These messages include the current location of the agent and the direction and distance where the forest is 134

10.4 The same framework can also be used for a completely differenct application: for processing order strategies in an automated trading system. 135

Abstract

Recent developments in the virtual representation of landscapes suggest that it might be possible to reliably observe the behavior and deduct landscape preferences of people in a more controlled laboratory environment as it is provided through new developments in computer and projection technology. Such visualizations permit to show the visual landscape both at present and at potential future situations, giving a tool to systematically investigate how people react to changes in the landscape.

However, such a visualization of a changed landscape would be unpopulated, since no data about human behavior exists for this changed landscape. Autonomous agent modeling, which is based on concepts from artificial life and computer simulations, enables one to populate the virtual world with rule-driven agents which can act as surrogates for real humans.

This work presents a multi-agent simulation to model the activities of tourists (primarily hikers). The goal is to have these agents populate a virtual world, where they are able to evaluate different development scenarios. At the same time, the project is used to explore general computational implementations of mobility simulations.

Most simulation efforts in spatial planning have focused on large spatial scales (such as at the city and regional levels) and on relatively abstract concepts (such as land use patterns, traffic and economic development), while one can argue that the planning decisions that have the most impact on individual citizens tend to be either at a relatively small scale or have very local impacts.

At these small scales, visual elements and the overall visual quality of the proposed planning intervention are extremely important. This is particularly true of areas dependent on tourism, which are often promoted based on their scenic qualities.

In addition to the community's desire to diversify its recreational economy, there are landscape policy issues that have the potential to change the desirability of the area for summer tourism. These issues include changes to the pattern of the landscape due to changing agricultural policy, shifts in forestry practices, closing of the gondolas and/or chairlifts, and increased holiday home construction. Any of these changes would have complex repercussions for the tourism industry: scenarios to test the agent model are selected from them.

Our approach is to model each tourist individually as an agent. A synthetic population of tourists is created that reflects current (and/or projected) visitor demographics. These tourists are given goals and expectations that reflect existing literature, on-site studies, and, in some cases where sufficient data is not available, are based on experts' estimates. These expectations are individual, meaning that each agent could potentially be given different goals and expectations.

These agents are given plans, and they are introduced into the simulation with no knowledge of the the environment. The agents execute their plans, receiving feedback from the environment as they move throughout the landscape. At the end of each run, the agents' actions are compared to their expectations. If the results of a particular plan do not meet their expectations, on subsequent runs the agents try different alternatives, learning both from their own direct experience, and, depending on the learning model used, from the experiences of other agents in the system.

This work has a strong computational component. Besides the presentation of the framework and its modules, computational aspects of the pedestrian simulation are discussed. Using sophisticated techniques which adapt the model to the rather special circumstances of hikers in the alps, it is now possible to simulate an area of $15km^2$ and more on a single CPU.

This new simulation also works close to obstacles, as are found e.g. close to buildings. Also the simulation of the inside of buildings is possible, which allows the usage of the same framework for e.g. evacuation simulation.

The known modular framework of traffic simulations was developed further by using the same modular structure, but introducing a new way for coupling these modules. New functionality was also developed. The modules are connected no longer by files, but by network messages. This allows real-time interaction between the modules, like an agent reacting to a new situation immediately.

It is also possible to attach new modules to the framework without modifications of the design. This allows one to use modules that process the raw data from the simulation and feeds value-added data back into the system. It is possible to have several modules jointly compute different aspects. The communication between the modules is presented in detail.

Zusammenfassung

Fortschritte in der Computer-Technologie und in der virtuellen Repräsentation von Landschaften deuten an, dass es möglich sein sollte, das Verhalten von Menschen in einem kontrollierten Experiment verlässlicher als in der Natur zu beobachten. Visualisierung erlaubt, sowohl die jetztige Landschaft als auch zukünftige Veränderungen darzustellen. Damit erhält man ein Werkzeug zur systematischen Ermittlung des Verhaltens der Menschen auf veränderte Landschaften.

Eine solche Visualisierung wäre jedoch unbewohnt, da für die veränderten Landschaften keine Daten über das Verhalten der Menschen existieren. Die *Modellierung mit Hilfe von selbständigen Agenten*, die auf Ansätzen aus künstlichem Leben und Computersimulationen basiert, ermöglicht eine Bevölkerung der virtuellen Welt durch regelbasierte Agenten, welche die echten Menschen ersetzen können.

In dieser Arbeit wird eine *Multi-Agenten-Simulation* zur Modellierung von Touristen (hauptsächlich Wanderern) präsentiert. Das Ziel ist, mit diesen Agenten die virtuelle Welt zu beleben, und durch sie verschiedene Entwicklungszenarien beurteilen zu lassen. Gleichzeitig wurde die Gelegenheit genutzt, um neue Konzepte der Computer-Technologie für Simulationen zu erkunden.

Bisher gingen die Anstrengungen für Simulation von landschaftsplanerischen Aufgaben in die Richtung von grösseren Massstäben (Stadt oder Region), und verwendeten relativ abstrakte Konzepte der Landnutzung oder Entwicklung von Verkehr und Ökonomie. Allerdings scheinen genau diejenigen planerischen Entscheide den grössten Einfluss auf das menschliche Verhalten zu haben, die entweder sehr klein sind oder einen lokalen Einflussbereich haben.

In diesem kleinen Massstab ist der visuelle Eindruck und die allgemeine visuelle Qualität sehr wichtig. Dies ist vor allem in Touristenregionen der Fall, wo geziehlt mit der optischen Qualität der Region geworben wird.

Allerdings existieren neben den Bedürfnissen der Bevölkerung auch landschaftsplanerische Regelungen, welche ebenfalls die Attraktivität einer Region verändern können. Dabei sind ein Ändern der Landnutzung aufgrund geänderter Regelung, Anpassungen in der Forstwirtschaft, das Schliesser oder Neueröffnen von Seilbahnen oder Sesselliften, aber auch der Bau von neuen Ferienhäusern. Da jede dieser Änderungen komplexe Auswirkungen auf die Tourismusindustrie haben kann, wurden entsprechende Testszenarien gewählt.

Unser Ansatz ist, jeden Touristen als individuellen Agenten zu modellieren. Eine synthetische Bevölkerung, die den demographischen Verhältnissen der tatsächlichen Touristen entspricht, wird erzeugt. Diese virtuellen Touristen erhalten Ziele und Erwartungen, die Werten aus der Literatur, eigenen Studien oder, wenn nicht anders verfügbar, Schätzungen entsprechen. Diese Ziele und Erwartungen sind

individuell, was heisst, dass sie sich von Agent zu Agent unterscheiden können.

Diese Agenten erhalten Tagespläne und werden in die Simuation gesetzt, ohne dass sie ein Wissen der Umgebung haben. Die Agenten führen diese Pläne aus und erhalten beim Wandern Rückmeldung der Umgebung. Am Ende jedes Durchganges werden die Taten der Agenten mit ihren Erwartungen verglichen. Falls die Resultate eines bestimmten Planes nicht den Erwartungen entsprechen, wird dieser für darauffolgende Durchgänge geringfügig modifiziert. Diese Modifikation basiert auf den eigenen Erlebnissen der Agenten, aber, je nach eingesetztem Lernmuster, auch auf der Erfahrung der anderen Agenten.

Diese Arbeit hat einen computertechnologischen Schwerpunkt. Nebst der Präsentation des Frameworks und seiner Module werden die techischen Aspekte der Fussgängersimulation näher betrachtet. Durch den Einsatz von neuen Methoden, die speziell für Fussgänger in den Alpen entwickelt wurden, wird es möglich, eine Fläche von $15km^2$ oder mehr auf einem einzigen Computer zu simulieren.

Diese neue Simulation funktioniert auch in der Nähe von Hindernissen wie Gebäuden. Die Simulation des Inneren eines Gebäudes wird ebenfalls möglich, was die Verwendung desselben Frameworks für z.B. Evakuierungssimulationen erlaubt.

Das Framework einer Verkehrssimulation wurde in dieser Arbeit weiterentwickelt. Es verwendet die selben Module, aber eine neue Art der Verbindung zwischen diesen. Die Module werden nicht länger durch Dateien verbunden, sondern über das Netzwerk. Dadurch wird ein Reagieren auf Ereignissen in Echtzeit möglich.

Es ist auch möglich, neue Module direkt in das bestehende Framework zu integrieren, ohne dieses zu verändern. Module, die die Daten der Simulation verarbeiten oder aufbereiten, werden nun möglich. So können mehrere Module gemeinsam die Aspekte der Simulation berechnen. Diese Kommunikation zwischen den Modulen wird detailiert betrachet.

Part I

A Framework for Modular Multi Agent Simulations

Chapter 1

Introduction

1.1 General context

The visual quality of the landscape is an important but often neglected factor in landscape and environmental planning. Especially for the recreation potential, the quality of the visual landscape plays a key role. Up to recently, the judgment of human preferences about the landscape and landscape elements was based on the existing landscape, either by placing people or monitoring people in the real environment, or by exposing test persons to visual surrogates such as photographs of the real landscape.

Recent developments in the virtual representation of landscapes suggest that it might be possible to reliably observe the behavior and deduct landscape preferences of people in a more controlled laboratory environment as it is provided through new developments in computer and projection technology. Such visualizations permit to show the visual landscape both at present and at potential future situations, giving a tool to systematically investigate how people react to changes in the landscape.

However, such a visualization of a changed landscape would be unpopulated, since no data about human behavior existed for this changed landscape. Autonomous agents modeling, which is based on concepts from artificial life and computer simulations, enables one to populate the virtual world with rule-driven agents which can act as surrogates for real humans.

The project "Planning with Virtual Alpine Landscapes and Autonomous Agents" used a multi-agent simulation to model the activities of tourists (primarily hikers). The goal was to have these agents populate a virtual world, where they are able to evaluate different development scenarios. Such scenarios include the question of re-forestation of meadows, or the summer use of chair lifts and the like. Left to themselves, many areas in the Swiss Alps would be covered by dense forest; it seems, however, that most hikers would prefer a more variable landscape. Many people, in particular families with children or people with health limitations, like mechanical aids to bring them nearer to the top of mountains.

The aim of this project was to implement a multi-agent simulation of tourists hiking in the Alps in order to investigate the achievable level of realism. At the same time, the project was used to explore general computational implementations of mobility simulations.

As many planning problems focus on processes that evolve over time in a complex environment, it is often difficult to evaluate the long term implications of planning decisions. Computer simulations have

long been used as a method for evaluating proposed future scenarios in planning (Timmermans, 2003).

However, most simulation efforts in spatial planning have focused on large spatial scales (such as at the city and regional levels) and on relatively abstract concepts (such as land use patterns, traffic and economic development), while one can argue that the planning decisions that have the most impact on individual citizens tend to be either at a relatively small scale or have very local impacts.

At these small scales (such as the sub-watershed or village), visual elements and the overall visual quality of the proposed planning intervention are extremely important. This is particularly true with areas dependent on tourism, which are often promoted based on their scenic qualities. Aesthetics are hard to quantify and therefore hard to integrate into a computer model. Because of the persisting perception that visual quality is too subjective a concept, it has largely been ignored in planning models and simulation. This is, in our opinion, a large oversight, as there is a whole class of planning problems where aesthetics are important, yet visual quality questions are largely left to designers at the individual project scale, with little consideration for how individual aesthetic choices combine to impact the larger landscape.

Our project integrates these aesthetic qualities with other factors such as availability of recreational opportunities, congestion and service levels using an agent based approach. We focus on their impacts on tourism, in particular summer hikers. Others (e.g. Gimblett, 2002) have used agent-based approaches to model tourists, but their focus has been primarily been on congestion issues. While congestion is a problem for the busiest and most famous tourism areas, it is our contention that the encroachment of development and changes to the management of the landscape have the potential for far greater impacts on the attractiveness of a given area for tourists.

The specific test site is a valley in the Gstaad-Saanenland region of south-western Switzerland. The communities of Schönried and Saanenmöser are at the two ends of the site; their economies are highly tourism dependent. While the primary tourism draw to the area used to be winter skiing, long term climate change is forcing the community to focus its efforts on building up a more diversified tourism economy. This includes capitalizing on its already strong reputation for summer hiking. The landscape is a mixture of pasture and coniferous forests, dominated by Norway Spruce (Picea abies). The test site is characterized by significant topography and is considered ideal for walking and hiking (see Figure 1.1). The trails are very accessible to a wide range of hiking abilities due to the summer operation of one chair-lift and two gondolas. In the high season, the area is busy with hikers and walkers, who easily fill the two main parking lots in Schönried.

A recent study in the area (Müller and Landes, 2001) identified that the biggest attraction for summer tourists are the area's scenic qualities. Hiking and walking is the primary recreational activity in the summer months. The focus on views was confirmed by our own study (Cavens and Lange, 2003), which confirmed that views and landscape variety are the most important factors that influence hikers in their choice of hiking routes.

In addition to the community's desire to diversify its recreational economy, there are landscape policy issues that have the potential to change the desirability of the area for summer tourism. These issues include changes to the pattern of the landscape due to changing agricultural policy, shifts in forestry practices, closing of the gondolas and/or chairlifts, and increased holiday home construction. Any of these changes would have complex repercussions for the tourism industry: scenarios to test the agent model will be selected from them.

Figure 1.1: The test site is characterized by significant topography and is considered ideal for walking and hiking. Most hikers like forests, but since these can also block the great view, a mixture is prefered (left). In the picture of our 3-dimensional offline visualizer (right), the virtual representation of the same scenario is shown.

1.2 Modeling Approach Overview

Our approach is to model each tourist individually as an "agent". The approach is adapted from one used in traffic microsimulations (e.g. Balmer et al., 2004). A synthetic population of tourists is created that reflect current (and/or projected) visitor demographics. These tourists are given goals and expectations that reflect existing literature, on-site studies, and, in some cases where sufficient data is not available, are based on experts' estimates. These expectations are individual, meaning that each agent could potentially be given different goals and expectations.

These agents are given "plans", and they are introduced into the simulation with no "knowledge" of the the environment. The agents execute their plans, receiving feedback from the environment as they move throughout the landscape. At the end of each run, the agents' actions are compared to their expectations. If the results of a particular plan do not meet their expectations, on subsequent runs the agents try different alternatives, learning both from their own direct experience, and, depending on the learning model used, from the experiences of other agents in the system.

A "plan" can refer to an arbitrary period, such as a day or a complete vacation period. As a first approximation, a plan is a completely specified "control program" for the agent. It is, however, also possible to change parts of the plan during the run, or to have incomplete plans, which are completed as the system goes.

After numerous runs, the goal is to have a simulated system that, in the case of a status quo scenario, reflects observed patterns in the real world. In this case, this could, for example, be the observed distribution of hikers across the study site over time. In the case of a forecasting scenario, it is then expected that the emergent properties of the simulation system change as a reaction to the modified scenario.

Any mobility simulation system does not just consist of the mobility simulation itself (which controls

the physical constraints of the agents in a virtual world), but also of modules that compute higher level strategies of the agents. In fact, it makes sense to consider the physical and the mental world completely separately (Fig. 1.2). The modules used in this project are:

- **Mobility Simulation**. The mobility simulation takes care of the physical aspects of the system, such as movement of the agents, interaction of the agents with the environment, or interactions between the agents (see Chapter 3).
- **Route Generator**. It is not enough to have agents walk around randomly; for realistic applications it is necessary to generate plausible routes (see Chapter 4).
- **Activity Generator**. Being able to compute routes, as the route generator does, only makes sense if one knows the destinations for the agents. A new technique in transportation research is to generate a (say) day-long chain of activities for each agent, and each activity's specific location. For our hiking simulations, possible activities include: be-at-hotel, visit-a-restaurant, etc. At this point, activity generation was done manually to fit the need of the different sensitivity studies (Chapter 9).
- In the **Synthetic Population Generation Module**, the agents are generated. This includes demographic attributes to each agent, such as age, gender, income, etc. In the future, this should follow some demographic information about the tourist population in the area of interest; at this point, this is entirely random.
- The **View Analyzer Module** describes to the system what individual agents "see" as they move through the landscape. The agents field-of-view is analyzed, and events are sent to the system describing what the agent sees. This role of module is discussed in Chapter 2; the module itself will be described in detail by Cavens (in preparation).
- **Agent Databases**. The strategy modules so far are capable to generate plans, and the mobility simulation is able to store exactly one plan per agent and execute it. There should, however, also be a place where plans and in particular their performance is memorized, so that memory can be retrieved later. Several different versions of agent databases were developed within the project "Planning with Virtual Alpine Landscapes and Autonomous Agents"; one example is presented in Chapter 5.
- **Visualizer Modules** are built so that they directly plug into the live system. The simulation sends agents' positions to the visualizer modules, which allows to look at the scenario from a bird's eye view and observe how the agents move. There is a chapter (6) which shows some of these visualizer modules in depth.
- **Helper Modules**. These modules glue the other modules together. One is the **TCP Multicast (TCP-MC) Module**, which is used to provide reliable point-to-multipoint communication in the framework. Additionally, there are the **Recorder Module** and the **Player Module**, which are used to record and playback a events stream, respectively. A player module can read the file and send the data stream to the visualizer (or any other module) exactly the same way it would come from the mobility simulation directly. Finally, in order to deal with data conversion issues, it is also possible to pipe the data stream from the simulation through the recorder directly to the player and from there to the viewer. These modules are described in Chapter 7.

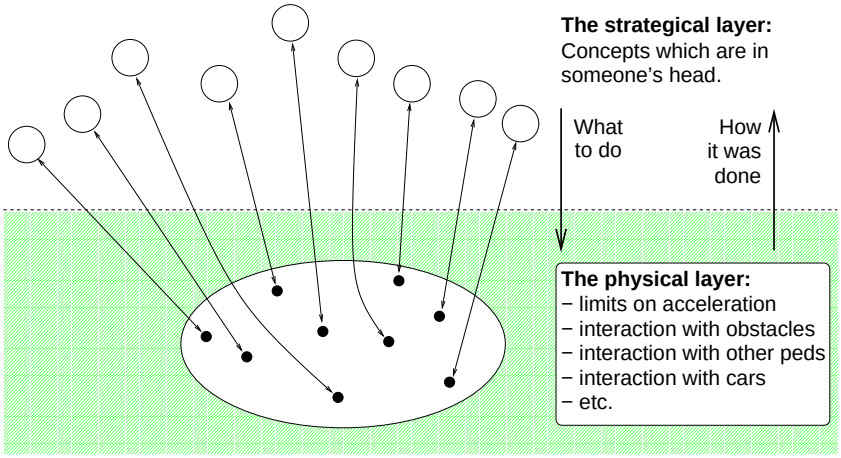


Figure 1.2: Any simulation system does not just consist of the pedestrian simulation itself (physical layer), but also of modules that compute higher level strategies of the agents (mental or strategic layer).

As mentioned earlier, every agent starts with a plan. Such an initial plan consists of a series of activities. For example, if the period of interest is a day, then such an initial plan might refer to a specific hike. To do this, the agent chooses *activity locations* it wants to visit, like hotel, peak of mountain, restaurant etc.

This chain of activity locations is then handed over to the *routing* module, which calculates the routes between activities according to the information available. This information can be static and global, like shortest path information based on the street network graph. Also information that is local to the agents memory and might be uncertain can be used.

The mobility simulation then executes the routes. The agent experiences the environment and sends its perception as events (see Chapter 7) to the other modules.

From here on, the system enters the *replanning* or *learning loop*. The idea is that the agents go through the same period (e.g. day) over and over again. During these iterations, they try to accumulate additional information, and to improve their plan. More details on how this learning mechanism works is given in Chapter 2.

These aspects of the simulation concern the modeling of human intelligence, which is an unsolved (and maybe unsolvable) problem. Yet, one should recognize that for our simulations it is not necessary to model individual people correctly, but it is sufficient to obtain correct *distributions* of behavior. Our approach should be considered as a first step into that direction.

Core functionality aspects of the approach will be illustrated in Chap. 9, where the reaction of the simulation system to strong stimuli will be described. This includes, for example, "spreading" of the hikers to avoid too many other hikers, reaction to different preferences by different agents, or intraday re-planning caused by unexpected rain.

This work has a strong computational component. Besides the "framework" Chapter 2, computational aspects of the mobility simulation are discussed in Section 3.4. Chapter 7 concentrates entirely on the communication between the modules. That chapter discusses many different protocols, and important

performance results of some of those protocols. Chapter 8 is dedicated to the discussion of synchronization methods. These methods become particularly important when different parts of the simulation run with different update steps and with different computational speeds. An example is the coupling of the hiking simulation with a (very simple, but much faster) cablecar simulation.

Chapter 2

Framework

2.1 Introduction

Multi-agent simulations, where each agent is modeled individually, allow to look at the problem of visual quality of a landscape from a point of view of a person walking in an area. As said before, any such simulation system does not just consist of the pedestrian simulation (also referred to as *mobility simulation*) itself, but also of modules that compute higher level strategies of the agents.

This kind of modularization has in fact been used for a long time (see, e.g. Ortúzar and Willumsen, 1995, who describe “trip generation”, “trip distribution”, “mode choice”, and “route assignment”). A major and very important difference to the traditional approach is that it is now possible to make all the modules completely microscopic on the level of the hikers. Microscopic means that in all modules each individual agent retains its identity, including, for example, gender, age, income, physical fitness, or remaining energy level. It is, we hope, easy to see how such information can be used for better modeling.

Traditional implementations of transportation planning software, even when microscopic, are monolithic software packages (Rickert, 1998; Babin et al., 1982; PTV www page, accessed 2004; Salvini and Miller, 2003). By this we do not dispute that these packages may use advanced modular software engineering techniques; we are rather referring to the user view, which is that one has to start one executable on one CPU and then all functionality is available from there.

The disadvantage of that approach is twofold: All the different modules add up in terms of memory and CPU consumption, limiting the size of the problem. And second, although the approach is helpful when starting as one software project, it is not amenable to the coupling of different software modules, developed by different teams on possibly different operating systems.

A first step to overcome these problems is to make all modules completely standalone, and to couple them via files. Such an approach is for example used by TRANSIMS (TRANSIMS www page, accessed 2005). The two disadvantages of that approach are: (1) The computational performance is severely limited by the file I/O performance. (2) Modules typically need to be run sequentially. Each module needs to be run until completion before starting the next module. For example, the routing module can only be run before or after the mobility simulation. This implies that agents cannot change their routes

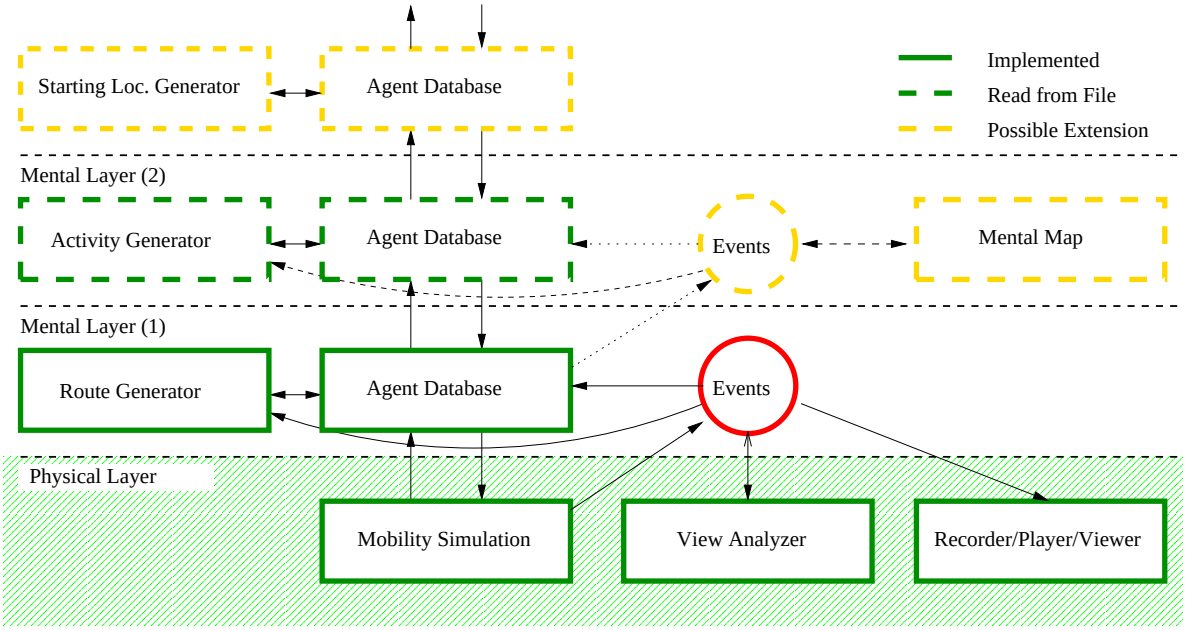


Figure 2.1: The complete simulation system, according to the modules used. It is divied into several layers: the physical layer, and mental layers. The modules communicate with each other using specialized data connections or an general, broadcast channel (events).

while the mobility simulation is running.

An alternative is to use a database system in order to couple the system (e.g. Waddell et al., 2003; Raney and Nagel, 2004b). In that case, however, it turns out that the centralized database quickly becomes a bottleneck, since each transaction between modules has to go through that one database. Therefore, each additional module slows down the system.

Our own approach to this type of simulation is to couple the modules by messages (Fig. 2.1). In this way, each module can run on a different computer using different CPU and memory resources. It is even possible (as is with file-based interfaces) to make the modules themselves distributed; for example, we have mobility simulations (for traffic) which run on parallel Myrinet-equipped clusters of workstations.

It is clear that the message-based approach allows real-time interaction between the modules: for example, if an agent gets wet from unexpected rain, the agent database module can react to this new situation and submit new routes or activities during a simulation run (see Chapters 5 and 9).

On simulations with many agents, issues such as bandwidth usage, packet loss, and latency become increasingly important. We use different network protocols and implementations tailored to specific requirements of inter-module communication (see Chapter 7).

2.2 The Structure of the Framework

2.2.1 Physical Layer Modules

As mentioned above, the *mobility simulation* takes care of the physical aspects of the system, such as interaction of the agents with the environment or with each other.

For our simulation approach, the assumption is that the mobility receives plans as input, and emits events as output. A *plan* contains the agent's intentions, which with respect to mobility mean where the agent wants to go to, and which path it intends to take.

A plan is sent from the agent database to the mobility simulation. Between the agent database and the mobility simulation is a special communication channel, the so-called *braincom channel*. This is a direct (point-to-point), reliable TCP connection, and is initiated by the agent database. This connection and more details about the communication channels used in this framework can be found in Chapter 7.

The plan sent through this channel is encoded as an XML fragment:

```
<request agent="AGENTID">
  <plan>
    <act end_time="07:00.0" x="588444.997878" y="150258.012133"/>
    <leg mode="ped" from="FROMNODE" to="TONODE">
      <act>NODE 1</act>
      <act>NODE 2</act>
      <act>NODE 3</act>
      <act>NODE 4</act>
    </leg>
  </plan>
</request>
```

The simulation parses this XML plan and translates the instructions into its internal data structure. In this example, the agent is expected to be already at the "hotel". It should stay there until 7 o'clock (`<act end_time="07:00.0" [..] />`). After that, it has to walk (`mode="ped"`) to a new location. The path is given by a sequence of nodes (NODE 1, NODE 2 .. NODE 4). Once at this new location, the agent waits for new instructions.

The simulation executes the plans of all agents simultaneously. If two agents want, according to their plans, to be at the same place at the same time, the physical interaction of the simulation will prevent that and compute physically plausible solutions instead. For example, there are physical restrictions to how many agents can be simultaneously on a path, or how they behave when hiking groups oppose each other. Also, restaurants can be full, etc. These results are encoded in the *events*. Events come together with a time stamp and the agent id number:

```
<event type="hitbyraindrop" id="42" time="100" x="588162" y="150664"
```

All events are sent to an *events channel* (circles in Fig. 2.1). Once again, the technical details of this communication channel are explained in Chapter 7. However, there is one important difference to the braincom channel seen before: the events channel is a *broadcast channel*. This means that every message sent to this channel is broadcast to all receivers. There is no distinction between sender and receiver, every sender does receive every message automactically.

Every module is connected to the same events channel. This basically says that every event sent by a module is received by every module.

Typical events are: position (where an agent is at the moment), pedpressure (if there are too many agents at the same place), enterlink/exitlink (the agent has entered/left an link segment), reached_destination (sent as soon as the agent has no more plan elements in its cache). A position event looks as follows:

```
<event type="position" id="42" time="100" x="588162" y="150664"/>
```

2.2.2 Mental Layer Modules

Routes

As said before, it is not enough to have agents walk around randomly; for realistic applications it is necessary to generate plausible routes. In terms of graph language, this means that agents need to compute the sequence of links that they are taking through the network.

A typical way to obtain such paths is to use a best path algorithm (Dijkstra, 1959). That algorithm uses as input the network link travel times plus the starting and ending point of a trip, and generates as output the fastest path. For traffic applications, one needs to make the costs (link travel times) time dependent, meaning that the algorithm can include the effect that congestion depends on the time-of-day: Trips starting at one time of the day will encounter different delay patterns than trips starting at another time of the day.

The algorithm used for the hiking simulation values individual links based on generalized costs, such as different views or different temperature levels. For example, a physically fit person might prefer a route that is steeper, while a physically less fit person might get a route that includes more possibilities to rest. See Chapter 4 for more details.

Activities

Being able to compute routes, as the route generator does, only makes sense if one knows the destinations for the agents. A new technique in transportation research is to generate a (say) day-long chain of activities for each agent, and each activity's specific location. For our hiking simulations, possible activities include: be-at-hotel, visit-a-restaurant, etc. In a hiking simulations, unlike typical travel simulations, the travel (= the hike) connecting these activities will generally not be seen as a negative but as a positive part of achieving their goals and expectations. As a result, for our hiking simulation the route generator module needs to connect the activities in a way such that the connecting route reflects the expectations of the agent in terms of aesthetics, difficulty, as well as travel (hiking) duration.

This module is not yet implemented, neither for the transportation simulation nor for the hiking simulation, but is an area of active research and development in our group. In the meantime, activities are constructed manually and read from a file instead.

Initial condition

The learning process needs to be started somehow. For this, an initial population needs to be generated, and one needs to compute initial activities and routes. For the generation of the initial population, a *Synthetic Population Generation Module* is used. This uses some aggregated information, such as known age or gender distribution of tourists, and generates individual agents from this, which have individual age, gender, income, etc. This module is not yet operational, and therefore agents with random attributes are generated.

Initial activity plans are constructed from some global knowledge of the area, i.e. agents are told where attractive view points or where restaurants are. For example, for some of our illustrative studies (Chap. 9), each agent was given *several* plausible activities plans, each one forming a possible plan for a day.

Learning loop

The above modules will in general not produce consistent results. For example, an activity plan can be planned under the assumption of empty hiking paths, but during the execution many other hikers may be encountered. If the hiker wants to be alone, the traveler would probably not select the same activity plan again. This is modeled by feedback iterations: Every agent generates an initial plan, all plans are simultaneously executed, some agents modify their plans, all plans are again simultaneously executed, etc., This resembles at the same time a numerical analysis relaxation method and a human learning method, and it can indeed, with some caution, be interpreted both ways. Analysis of the precise mechanics of these feedback iterations, including, for example, plausible stopping criteria, is a subject of ongoing research.

Agent database

One issue with the approach described so far is that the agents to not have a memory of past performance, which makes the architecture brittle. For example, if the router gives the agent a bad route, then the agent is nevertheless forced to execute that bad route many times, until the agent is again picked for re-planning. A considerable improvement terms of robustness can be reached when the agent obtains memory where it keeps multiple plans around; the agent samples scores for each strategy; and the agent selects between strategies according to the scores (Raney and Nagel, 2003). In this way, the agent architecture is exploited much better. The agent database developed by Raney (2005) goes into that direction.

How to aggregate knowledge

The two critical questions now are (1) how to accumulate, store, and classify that information, and (2) how to come up with new plans. Both questions are related to (artificial) intelligence, and we are certainly far away from answering them in their entirety.

As said before, the mental modules extract all their knowledge from the events stream. However, that

approach allows a wide variety of methods how information is actually extracted and used. Some examples are:

- As one can see in Fig. 2.1, agent databases are associated with each level of the planning hierarchy (e.g. routes, activities). They listen to the events emitted by the mobility simulation, and use these events to calculate a score for *each route plan* (or activity plan, respectively) once it has completed. It extracts, for the plan that was just executed, the performance-relevant aspects from the events, such as the times when activities are started or ended, when hikes are started or ended, or how often pleasant ("nice view") or unpleasant ("rain") experiences are recorded.

- Also the route generator module listens to the same stream of events. In its current implementation, it generates a global view of the network. For this, it accumlates all events for all links on a *link-specific basis*. From this, it will extract global knowledge such as typical hiking times on links, the frequency of pleasant or unpleasant experiences on the link, etc.

- Another example of a possible mental module is a mental map. This is easiest to explain inf one assumes that that module takes care of just one agent. The mental module follows the actions of the agents over multiple days, recording what the agent sees, feels, etc. Note that it is possible to extract events for just a single agent from the events stream. From this information, the module builds a "mental map" of the environment, including, for example, the locations and quality of sites that were visited, the locations of sites that the agent has seen but not visited, or the quality of certain hiking paths. In contrast to the above route generator, such a mental map would be incomplete; information could, for example, only be recorded when it is interesting enough.

  After the agent has accumulated some information in this way, it could become explorative and creative. For example, it could visit sites that it has only viewed, or knows from guide books. In this way, the explorative accumlation of experience by an agent could be modelled, with the ultimate goal to test if this yields significant differences in the emergent behavior of the system when compared to "global" mental modules. For transport systems, for example, it is speculated that travelers have a tendency to stick with parts of the system that they know, even if this yields sub-optimal solutions. – A prototype of such a module was implemented into our system by Kistler (2004).

  Such modules can also take care of groups of agents, having separate mental maps for each agent. It is also possible to run several copies of such a module, each copy on a different CPU and taking care of different sub-groups of agents. In this way, it should be possible to simulate mental maps for somewhat larger groups of agents without compromising computational performance. Although this was not tested in practice, the framework should be able to support this.

- It is possible to insert mental layer modules that observe the agent on its path through the virtual environment without storing any information at all. As soon as the module detects an option that might yield a better score that the current plan, e.g. using a shorter path, or entering a restaurant, it notifies the agent in the simulation. Using this mechanism, agents are able to react to unpredicted changes in the environment, like weather changes or congestion (see Chapter 5).

It should be noted that the distinctions between these modules are not sharp. For example, an agent database may run out of memory if it memorizes as separate entities plans that differ only in small

details; in that case, the agent database might have to start building a mental map of the world. In this case, it becomes similar to the activity generation module as described above.

As said before, these aspects of the simulation concern the modeling of human intelligence, which is an unsolved (and maybe unsolvable) problem. Yet, one should recognize that for our simulations it is not necessary to model individual people correctly, but it is sufficient to obtain correct statistical *distributions* of behavior. Our approach should be considered as a first step into that direction.

2.2.3 Other/Helper Modules

Visualizer Modules

There are some modules that do not belong to a layer of the framework. Visualization of processes in one or multiple of the modules can be done for each module separately. It is, however, sometimes needed to display the state of more than one module at once, e.g. pedestrian positions from the mobility simulation together with route calculations from the router. A visualization module can connect to the event stream and filter those events it is interested in.

Our Visualizer Modules (see Chapter 6) are stand-alone applications that connect to the simulation system via the computer network. Visualizer modules are built so that they directly plug into the live system, which allows to look at a running simulation in "real time". The simulation sends agents' positions to the viewer, which allows to look at the scenario for example from a bird's eye view and observe how the agents move.

There can be any number of visualizers connected to an event stream, each showing the same scenario from a different perspective, or displaying different accumulations of events. Some or all of the viewers can also show what individual agents see. It should even be possible (although it was not tested) to have separate viewers on separate graphics workstatiosn compute different sections of what agents see, in order to compose a 360-degree panoramic view around a specific agent; a human observer could then be placed in the center of such a projection.

We have implemented two different kind of visualizer modules:

- **Real-time visualizers.** One kind of visualizer module displays a real-time view in 2D or 3D.

  The 2-dimensional real-time visualizer is suited for situations where a lot of detailed information is needed, for example while debugging.

  The 3-dimensional real-time visualizer has been implemented (by Duncan Cavens, see Section 6.3), as one of our overall project goals is to integrate decisions based on visual impression. The user can move independently of the agents or can attach the camera viewpoint to a specific agent and see the landscape through the eyes of the agent.

- **Offline visualizers.** The other kind of visualizer module renders a more realistic image of the scene, but runs too slowly for a real-time representation. For example, Fig. 6.13 is rendered using the offline visualizer.

All the visualizer modules connect to the simulation using the same protocol.

Recorder, Player

It is also possible to attach a recorder module to the events stream. This recorder module then writes the stream to a file. Later, a player module can read the file and send the data stream to the visualizer exactly the same way it would come from the simulation directly.

Note that this still makes it possible to connect multiple viewers to the scene. In this case, an additional advantage over the online view is that one can play and step forward and backward through time, while all viewers are synchronized in time. This makes it, for example, possible to observe the overall scenario in one viewer, while the other viewer displays what one agent sees, and it is possible to step through some interesting sequence multiple times.

Finally, in order to deal with data conversion issues, it is also possible to pipe the data stream from the simulation through the recorder directly to the player and from there to the visualizer. This is useful for example if a new kind of protocol for coupling the visualizer modules is tested. Only the code in the player and the visualizers has to be changed, the other modules can remain untouched.

There are several other reasons why the simulation does not directly write the events files itself:

- The simulation can be parallelized, and we need the events output of all the instances in a single file.
- Writing to a remote file system (i.e. NFS) can be very slow.
- By splitting the writing functionality from the simulation, we are able to change implementation details for all of the simulation modules without needing to modify their code. For example, one could use a database to store the events instead of a file.

View analyzer module

Using the visualizer modules, human operators of the system are able to see the simulated world. The agents that are simulated need a way to perceive the landscape as well. Rather than using a single visual quality model, our approach has been to give the agents the ability to see the landscape, and integrate their visual experience into the factors that are evaluated by the agent database modules. This allows us to model sequences of views, and provides a lot of flexibility in terms of exploring the importance of various visual parameters.

It turns out that this can be done by re-using the 3d real-time visualizer that is also used for humans to interact with the computer system. However, instead of displaying a view on the screen after it is computed, the video memory is read out again in order to determine what a individual agents “see” as they move through the landscape.

The *View Analyzer Module* (developed by Cavens, in preparation) exploits the capabilities of modern 3D graphics hardware to quickly perform visibility calculation. Using a similar technique as that described by Bishop et al. (2001), objects are rendered in perspective using false colors. These colors are assigned based either on unique objects or on logical groupings (such as stands of trees.) As the agents move through the landscape, the scene is rendered from the viewpoint of each individual agent. The rendering process produces a color image and a depth buffer, which is a natural byproduct of the rendering process

used in current graphics hardware and describes how far away an object is from the viewpoint. As these visibility calculations are performed on specialized hardware, the process is able to scale well to complex scenes with little effort from the user perspective.

The agent's field-of-view is analyzed, and events are sent to the system describing what the agent sees. Depending on the needs of the agent database modules, these events either list all of the individual objects (houses, restaurants, forest stands, or individual trees) or return synthesized information about particular visual metrics (such as enclosure, percentage of view that is non-vegetative, etc.).

While the process is considerably faster than other visibility approaches, it is not quick enough for our purposes. At a frame rate of 15+ frames per second, it quickly becomes the bottleneck for the entire simulation system.

Duncan Cavens is exploring two different approaches to eliminate this bottleneck:

- pre-rendering the landscape using a grid of viewpoints. At each viewpoint, a series of view slices (each comprising a view angle of 15 degrees) is computed and analyzed. During the simulation, the nearest viewpoint to a given agent is selected, and these slices are reassembled to reflect the view direction of the agent.
- distributing the view modules across a cluster of rendering machines, with each one being responsible for a subset of the agents.

Both of these approaches offer considerable opportunity for speed improvement, and both are facilitated by the modular structure and communication strategies of the entire simulation system. It is very easy to swap the module that calculates the views for every agent at every time step with the pre-rendered implementation.

The question now is how to introduce this in the framework presented so far. Clearly, what an agent sees is a physical property, depending on the agent's position in the landscape and its viewing direction. These aspects should thus be included into the simulation of the physical layer. On the other hand, the *interpretation* of what the agent sees belongs into the mental layer.

It turns out that such a view-analyzer module can be entered into the events stream without major modifications of the framework design. Recall that it is already possible to follow the views of an individual agent in the 3d-visualizer, based on events information by the mobility simulation. The only additional element now is that the results of the view analysis are *also* fed back into the *same* events stream. The resulting events stream is then a combination of events generated by the mobility simulation, and secondary events generated by the view analyzer. In this way, it is possible to have several modules jointly compute different aspects of the events stream, which are then picked up by the mental modules.

For example, the view-analyzer module reads position events like this as input:

```
<event type="position" id="42" time="100" x="588162" y="150664"/>
```

and emits events such as

```
<event type="vis_analysis"
```

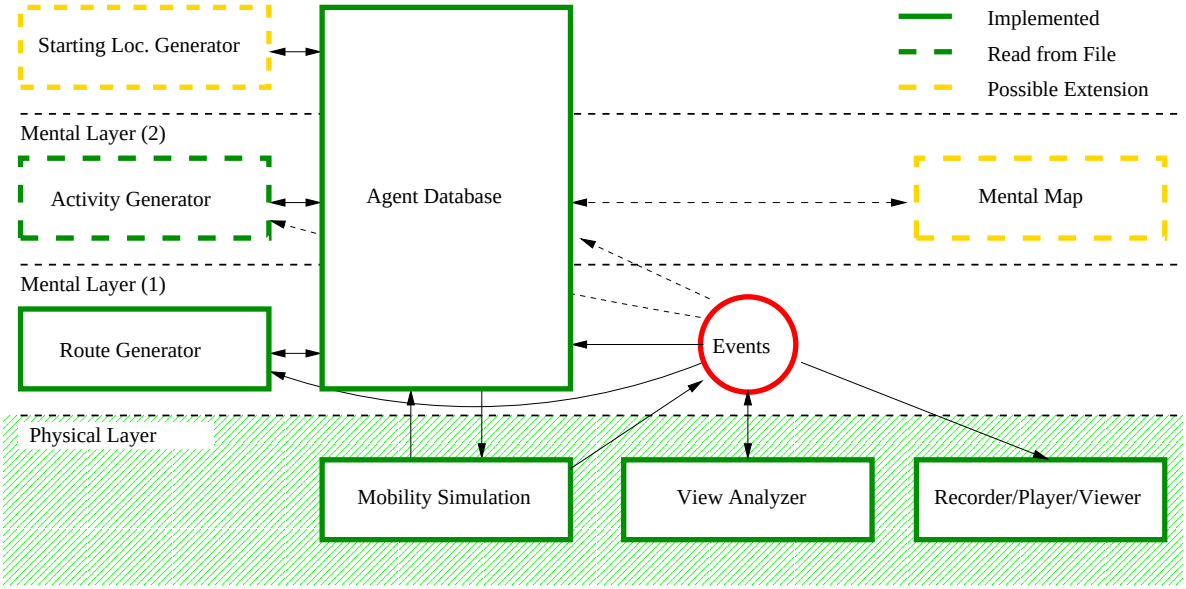


Figure 2.2: Alternative approach to a full simulation system, where the multi-layer approach to the agent databases of Fig 2.1 is replaced by a single monolithic agent database. Since at this point the higher level modules are not fully developed, the current implementation can be extended both according to Fig 2.1 and according to this figure.

```
        id="42" time="100" x="588162" y="150664"
        sky="40%" forest="30%" ground="30%"
/>
```

The mental modules will then assign scores to that, based on the preferences of that specific agent. For example, it is able to determine whether the agent is in an open area or not, depending on how the ratio between “sky” and “forest” is.

2.3 Discussion and Outlook

It is important to note that the task of the simulation of the physical system (= the mobility simulation) is simply to send out events about what happens; all interpretation is left to the mental modules. In contrast to most other simulations in the area of mobility research, the simulation itself does not perform any kind of data aggregation. For example, link travel times are not aggregated into time bins, but instead link entry and link exit events are communicated every time they happen. If some external module, e.g. the route generator, wants to construct aggregated link travel times from this information, it is up to that module to perform the necessary aggregation. Other modules, however, may need different information, for example specific progress reports for individual agents, which they can extract from the same stream of events. This would no longer possible if the simulation had aggregated the link entry/exit information into link travel times.

Despite this clean separation – the mobility simulation and the modules in the physical layer compute

"events", all interpretation is left to mental modules – there are conceptual and computational limits to this approach. For example, reporting everything that an agent sees in every given time step would be computationally too slow to be useful. In consequence, some filtering has to take place "at the source" (i.e. in the simulation), which corresponds to some kind of preprocessing similar to what real people's brains do. This is once more related to human intelligence, which is not well understood. However, also once more it is possible to pragmatically make progress. For example, it is possible to report only a random fraction of the objects that the agent "sees". Calibration and validation of these approaches will be interesting future projects.

Up to now, only the modules in the lowest mental layer (see Fig. 2.1) are implemented. The modules of the second mental layer and above were replaced by files, and are not able to receive events. Therefore, there is no need to distinguish between different events channels.

However, it is important to mention that in a fully developed framework, from the view of every mental layer, all the layers below are considered as one single (physical) layer. For example, the agent database in the second mental layer does not know anything about the concept of routes. This is encapsulated in the layers below.

Given this, it would be reasonable to use different events channels for each pair of layers. For example, between the physical and the first mental layer, information about nodes is exchanged. An events channel between the first and second mental layer would carry information regarding activities. This concept is already used inside the mobility simulation, which receives node-based instructions. However, internally, it passes the nodes to another level that breaks these up into finer grained *link segments*, which in turn are defined as single path where pedestrian dynamics are applied (see Chapter 3).

Therefore, it makes sense that modules in each layer aggregate the information they are able to understand, and retransmit this into the event stream one level above. For example, the route generator listens to timestamps when an agent visited certain nodes. It could generate information about when an agent performed certain activities.

The possibility to use one single agent database still exists. The agent databases in Figure 2.1 can be replaced by one agent database that spans all mental layers (Figure 2.2). This is what the traffic project uses and what has proven to work in the past. Since most of the modules presented seem be cover to some extent functionality from a different mental layer (e.g. the mental map module calculates its own route in order to determine feasible activities), this model is also simpler to implement.

Part II

The Modules of the Framework

Chapter 3

Mobility Simulation

This chapter discusses which kind of simulation approach is suitable for the mobility simulation, and what model was finally selected. The model is modified for our particular purpose, i.e. for hiking in the Alps rather than crowd or panic simulations in small enclosed spaces. The modified model is then calibrated with real-world data.

The aim of this project is to implement such a simulation in order to investigate the achievable level of realism. At the same time, the project is used to explore general computational implementations of mobility simulations. As it turns out, mobility simulation systems generically consist of at least two components: the simulation of the physical system, and the simulation of the strategic decisions of the agents (e.g. Ferber, 1999, Chap. 4). While the former is concerned with physical aspects, such as limits on acceleration or speed, or the physical interaction with other agents, the latter is concerned with the strategic or mental decisions of the agents.

The fact that both of them, plus their interplay, are important, has often been neglected in the past – either people coming from physics or similar areas were concerned about the physical simulation but neglected strategies, or people coming from artificial intelligence or similar areas concentrated on simulating "intelligent" agents but neglected to implement a realistic representation of the physical system (e.g. Ferber, 1999, Chap. 4).

This chapter concentrates on the physical simulation, also called mobility simulation. It first reviews which models are available for our application, which type of model was selected as a starting point for our work, and the reasons for this choice (Sec. 3.1). The main problem with existing pedestrian models for our purposes is that they do not work when confronted with such large areas as are necessary to cover a complete hiking area: Covering an area of $(50\ km)^2$ area (necessary to allow hikes of length $25\ km$ in all directions from a central starting point) with cells of $(0.25\ m)^2$ results in 10^{10} cells, which needs at least $40\ GByte$ of memory. This makes the straightforward application of a cellular automata model impossible. Models based on continuous coordinates represent the environment by objects rather than by cells. Again, a straightforward implementation goes beyond available computer memory because there are too many objects. All models have the problem that long-distance path following has never been looked at, at least not to our knowledge.

We extended the chosen model that the agent is now able to follow arbitrary paths (Section 3.2). An

attempt to calibrate the model was taken in Section 3.3. Computational aspects that lead to further improvement in execution speed are presented in Section 3.4.

The model presented so far is based on a graph (street network). This approach leads to some problems inside towns or even buildings. Two possible solutions are compared in Section 3.5.

Finally, two additional mobility simulations are presented: An open-source alternative to the pedestrian simulation, which demonstrates most of the principles, but is much simple to use (Section 3.6) and a cable-car simulation, used to demonstrate synchronization and timing issued (Section 3.7).

This chapter is concluded by a summary.

3.1 Existing approaches

3.1.1 General simulation techniques for mobility simulations

As mentioned above, the mobility simulation takes care of the physical aspects of the system, such as interaction of the agents with the environment or with each other. Typical simulation techniques for such problems are:

- In *microscopic simulations*, each particle is represented individually.
- In macroscopic or *field-based simulations*, particles are aggregated into fields. The corresponding mathematical models are partial differential equations, which need to be discretized for computer implementations.
- It is possible to combine microscopic and field-based methods, which is sometimes called *smooth particle hydrodynamics* (SPH; Gingold and Monaghan, 1977). In SPH, the individuality of each particle is maintained. During each time step, particles are aggregated to field quantities such as density, then velocities are computed from those densities, and then each individual particle is moved according to those macroscopic velocities.
- As a fourth method, somewhat on the side, exist the *queuing simulations* from operations research. Here, particles move in a networks of queues, where each queue has a service rate. Once a particle is served, it moves into the next queue.

For our simulations, we need to maintain individual particles, since they need to be able to make individual decisions, such as route choices, throughout the simulation. This immediately rules out field-based methods. We also need a realistic representation of inter-pedestrian interactions, which rules out both the queue models and the SPH models.

3.1.2 Microscopic simulation techniques for mobility simulations

This leaves the microscopic models. There are several versions of this:

- The agents' movements can be given by *coupled differential equations*. For computer implementations, the differential equations need to be discretized with a time step h. The original differential equations are recovered for $h \to 0$. This is the same technique as applied in molecular dynamics simulations (e.g. Beazley et al., 1995).

- Coupled differential equations use continuous space and time, which are discretized only for computational reasons. Sometimes, it makes sense to use discrete time explicitly in the model formulation. This is for example useful for traffic, where it works well to make the reaction time and the simulation time step coincide (Krauß, 1997). This implies that that in such models, the time step h needs to be selected with care, and the limit $h \to 0$ is *not* meaningful for such models. Such models are sometimes called *coupled map lattices*.

  We are not aware of coupled map lattice models for pedestrian movement.

- Instead of using fixed time steps, it is possible to make the time-step adaptive for each agent: Every time an agent event takes place, the simulation also computes when that agent should be considered again, and puts that next event into an event queue. The simulation proceeds by always taking the event with the smallest time from the event queue, and then processing it. Such models are called *event-driven* or *discrete event simulations*.

  Such simulations are relatively easy to do if the next event of the particle can be determined with ease. This is easy to do in queuing network simulations as already mentioned earlier. It is considerably more difficult in models with collisions in free space, such as pedestrian models. As an example, assume a simulation where particles move in straight lines between collisions. After a collision, a particle needs to compute with which particle it will collide next. This, however, invalidates the scheduled collision of that other particle, which invalidates the trajectory of that other particle's collision partner, and so on. In consequence, discrete event simulations are not easy to use for particle systems in free space, which is probably the reason why we have not found a reference to this simulation technique for pedestrian simulations.

- Analogous to explicitly discrete time, it is also possible to use explicitly discrete space. These models are typically called *cellular automata* (CA, e.g. Wolfram, 1986). Once more, one does not consider the limit $\xi \to 0$ of the discretization constant, but the model is explicitly formulated with a specific spatial resolution in mind.

As said above, we are not aware of examples of coupled map lattices or of discrete event simulations for the area of pedestrian simulations. In contrast, both CA models (Schadschneider, 2001; Blue and Adler, 2001; Hoogendoorn et al., 2001; Dijkstra et al., 2001; Keßel et al., 2001; Galea, 2001; AlGadhi et al., 2001; Paul M. Torrens www page, accessed 2005) and coupled differential equations (Helbing et al., 1997; Hoogendoorn and Bovy, 2002; Tsuji, 2003) are in use. The CA models typically use cells of $(0.25\ m)^2$, which are either empty or occupied by exactly one pedestrian. Pedestrians move between cells; usually, only close neighbor cells are considered. A direct consequence of this is that straight movements can only be represented along the main axes; any off-axis movement in CA models will contain some erratic movement to stay within the cell structure. For crowd simulations, such as for evacuation (e.g. Meyer-König et al., 2001), this is not of major importance. But for our application, where it may even be necessary to follow the eye movements of an individual agent, this makes CA simulations awkward.

This leaves the simulations based on continuous space. Here, models derived from coupled differential equations seem to be much better understood than coupled maps (for which we are not even aware of an example), which is why we decided to use the former. A generic coupled differential equation model for pedestrian movement is the social force model by Helbing et al. (2000)

$$m_i \frac{d\mathbf{v}_i}{dt} = m_i \frac{\mathbf{v}_i^0 - \mathbf{v}_i}{\tau_i} + \sum_{j \neq i} \mathbf{f}_{ij} + \sum_W \mathbf{f}_{iW} \tag{3.1}$$

where m_i is the mass of the pedestrian i and $\mathbf{v}_i$ its velocity. $\mathbf{v}_i^0$ is its desired velocity; in consequence, the first term on the RHS models exponential approach to that desired velocity, with a time constant τ_i. The second term on the RHS models pedestrian interaction, and the third models interaction of the pedestrian with the environment.

The specific mathematical form of the interaction term does not seem to be critical for our applications as long as it decays fast enough. Fast decay is important in order to cut off the interaction at a relatively short distance. This is important for efficient computing, but it is also plausible with respect to the real world: Other pedestrians at, say, a distance of several hundred meters will not affect a pedestrian, even if those other pedestrians are at a very high density. We use an exponential force decay of

$$\mathbf{f}_{ij} = \exp\left(\frac{|\mathbf{r}_i - \mathbf{r}_j|}{B_p}\right) \frac{\mathbf{r}_i - \mathbf{r}_j}{|\mathbf{r}_i - \mathbf{r}_j|} ,$$

which seems to work well in practice. $\mathbf{f}_{ij}$ is the force contribution of agent j to agent i; $\mathbf{r}_i$ is the position of agent i. Alternative more sophisticated formations are described by Helbing et al. (2000).

For the environmental forces, $\mathbf{f}_{iW}$, often only interaction with fixed objects, such as trees or houses, is considered. For those, one can use the same mathematical form as for the pedestrian-pedestrian interaction.

3.1.3 Discretization

For a computer implementation, the differential equation needs to be discretized in time. A standard discretization is to first go through all pedestrians and calculate

$$\mathbf{v}_{i,t+h} = \mathbf{v}_{i,t} + h\left(\frac{\mathbf{v}_i^0 - \mathbf{v}_{i,t}}{\tau_i} + \frac{1}{m_i}\sum_{j \neq i} \mathbf{f}_{ij} + \frac{1}{m_i}\sum_W \mathbf{f}_{iW}\right) , \tag{3.2}$$

and then again go through all pedestrians and move them according to

$$\mathbf{r}_{i,t+h} = \mathbf{r}_{i,t} + \mathbf{v}_{i,t+h} .$$

An issue with this approach is that the velocity, as a result of Eq. (3.2), can be much larger than what is physically possible. Computationally, this can be prevented by limiting velocities to some maximum.

In our simulation, the velocity that comes out of Equation 3.2 is treated as the velocity the pedestrian *would like to move*. If this velocity is smaller or equal the maximal velocity the agent is able to walk, it is applied directly. Otherwise, the velocity is scaled down to the maximal velocity. This also means that even under extreme pressure, the pedestrians do not move faster than they are able to walk.

3.1.4 Desired velocity

An open issue with the above formulation is how to select the desired velocity. Many early example implementations seem only to consider pedestrians with a constant desired velocity, i.e. where in the absence of other pedestrians and obstacles the pedestrians would follow a straight line. Clearly, this does not work for any kind of more realistic scenario.

The CA methods, when applied to evacuation scenarios, solve this problem by reducing Eq. (3.2) to a few boolean rules: Pedestrians cannot move into walls; they cannot move into other pedestrians; and apart from that they follow some kind of local potential to the exit. It is difficult to translate this into the continuous formulation; it essentially means that desired speed is computed locally and never leads into obstacles, and it also means that pedestrians can only obstruct other pedestrians' movements but that they cannot push them from behind, as they can in Eq. (3.2).

More recent implementation of the continuous models (e.g. Hoogendoorn and Bovy, 2002) go the same way as the CA implementations, in the sense that there is a local field that gives the local desired velocity; i.e. we now have $\mathbf{v}_i^0(\mathbf{x}_i)$. Early implementations of this concept derive this field from an optimization argument and solve a discretized partial differential equation to obtain it in practice, which takes considerable computing time. In contrast, the CA methods recognized that an approximation to these potentials could be computed by simple flooding algorithms (Schadschneider, 2001), and that the generalization of those flooding algorithms to more complicated geometries is just the Dijkstra algorithm (Nishinari et al., 2001; Stucki, 2003).

However, none of these approaches solves the distinction between desired velocity and environmental forces in a satisfying way. The discussion can maybe be summarized as follows:

- There seems to be some overlap between environmental forces and desired velocity. This problem can be solved when environmental forces remain restricted to those forces that are independent from the destination of the pedestrian.

- The consequence of this is that for all but the simplest geometries, the desired velocity is a local property. This is in contrast to early implementations, where for example the movement around a column was (as far as we can tell) computed from a spatially and temporally constant desired velocity, that is, the movement around the column was achieved by the wall forces only.

- An open conceptual problem remains the balancing of the wall forces and the desired velocity. In particular, a larger τ_i implies a smaller influence of the desired velocity, which seems implausible. Also, if the floor field is obtained by the optimization argument mentioned above, then a single pedestrian following that field should never feel the wall forces because it is already included in the computation. The wall forces would come into play only when other pedestrians push the pedestrian away from the optimal path. This, however, implies that the wall forces have a very limited interaction range, and that a single pedestrian always stays outside this range. It may be possible to implement this computationally, but it is clearly inconsistent with current model formulations.

3.2 Our Approach to the Desired Velocity Problem

For our simulation, we need to assume that the geometry is given by a graph (network) of hiking paths plus some terrain features, and that the desired velocity of the hiker is consistent with this graph.

3.2.1 Naive approach

The maybe first implementation that comes to mind is as follows (Fig. 3.2):

- Make the desired velocity point directly to the next waypoint, R:

$$\mathbf{v}_i^0 = v_i^0 \, \frac{\mathbf{R} - \mathbf{r}_i}{|\mathbf{R} - \mathbf{r}_i|} \, ,$$

  where $\mathbf{r}_i$ is the hiker's current position, and v_i^0 is the magnitude of the desired velocity. Once a waypoint is reached, $\mathbf{R}$ is moved to the next waypoint.

- Set the environmental force field zero on the path, and such that it pushes the hiker back onto the path outside.

However, this approach has the disadvantage that, close to a waypoint, agents are artificially pulled toward that waypoint even if that does not make sense (see Fig.3.1).

This becomes particularly awkward with pedestrians moving in different directions, since they will all crowd around that waypoint although the street or path is wide enough. This could be avoided by switching to the next waypoint before actually reaching this waypoint, but then without special measures pedestrians may not be able to execute a switchback because the next waypoint may pull them back on the current segment.

Also, if two paths are close together and a pedestrian is pushed away, say by other pedestrians, too far from its own path, the environmental forces which pull the pedestrian to the nearest path may pull the pedestrian to the wrong path. This will eventually be corrected when the simulation continues since the pedestrian is still pulled toward the correct waypoint, but it looks implausible. The conceptual reason behind this is that in the naive implementation the path forces depend on the location rather than on the agent's own intentions.

Finally, encoding the width of the path by forces in the cells forces us to use relatively small cell sizes, say $(0.25\ m)^2$, to reduce artifacts caused by the cellular structure for off-axis movement (see Fig. 3.2). The concept of cells to represent the environment is introduced in Section 3.4.

3.2.2 A More Sophisticated Approach

In this section, a model will be presented that allows agents to follow a path without those two artifacts. The idea is to move to a force system that follows the path. The path is given by a line, which in turn is given by our input data. This path line may for example be a piecewise linear object, or a Bezier

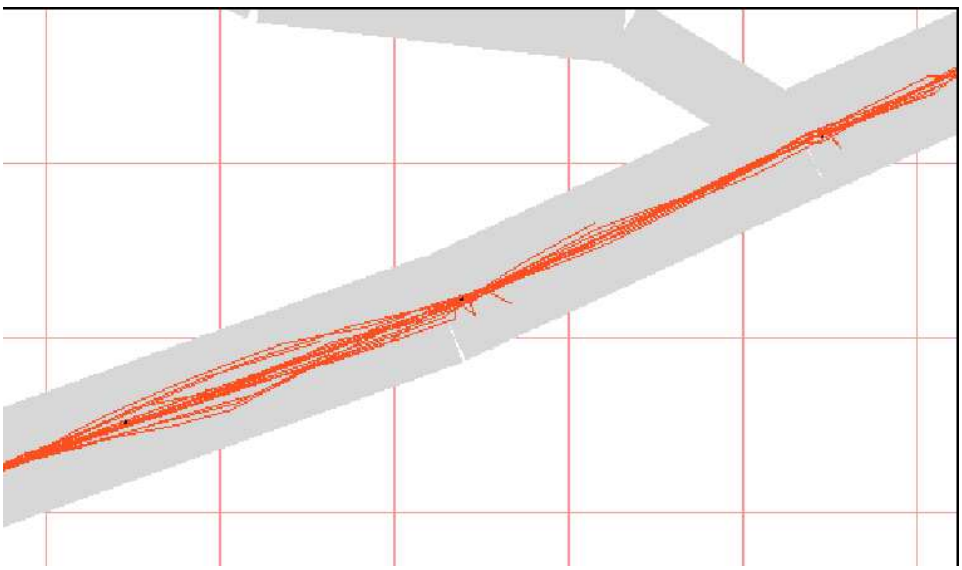


Figure 3.1: Traces of hikers in the first model, where they are all pulled toward the same waypoint. Note how the trajectories focus near the waypoint, and diverge before and after. The width of the path remains unchanged.

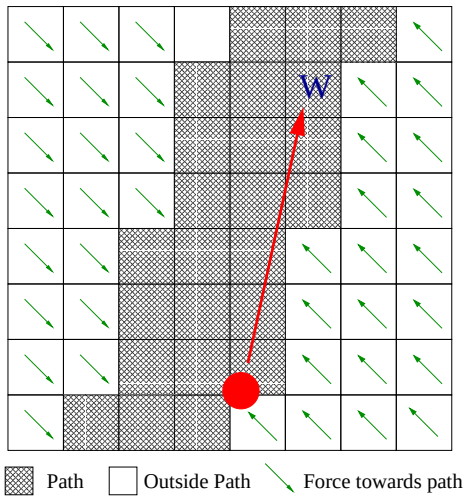


Figure 3.2: Example of the force toward the waypoint. It is at the same time an illustration of the use of cells for local force information.

curve, or a spline.[1] Each pedestrian now has, besides its true location, a *shadow tag* on that line. The shadow tag is always computed such that the connection between the shadow tag and the pedestrian is orthogonal to the path line. The desired speed $\mathbf{v}_i^0$ is now computed such that it is in the direction of the

[1] In fact, standard B-splines do not work well; instead, one can use Akima splines (Akima, 1972).

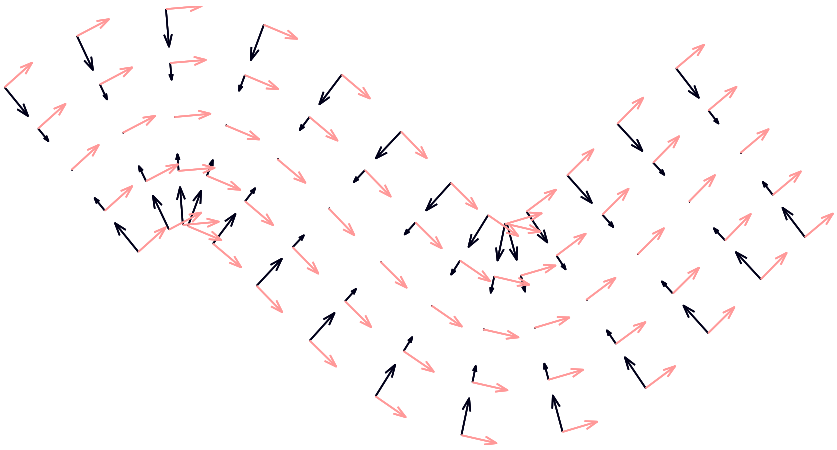


Figure 3.3: Path-oriented coordinate system for the computation of the desired velocity and the path forces. The light arrows show the desired velocity, which drives the agent forward along the path. The dark arrows show the path force, which pull the agent toward the middle of the path (from Mauron (2002)).

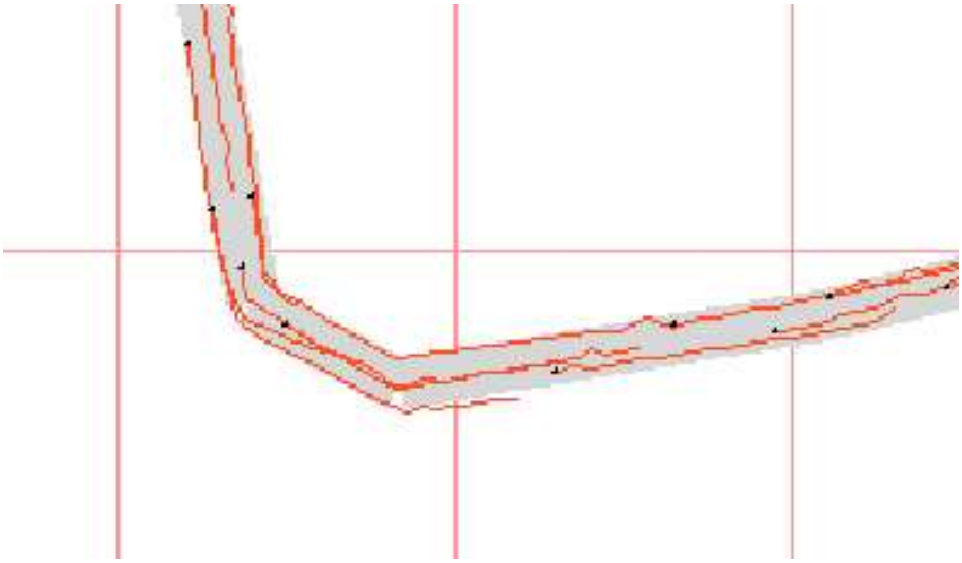


Figure 3.4: Traces of pedestrians walking in the same direction according to the second model. Note that they stay on their side of the path, even at a bend.

path line (light arrows in Fig. 3.3); and there will be a weak environmental force component toward the center of the path (dark arrows in Fig. 3.3). In consequence, agents now essentially move parallel to the path direction and do *not* have a tendency any more to go to the center of the path near waypoints. The success of this is documented in Fig. 3.4.

Note that the shadow tag is used only for computing the path force. Once the agent has moved, the position of the shadow tag needs to be computed again. In general, this is a non-linear problem and thus needs to be solved by an iterative algorithm, for example the one by Brent (1972). Even with this algorithm, it is clear that it can become stuck in a local minimum, and care needs to be taken to prevent that situation. The situation gets confounded by the fact that shadow tags need to be passed on across waypoints, and the best position for a shadow tag could be already on the next segment. Some details of this are discussed by Mauron (2002).

In our case, it turns out that the path data is of high enough detail that piecewise linear interpolation between waypoints is sufficient. For this case, the position of the shadow tag can be computed directly, and the only thing that can happen is that is has moved on to the next segment.

What we do is to calculate the distance d_{i+1} of the pedestrian also to the next segment P_{i+1} and compare it with d_i. If d_{i+1} is smaller or equal than d_i, then P_{i+1} becomes the current segment.

However, what we have currently implemented is a simples variation of the algorithm described above: at the end of each link segment is a virtual finishing line, which is perpendicular to the link segment. As soon as the agent crosses that line, the shadow tag is assigned to the next segment.

A resulting artifact of this is that at connections with angles smaller than 90 degrees, agents will move toward the inner side of the curve at an implausible location (see Figure 3.5). For our purposes, this artifact can be accepted, since such situations are rare with the given data.

There exist other solutions to this issue, including one that uses the bisecting line of an angle as criterion when to switch to the next segment. However, in order to calculate this line, both the current and the potential new segment are involved in the calculation. This is a more complicated operation, in particular when the route needs to be retreived in order to find out the relevant out of several links across an intersection. Whereas by using the perpendicular finishing line, which is based on the current segment only, the decision which segment will be the next segment can be determined at the moment the agent leaves the current segment.

Technically, this means that the environmental force contribution, $\sum_W \mathbf{f}_{iW}$, is now separated into two terms,

$$\sum_W \mathbf{f}_{iW} = \mathbf{f}_{obstacles} + \mathbf{f}_{path} \ . \tag{3.3}$$

Obstacles are still treated as non-moving pedestrians. The path force $\mathbf{f}_{path}$ (denoted by dark arrows in Fig. 3.3) keeps the pedestrian on the path and is perpendicular to the path line. It is given by

$$\mathbf{f}_{path,i} = A_{path,i} \left[\exp\left(\frac{h_1 - h}{B_{1,i}} \right) - \exp\left(\frac{h - h_2}{B_{2,i}} \right) \right] \mathbf{n}(s) \ , \tag{3.4}$$

where $\mathbf{n}(s)$ is the vector normal to the path, h_1 and h_2 characterize the path width, and the constants $B_{1,i}, B_{2,i}, A_{path,i}$ characterize resp. the range and strength of the pedestrian-path interaction, individually for each pedestrian i. Some plots of $|\mathbf{f}_{path,i}|$ can be found in Fig. 3.7. The force Eq. (3.4) can be derived from a potential as $\mathbf{f}_{path,i} = -\partial_h V_i$ with

$$V_i = A_{path,i} \left[B_{1,i} \exp\left(\frac{h_1 - h}{B_{1,i}} \right) + B_{2,i} \exp\left(\frac{h - h_2}{B_{2,i}} \right) \right] \ .$$

Since the potential is more intuitive than the force plot, it is given in Fig. 3.8.

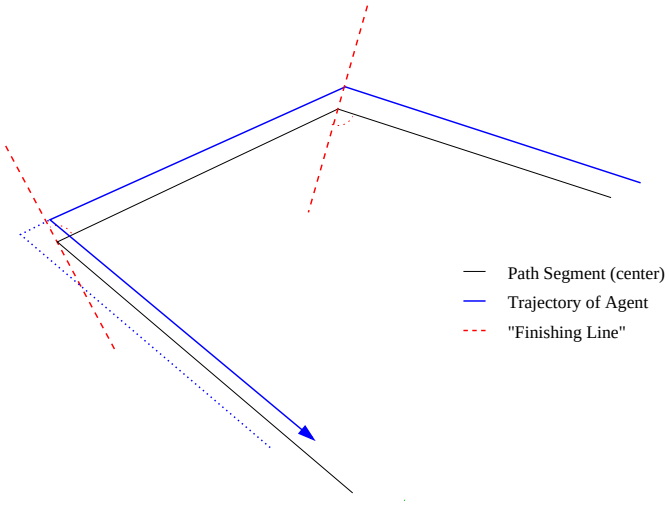


Figure 3.5: Red, dotted: perpendicular bisector of the side at the end of a link. If the agent (blue, solid) passes this virtual finishing line, it switches to the new link. If the angle of a connection is smaller than 90 degrees, it moves toward the inner side of the curve. More plausible would be the dotted path.

Note that an equilibrium position can be found by setting the force equal to zero; this results in

$$h_{0,i} = \frac{B_{2,i}\, h_1 + B_{1,i}\, h_2}{B_1 + B_2} \,. \tag{3.5}$$

Sec. 3.3 will describe an asymmetric walkway where all these parameters are indeed used. In general, however, we will use $h_1 = -h_2$, $B_{1,i} = B_{2,i}$, and a uniform A_{path}.

3.2.3 Implementation details

In the implementation, a time-step of $h = 0.5s$ is used. τ is set uniformly to 1. Pedestrians are considered one at a time, and *both* the velocity update and the movement step are performed. No large differences to a parallel update were observed. As mentioned in Sec. 3.1.3, it may happen that the velocity according to Eq. (3.2) is much larger than is plausible. For that reason, if the magnitude of the velocity is larger than v_i^0, it is artificially reduced to v_i^0.

3.3 Model Calibration

In this section field measurements on pedestrian flow are presented. These were performed by Mauron (2002) using a video camera. He took video footage of pedestrians walking on a sidewalk and measured their distance to the borders. We used these observations to select an appropriate path force model and calibrate various model parameters.

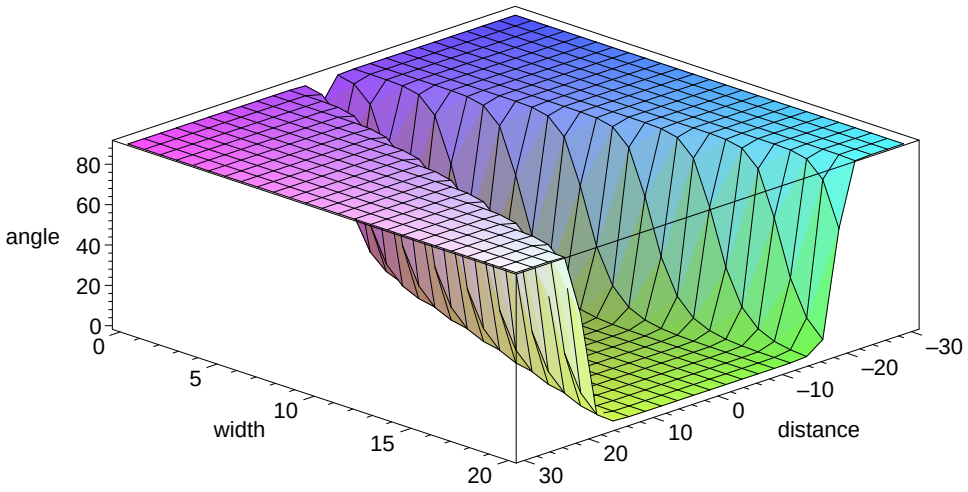


Figure 3.6: Angle between the "force" caused by the desired velocity, and the path force. This is both a function of the path width, and the distance from the center of the path. At large distances from the path, the agent is driven toward the path with no component in its desired walking direction. On wide paths, the lateral component of the force is nearly zero. For the plot, the velocity of the agent, $\mathbf{v}_i$, is assumed to be zero.

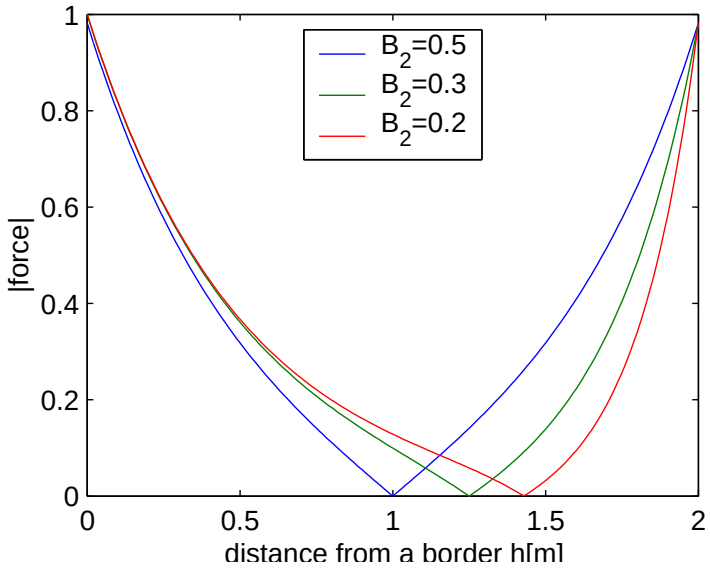


Figure 3.7: Force $|\mathbf{f}_{path}(h)|$ in a 2 meters wide sidewalk ($B_1 = 0.5$). Note that the plot shows the absolute value $|.|$ of the force; to the right of where the force becomes zero, the force is actually negative (from Mauron (2002)).

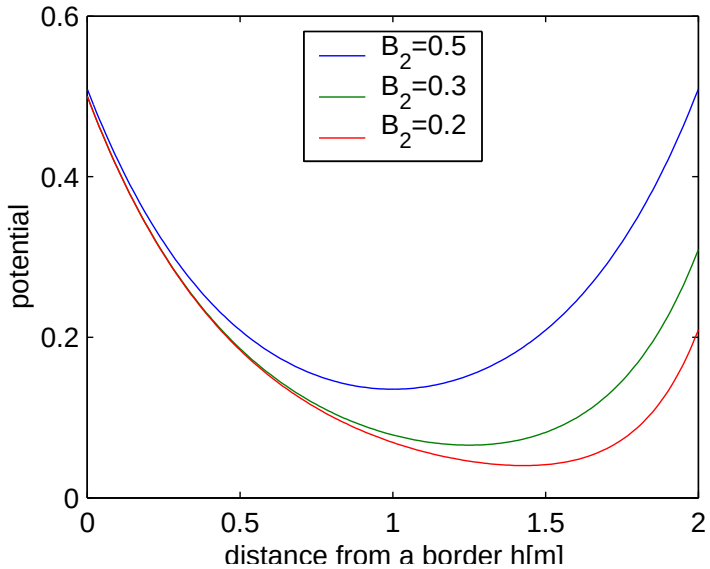


Figure 3.8: Potential in a 2 meters wide sidewalk ($B_1 = 0.5$), from Mauron (2002).

It was argued in (Hoogendoorn et al., 2001) that the Helbing model oversimplifies the dynamics of pedestrian flow. Although the Helbing model and his variations reproduce many observed phenomena including lane formation and oscillatory flows trough bottlenecks, and although simulation visualizations are qualitatively satisfying, it is not clear if the model describes accurately, *on the average*, the trajectories. We were interested in extracting macroscopic flow properties and compare them with simulations measurements of the same properties in order to test the relevance of the Helbing model.

The calibration of the model parameters like the force constants A and B in Eq. (3.4) can be difficult. A common method in microscopic modeling is to use observed macroscopic properties of the flow, like speed-density curves or flow-density and adapt the simulation parameters to match them (Hoogendoorn et al., 2001). A major difficulty in the case of pedestrian flow is that the speed-density curves, the so-called *fundamental diagrams*, are scarce. The usual reference curve by Weidmann (1993), although generally acknowledged as realistic, was not tested in the field when we started to work on this issue. An additional problem was the lack of specificity with speed-density curve calibration, since it was only possible to calibrate the whole set of parameters and not single parameters. A more precise fine grained calibration would be obtained by measuring specific properties of the pedestrian flow.

3.3.1 Experimental setup

Pedestrian movements were observed at a sidewalk on Tannenstrasse, between the CLA building and the main building of ETH in Zurich (Fig. 3.10). Advantages of that location included: it was possible to place the camera high above the observed area; different observation times resulted in different flow characteristics; the area is devoid of inhomogeneities such as additional entrances/exits, shopping windows, etc. A disadvantage is the slight uphill grade of the sidewalk.

After the video footage was taken, pedestrian movements had to be translated into a Cartesian field coordinate system. This was achieved by a half-automatic image analysis and coordinate conversion software written explicitly for this purpose (Mauron, 2002). The system was calibrated by four control points that were marked with bright duct tape on the pedestrian side walk. The field coordinates of those points were determined using standard measuring tape. A 1-meter reference stick, randomly placed on the sidewalk and analyzed via the system resulted in a length error of about $5cm$, which is reasonable for the purpose here. The width of the sidewalk was $2.5\ m$, resulting in

$$h_1 = 0.0\,m \text{ and} \tag{3.6}$$

$$h_2 = 2.5\,m\,. \tag{3.7}$$

After this, movements of real pedestrians were tracked. For this, second-by-second video images were read into the software, and their estimated projection of their center of gravity to the sidewalk was manually determined. That position was then converted by the software into field coordinates. Two situations were distinguished:

- *Single pedestrian.* No other pedestrian was within a 10 meter radius.
- *Opposing pedestrians.* Two pedestrians of opposing direction pass each other, without any other pedestrian within a 10 meter radius.

As will be explained below, the first scenario was used for the calibration of the B_i, while the second scenario was used to calibrate pedestrian-pedestrian interaction. 475 non-interacting pedestrians and 150 crossing events were used for the analysis. The resulting distributions are shown in Fig. 3.11.

3.3.2 Calibration of the path force

The observation of *non-interacting pedestrians* on a straight sidewalk have shown that they tend to walk in a straight line keeping a constant distance h from the road border. The radial pedestrian distribution obtained with the field measurements reveal that the h's are statistically distributed around a mean value $\overline{h}_0 \approx 1.6\ m$. The pedestrians tend to keep a larger distance from the street than from the wall which is understandable since the street is potentially more dangerous.

For a calibration of the path force constants, it was assumed that the path force constants $B_{1,i}$ and $B_{2,i}$ are normally distributed with a mean value $\overline{B}_1$, $\overline{B}_2$ and the same standard deviation ΔB. This results, via Eq. (3.5), in a distribution for the equilibrium values $h_{0,i}$. The constants $\overline{B}_1$, $\overline{B}_2$, and ΔB were now varied so that the resulting distribution of the $h_{0,i}$ matches as well as possible the field measurement distribution; note that the value of A does not matter here. The method converged to the following values,

$$\overline{B}_1 = 3.3\,m, \tag{3.8}$$

$$\overline{B}_2 = 1.1\,m, \tag{3.9}$$

$$\Delta B = 0.8\,m. \tag{3.10}$$

The corresponding pedestrian distribution is shown in figure 3.12. Although the simulated distribution peaks roughly at the same value h_o, the field distribution is significantly higher at h_o and decreases faster to the right.

3.3.3 Calibration of pedestrian interaction

Given the above calibration of the path force, the field measurements of the two-pedestrian-encounters can be used to calibrate the pedestrian-pedestrian interaction. The basic assumption here is that this interaction follows the same functional form as the path force, i.e.

$$\mathbf{f}_{pp}(\mathbf{x}_{ab}) = A_{pp} \exp\left(\frac{|\mathbf{x}_{ab}|}{B_{pp}}\right) \frac{\mathbf{x}_{ab}}{|\mathbf{x}_{ab}|} ,$$

where $\mathbf{x}_{ab}$ is the vector from the position of b to the position of a. The strength of the interaction force, A_{pp}, is assumed to be the same as the strength of the path force, A_{path}. Finally, it is assumed that the path force, with $B_{1,i} = \overline{B}_1$ and $B_{2,i} = \overline{B}_2$, should be exactly in equilibrium with the interaction force when the pedestrians are side-by-side.

The measured average positions when the pedestrians are side-by-side are

$$\overline{h}_a = 0.85 \pm 0.25\, m, \tag{3.11}$$
$$\overline{h}_b = 1.88 \pm 0.20\, m; \tag{3.12}$$

in addition, $\overline{h}_{ba} = \overline{h}_b - \overline{h}_a = 1.03\ m$ is the distance between the two average values.

For the forces to cancel out, this results in two equations, one for each pedestrian:

$$\mathbf{f}_{path}(\overline{h}_a) + \mathbf{f}_{pp}(\overline{h}_{ab}) = \mathbf{0} \tag{3.13}$$
$$\mathbf{f}_{path}(\overline{h}_b) + \mathbf{f}_{pp}(\overline{h}_{ba}) = \mathbf{0} \tag{3.14}$$

(where $h_{ab} = -h_{ba}$). Inserting the forces according to Eq. (3.4) into the first equation one obtains

$$\exp\left(\frac{h_1 - \overline{h}_a}{\overline{B}_1}\right) - \exp\left(\frac{\overline{h}_a - h_2}{\overline{B}_2}\right) - \exp\left(\frac{\overline{h}_a - \overline{h}_b}{B_{pp}}\right) = 0 , \tag{3.15}$$

where the assumption was made that all force strengths A are the same. Solving this equation with respect to B_{pp} results in

$$B_{pp} = \frac{\overline{h}_a - \overline{h}_b}{\log\left(\exp\left(\frac{h_1 - \overline{h}_a}{\overline{B}_1}\right) - \exp\left(\frac{\overline{h}_a - h_2}{\overline{B}_2}\right)\right)} \approx 1.71\ m . \tag{3.16}$$

Given our previous values for $\overline{B}_1$ and $\overline{B}_2$, this value seems plausible.

Doing the same procedure for the second equation results in a much lower estimate for B_{pp} of $0.18\, m$. This is due to the fact that for a pedestrian close to the wall the presence of a second pedestrian does not shift the average value for h a lot – from $\overline{h}_0 \approx 1.6\, m$ to $\overline{h}_b \approx 1.88\, m$ – and therefore the other pedestrian does not have to "push" a lot to achieve this. The problem lies with the fact that we assume the same force strength A for everything; unfortunately, the available data is not enough to evaluate both the B_i and the A_i separately.

We conclude that for our purposes an interaction range of $B = 1\, m$ is plausible. This is true both for the path force and for the interaction force.

3.3.4 Fundamental diagrams, and the interaction strength A

Results for the following were obtained with a different implementation of the pedestrian simulation. There may be differences to the algorithm used in the hiking simulation especially at high densities. Since high densities are not the focus of the hiking simulation, it is expected that the general order of magnitude of the parameters is still useful also for the current implementation of the hiking simulation.

We measured the behavior of simulated pedestrians on a path with width $2\,m$, i.e. $h_1 = 0\,m$ and $h_2 = 2\,m$. The length of the path segment was $10\,m$. For this section, a constant and large path force strength of $A_{path} = 6000$ was selected, while the pedestrian interaction strength, A_{pp}, was varied as indicated. The large path force models solid walls. We were interested in the change of mean walking speed $\overline{v}$ of the agents if we change their density ρ.

A curious result when doing this with periodic boundary conditions is that the velocity does not go down at all, even with very high density. The reason is that the pedestrians move as a single block even at very high densities; this is caused by a combination of two properties of the equations: The pedestrian interaction is uniform in all directions, that is, a pedestrian from behind pushes as much as a pedestrian in front. Second, nothing in the formulation says that their velocity should depend on some distance from each other.

This is clearly unrealistic. It however turns out that with "randomized" boundary conditions, the problem goes away and reasonable fundamental diagrams result (Fig. 3.13). With randomized boundary conditions, a pedestrian reaching the end of the segment will reappear at the start of the other end with a randomized lateral position. This can cause unrealistic overlapping pedestrians, since another pedestrian may already be at that same position. In order to avoid that, the simulation checks if there is space available at the randomized location (for the precise algorithm see Mauron, 2002). If the location is already occupied, the pedestrian is stuck and the procedure is repeated at every simulation time step till there is space at this location. This essentially means that with randomized boundary conditions the "pushing from the back" is not done at the entry to the path segment.

From Fig. 3.13, one observes that an interaction strength of $A_{pp} = 100$ is too weak. Values of $A_{pp} \approx 300$ are plausible.

Finally, Fig. 3.14 shows fundamental diagrams when a share of pedestrians, C, is moving in the opposing direction. Clearly, with C increasing toward 0.5, the average velocity goes down. As expected, the effect is more pronounced for high than for low densities. We are not aware of any systematic field measurements to compare this data against.

3.4 Computational aspects

It was argued earlier that a cellular automata representation of space did not seem appropriate for our purposes. Instead, we use a continuous representation of space. However, some aspects of our simulation, such as walkability or obstacle forces, depend on the spatial location of the agent. These forces are relatively expensive to calculate, since one needs to enumerate through all possible objects that could influence a given location.

Fortunately, since these forces do not depend on time, they can be pre-computed before the simulation

Figure 3.9: Observed sidewalk at the Tannenstrasse (at ETH Zurich, between Hauptgebäude and CLA building, from Mauron (2002))

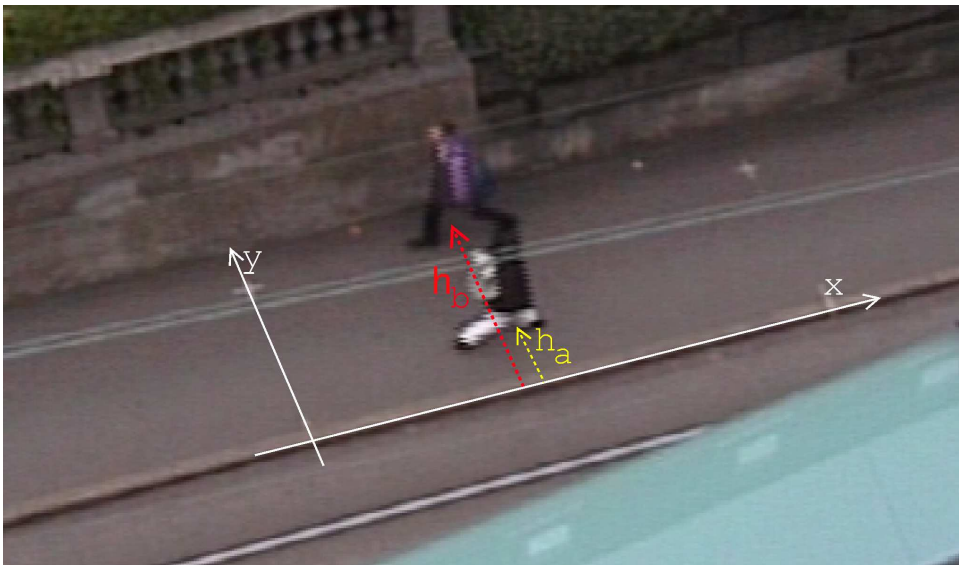


Figure 3.10: Measured quantities (from Mauron (2002))

starts. In order for this to be successful, some coarse-graining of space is necessary. For this, we use cells of size $25cm \times 25cm$, and assume that all time-independent forces are constant inside a cell. The

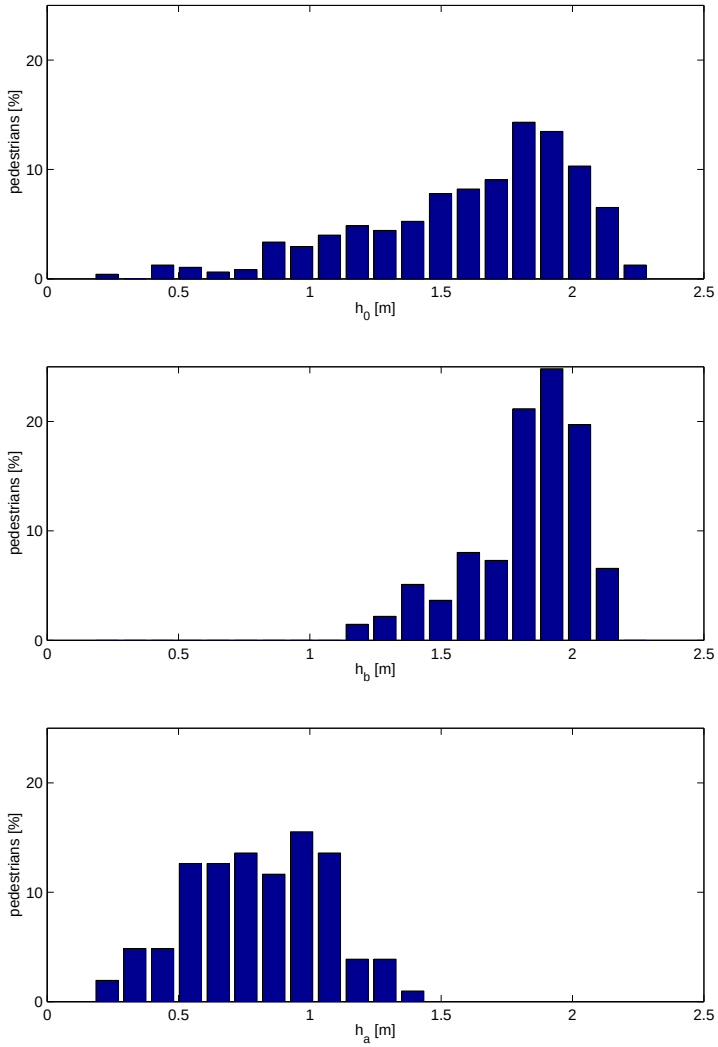


Figure 3.11: Real-world pedestrian distribution for non-interacting and interacting pedestrians (from Mauron (2002))

resulting force field (Fig. 3.2) becomes non-continuous in space, but this is not a problem in practice since this only influences the acceleration of pedestrians. That is, the acceleration contribution from the environmental forces can jump from one time step to the next, but since time is not continuous, this is not noticeable.

Yet, precomputing the values for all cells in a hiking region of, say, $50km \times 50km$, does not fit into regular computer memory. To avoid this problem, we implemented two methods: lazy initialization, and disk caching (Fig. 3.15). By *lazy initialization*, we mean that the values are computed only when an agent really needs them. In practice, the simulation area is divided into blocks of size $200m \times 200m$. Every time an agent enters one of these blocks, the values for *all* cells inside that block are computed. Since hiking paths cross only a small fraction of those blocks, the cell values for many blocks in our

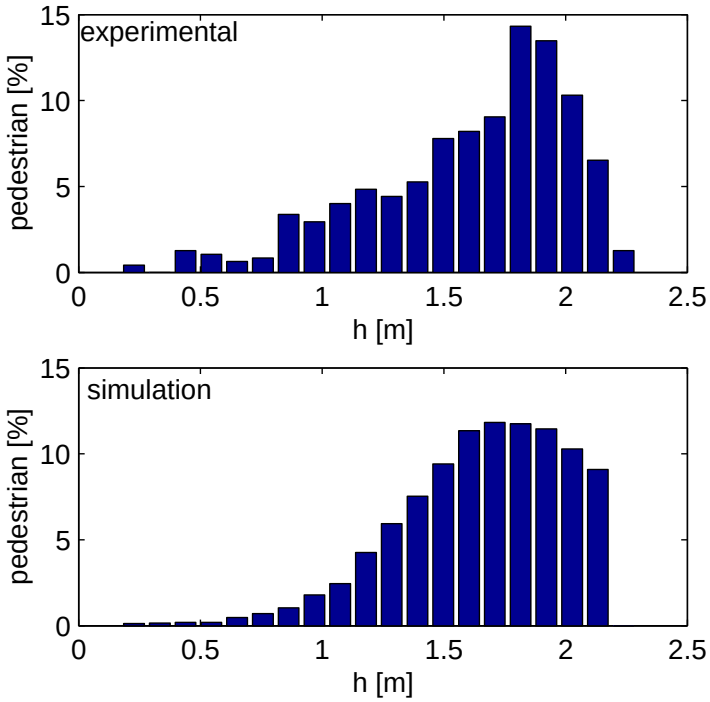


Figure 3.12: Validation: Real-world and simulated single pedestrian distribution (from Mauron (2002))

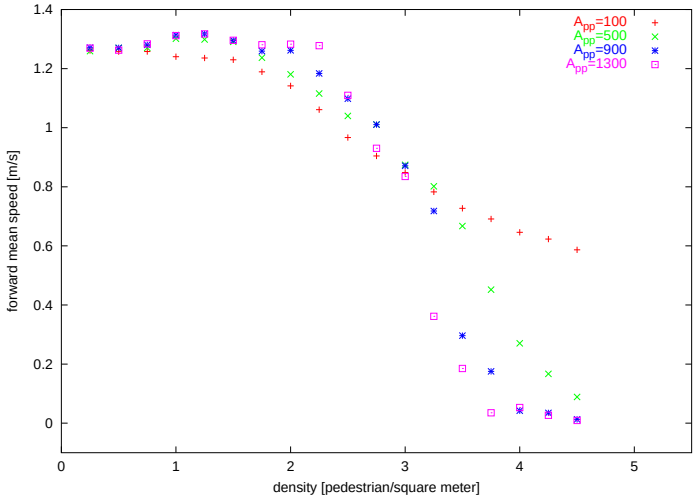


Figure 3.13: Average speed $\overline{v}$ vs. pedestrian density for uni-directional flow and randomized boundary conditions ($B_{pp} = 1$, $L = 10[m]$, $W = 2[m]$, from Mauron (2002)))

hiking area will never be calculated.

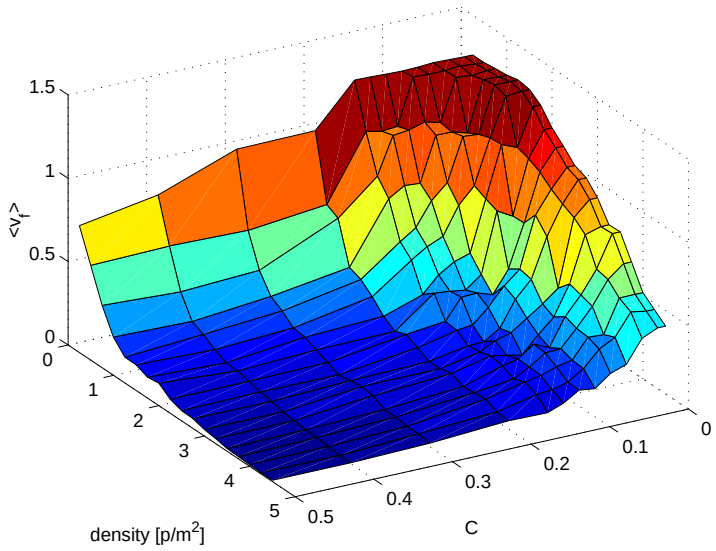


Figure 3.14: The maximal velocity of walking pedestrians decreases if their density rises, and when oncoming traffice increases. $A_{pp} = 300, B_{pp} = 1, A_{path} = 6000$*, random boundary conditions. (from Mauron (2002))*

In addition, the cell values, once computed, are stored on disk (*disk caching*). Every time when an agent encounters a block for which the cell values are not in memory, the simulation first checks if they are maybe on disk. Computation of the cell values is only started when those values are not found on disk. In consequence, a simulation started for the first time will run more slowly, because the disk cache is not yet filled.

If the simulation runs out of memory, then blocks which are no longer needed (i.e. which have not been crossed by an agent for a long time) are unloaded from memory. If they are needed again, they are just re-loaded from disk. This is the same mechanism like in operating systems, when access to a memory cell fails and a system call blocks. It would also be possible to allocate main memory for all the blocks and rely in the paging mechanism of the operating system. However, since we know more about the simulation, we are able do optimize the parameters for this special purpose.

For a testing scenario, of size $12\,km \times 15\,km$, we would need approx. 2.9×10^9 cells or 4500 blocks, resulting in 9 GByte memory requirement. The result of the lazy initialization together with the caching mechanism is that 50 MByte are enough for the scenario shown in Chapter 9.2.3 (Fig. 9.6). The computational speed for that simulation, with 300 hikers, was about 40 times faster than real time.

An additional advantage of the blocks, well known from molecular dynamics simulations, is that one can use them to cut off the short-range interaction between the pedestrians. Agents which are not in the same or one of the eight adjacent blocks are ignored (Fig. 3.16). This implies that there needs to be some data structure where agents are registered to the block. Agents that move from one block to another need to unregister in the first block and register in the second one. In this way, an agent searching for its neighbors only needs to go through the registered agents in the relevant blocks. This brings the computation complexity from O(N^2) down to O(NM), where N is the number of all agents

in the simulation, and M is the number of agents in a single block. M is a reasonably small number when compared to the number N of all agents in a real-world scenario.

However, it turned out that the number of agents used in our test scenarios (up to 2500) is still to small. We were not able to show the expected speed-up. The simulation starts to be really slow as soon as there are more than 2500 agents in the system, since it runs on a single CPU.

In order to reduce the time needed for computing the social force, an alternative idea would be to sort the pedestrians with respect to the *distance along the path*, s. The Δs between two pedestrians with path coordinates (s_1, h_1) and (s_2, h_2) is given by

$$\Delta s = |s_2 - s_1| \,. \tag{3.17}$$

Calculating Δs requires only the subtraction of two scalars, which is a less expensive operation than calculating an Euclidean distance. In addition, one just needs to search along the graph. One can assume that when Δs is above than a certain value s_c, two pedestrians will not interact. This assumption is based on the fact that humans do not always evaluate distance the "Cartesian way", for example if two pedestrians are on a U-shaped path. In that case, the Euclidean distance may be small but agents still do not interact because the distance along the path Δs is important. This technique is, however, not implemented in the current pedestrian simulation.

An alternative approach to make the mobility simulation faster is to use a parallel computer. An example of a parallel implementation of the social force model is by Quinn et al. (2003). However, that implementation introduces a parallel version of a plain social force model, which lacks the additional features that we have introduced here, such as path following. Also it divides the simulated are into rectangular blocks. However, since our approach is graph-based, it would be better to use a graph-based partitioning scheme as well (as used for the parallel traffic simulation by Cetin, 2005).

Despite the fact that the simulation can be implemented as a parallel program and runs on multiple nodes, it would appear as a single module to the rest of the system. For most of the other modules there is no difference between a parallel and a single CPU mobility simulation.

3.5 Movements inside Town or Buildings, or close to obstacles

The sections before introduced a model that uses only sparse information which fits into computer memory, runs efficiently on our scenarios, and has agents follow paths without major artifacts. The model uses a path-oriented coordinate system (see Fig. 3.3) for the computation of the desired velocity. This model also uses a so-called *path-force*, which pulls the agents back on the path when he moves away from its center (e.g. due to interaction with other agents or obstacles).

A question is what happens when there is no path. This can, for example, be the case in cities or in buildings. The following two sections will discuss implementations of two approaches to this problem when destinations are known. The first approach computes $\mathbf{v}_i^0$ as a function of the spatial location. The second approach first overlays a special graph over the system, and then lets the pedestrian move along the edges of the graph.

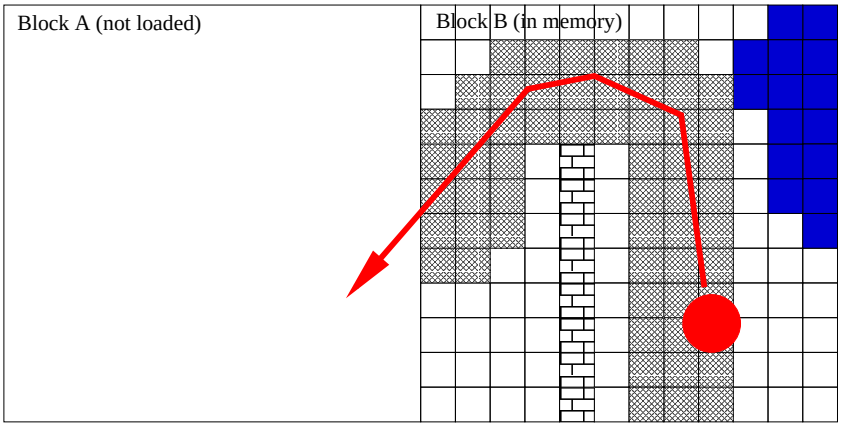


Figure 3.15: Since block consume a lot of memory, they are loaded into memory as soon as an agent walks over, and are deleted after a while.

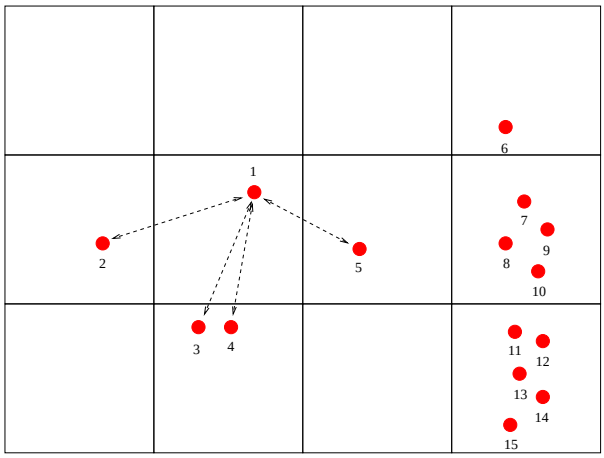


Figure 3.16: The scene is divided into multiple "blocks". Only forces from adjacent blocks have an influence to an agent.

3.5.1 Potential Field Model

The first approach looked at in this paper is based on a *utility maximization* model (Hoogendoorn et al., 2001). For this model, a potential field is generated for the simulated area, which allows the pedestrians to find their destinations by walking toward the minimal potential.

Hoogendoorn et al. (2001) solve partial differential equations (PDEs) on a grid in order to obtain the potential. The computational requirements for this computation are considerable. It turns out that an approximation to the potential can also be generated by a much simpler *flooding algorithm* which starts at the destination and goes backward. The main difference is that the PDE solution treats off-axis distances exactly, while the flooding algorithm uses an approximation. For example, for the knight's

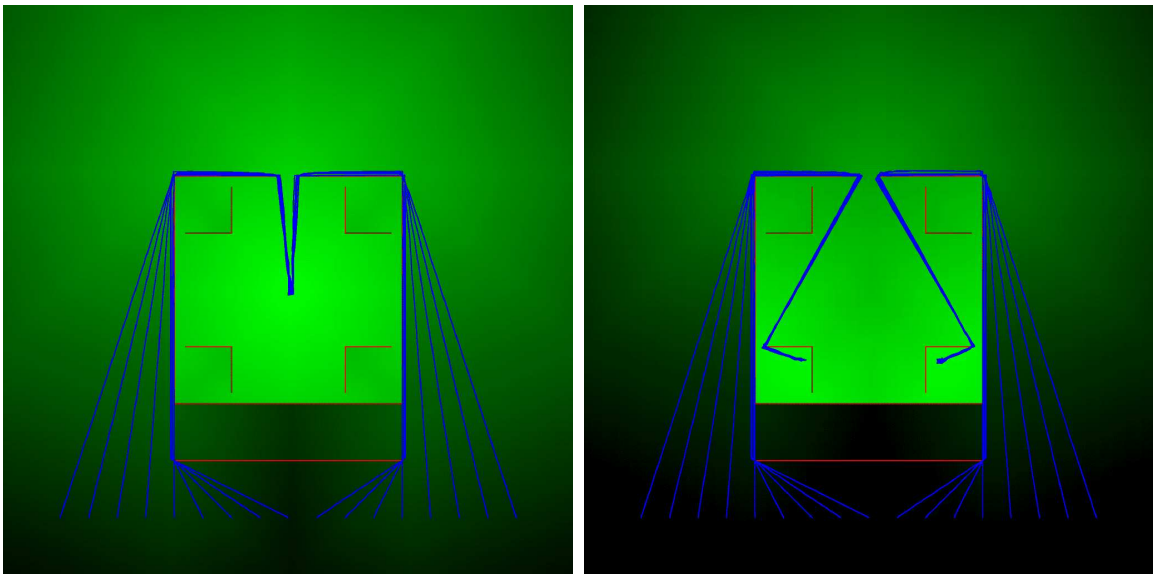


Figure 3.17: A potential field generated for one destination in the center of the formation (left) and for two destinations (right). A lighter color means a lower potential (close to the center, the potential is lowest). Some example walking directions derived from this potential field are shown in blue (from Stucki (2003)).

movement of chess, our algorithm would obtain a distance of $1 + \sqrt{2} \approx 2.41$, while the exact distance would be $\sqrt{2^2 + 1^2} = \sqrt{5} \approx 2.24$. As a result, a potential field as shown in Figure 3.17 emerges. Note that for each possible destination a separate potential field needs to be created. However, this field can be used for all agents heading to that destination, regardless of their starting position.

Before the potential of a new cell can be calculated, a visibility check has to be performed to ensure that the cell is accessible (visible) from the neighbor cell. As the check has, with a naive implementation, to iterate over all obstacles in the simulated area, the calculation time increases linearly with the number of obstacles (Fig. 3.18).

To deduce an agents' desired velocity, or direction, from this potential field is not trivial. The first idea might be to just walk in the direction of the neighbor cell with has the lowest potential, and which is therefore closest to the destination. Using this method, however, yields a zigzagging path, because the possible walking directions are limited by the number of neighbor cells. One possibility to increase the number of possible walking directions is to increase the number of cells looked at. For example, if the 16 cells which are two steps away are considered, 16 directions are possible.

A more realistic result can be achieved if the algorithm considers more than the nearest neighbor cells. One can follow the minimal potential until the cell can no longer be reached directly from the starting cell and let the agents walk directly into this direction (Figure 3.19).

It is possible to solve the problem of finding the optimal walking direction using the correct distances to the destination instead of using Manhattan distances (Nishinari et al., 2001). This basically yields in a graph connecting all corners of every object. The pedestrians appear to walk along the edges. They

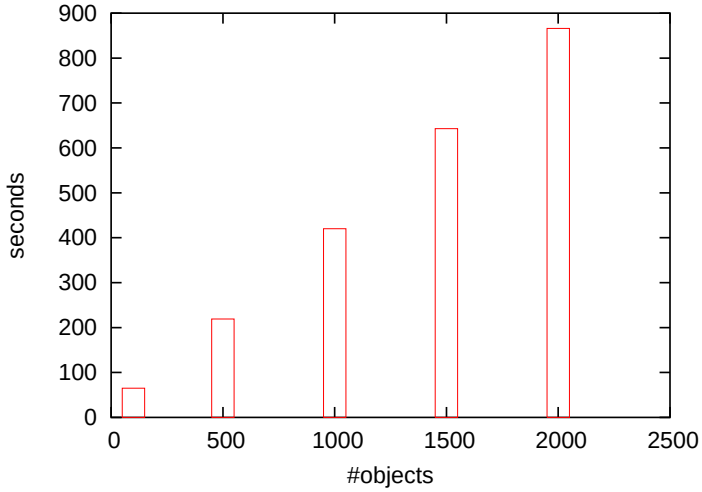


Figure 3.18: For the computation of the potential field (Section 3.5.1), different parameters influence the time needed: If we increase the number of obstacles in the scenario, the time to precompute the potential field rises. This test was calculated using 25x25cm cells in the Zurich Main Station Scenario.

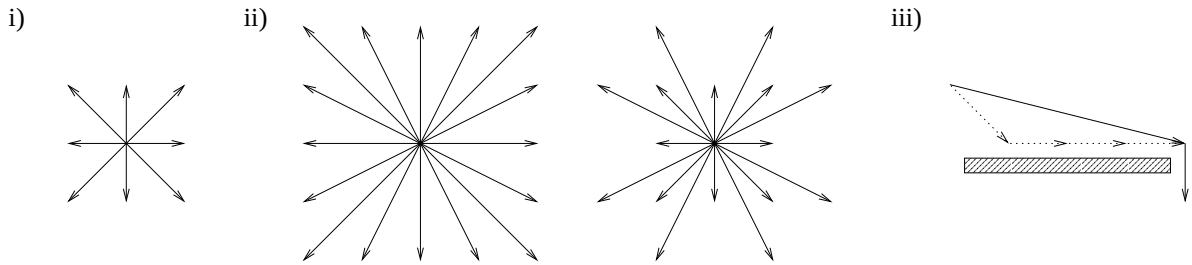


Figure 3.19: Possible walking directions $\mathbf{v}_i^0$ are limited by the number of neighbor cells looked at for a lower potential (i and ii). A non-local search can yield in a much shorter path, but is more expensive to compute (iii).

describe the *dijkstra-method*, which is about finding the real potential of each cell using a *visibility graph* and Dijkstra's algorithm((Dijkstra, 1959), while the pedestrian dynamics remain cell-based.

3.5.2 Graph Model

For the ALPSIM project, we use a network of hiking paths in the Alps. This network has a resolution of approximately 2m, which is accurate enough for paths through forests or meadows. However, inside villages or close to obstacles, a resolution of 25cm is needed to model a realistic behavior of the pedestrians.

One possibility would be to switch to the potential field model (Sec. 3.5.1) where needed. Since only

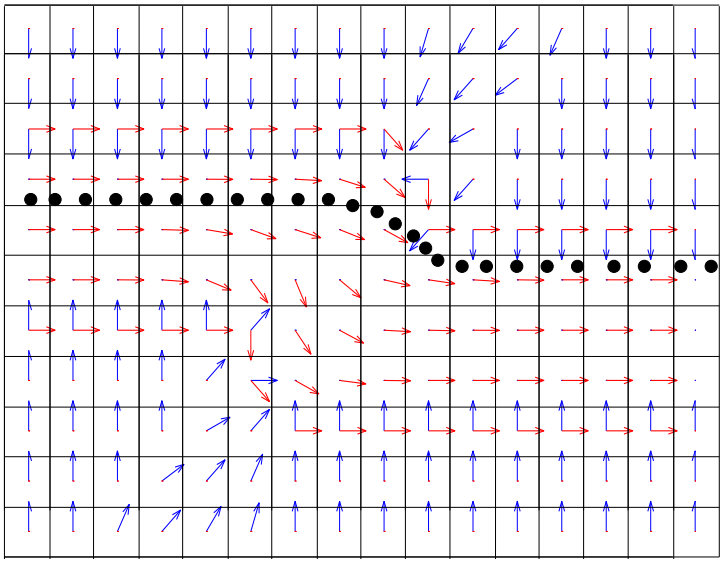


Figure 3.20: The hybrid simulation technique. The forces (arrows) are valid for the whole cell; a pedestrian's trajectory (dots) can follow arbitrary positions.

$\mathbf{v}_i^0$ is calculated by the potential field, all other aspects of the pedestrian dynamics, as forces between pedestrians or from obstacles, would remain the same.

However, a more intuitive solution is to keep the existing graph, but to add more details where needed. We were looking for a method that generates a graph around a given set of obstacles. This path can then be merged with the global hiking path graph.

In a first step, one has to decide where the *nodes* of the new graph should be. There should be enough nodes that an agent is able to circumvent obstacles on a naturally looking path. However, each additional node has to be considered in route choices and adds to path lengths. Paths, which are lists of nodes, are stored in the agents' brains or are transmitted over the network.

It is not that easy to find a simple yet realistic position of a node. The simplest solution would be to add a node to each corner of all objects (see Figure 3.21a). However, people tend to keep a certain distance to objects. Weidmann (1993) outlines that pedestrians keep an distance of 0.25 m (inside buildings) and 0.45 m (outdoors) to walls, even more to obstacles like fences. The paths would be too close to the objects. This would not be a problem, since agents as well keep a distance to obstacles due to environmental forces, but it is better to avoid the problem in the graph directly.

It was decided to use multiple nodes for each corner of an object. Each of them is in a distance of 0.25 m to the object corner (see Figure 3.21c). It does not matter in which direction the pedestrian approaches the corner, the distance to the corner is more or less equal. In order to reduce the numbers of nodes, an algorithm is run that eliminates nodes lying inside an object.

Based on these nodes an initial *graph* is constructed. At first, every node is connected to each other node, and each such edge has an weight of the distance between the nodes. We run a visibility check algorithm, known from computer graphics, to determine all the edges that intersect with obstacles.

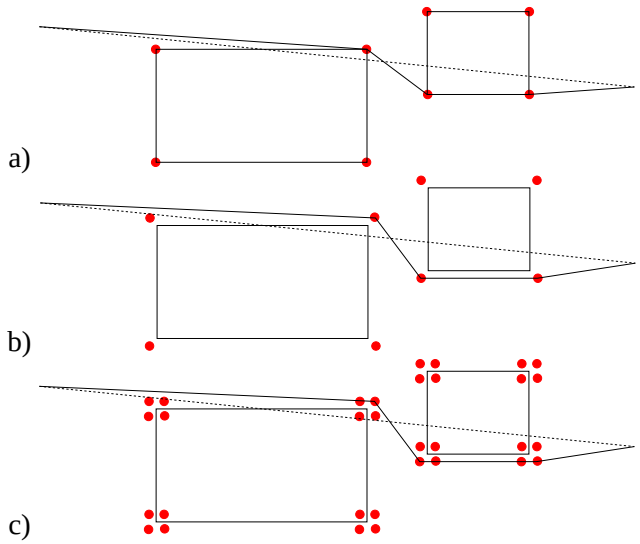


Figure 3.21: a) The simplest solution to place nodes would be to add a node to each corner of all objects. b) Since people tend to keep a certain distance to objects, the nodes should be placed at a distance of 25cm to the corner of obstacles. c) It is, however, easier to place 4 or more nodes close to each corner and remove the ones inside an object in a later step.

These edges are deleted.

In order to find the *shortest path* through this network, the shortest path algorithm by Dijkstra (1959) is used.

3.5.3 Simulation of Zürich Main Station

As a real-world scenario for testing our models, we chose a simulation of an evacuation of Zürich Main Station (Figure 3.22). We simulated an area of 700m $\times$ 200m with more than 3000 obstacles. Agents were placed randomly within the simulated area (neither inside buildings, nor on the tracks). We considered eight different exit locations, every pedestrian chooses the one closest to him. Note that we did just one iteration, which means that congestion at a certain exit can occur and is not avoided by the pedestrians. It would be possible, however, to run multiple iterations of the scenario to enable the agents lo learn from such a situation (e.g. Gloor et al., 2003; Raney and Nagel, 2004a).

To handle the different levels of the Main Station, we had to introduce the concept of stairs and elevators. Different levels were placed beside each other. Elevators were implemented like teleportation (with a time delay of some seconds), stairs are divided into two halves, each simulated in a level, with teleportation in the middle. However, forces between pedestrian on different sides of this boundary still contribute to pedestrian movements.

Since Zürich Main Station has more than one exit, we had reflect this in our models. For the potential field model, this is simple: starting the flooding algorithm from all exits simultaneously is sufficient. All the different exits are stored in the same precomputed map, since for every given point in the simulated

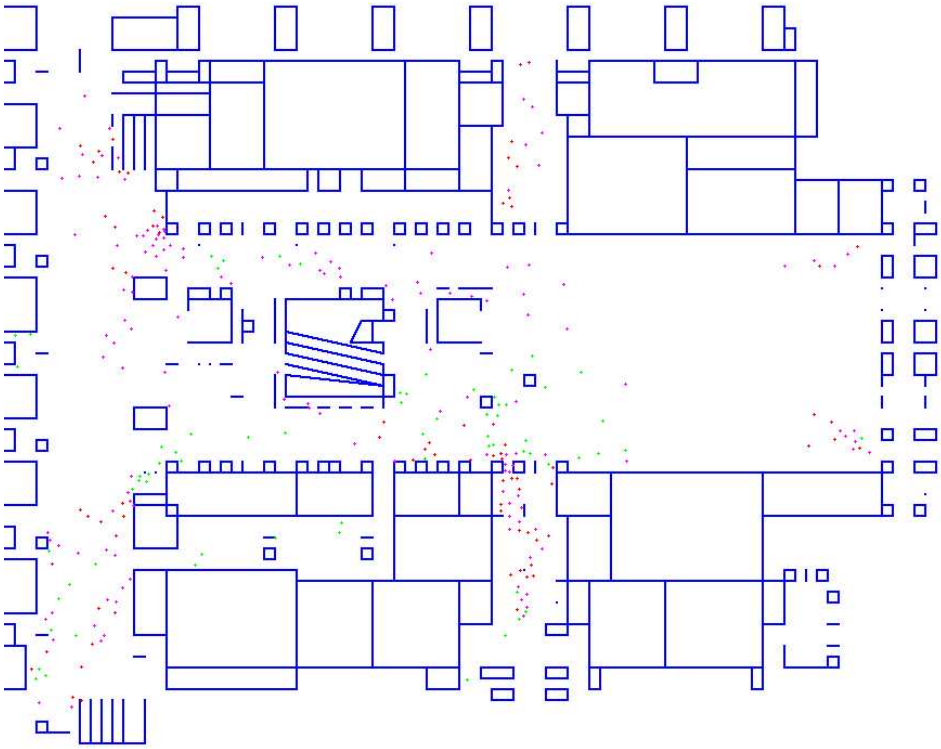


Figure 3.22: An evacuation of Zürich Main Station using the potential field model. The pedestrians hurry to the closest exit available. This is a capture of the first of multiple iterations, which means that congestion at a certain exit can occur and is not avoided by the pedestrians (from Stucki (2003)).

area, there is exactly one closest exit.

For a pedestrian simulation, two measurements are important: i) how realistic the results are, and ii) how fast the computation is.

A comparison of the presented models is shown in Table 3.1. The Zürich Main Station scenario was run for 100 and 500 agents using each model.

For the potential field model, the size of the cells that store the potential field affects the time that the pre-computation takes (Fig. 3.24). For our purpose, a cell size of 50cm$\times$50cm was chosen.

To pre-compute the graph in the graph model takes 175 minutes for the full scenario, containing all of the 3000 obstacles. If small obstacles like pillars or benches are removed for the graph generation, the time can be reduced to 35 minutes (Fig. 3.23). Small objects hardly influence the path of a pedestrian chooses. However, the pedestrians still do not walk through these obstacles, since the 3^{rd} term of the RHS of equation (3.1) still pushes them away from obstacles.

For the graph model (column c) and d)), the time to simulate the first steps takes longer than the average. This is because the agents are on an arbitrary position that is most likely not on the graph. They have

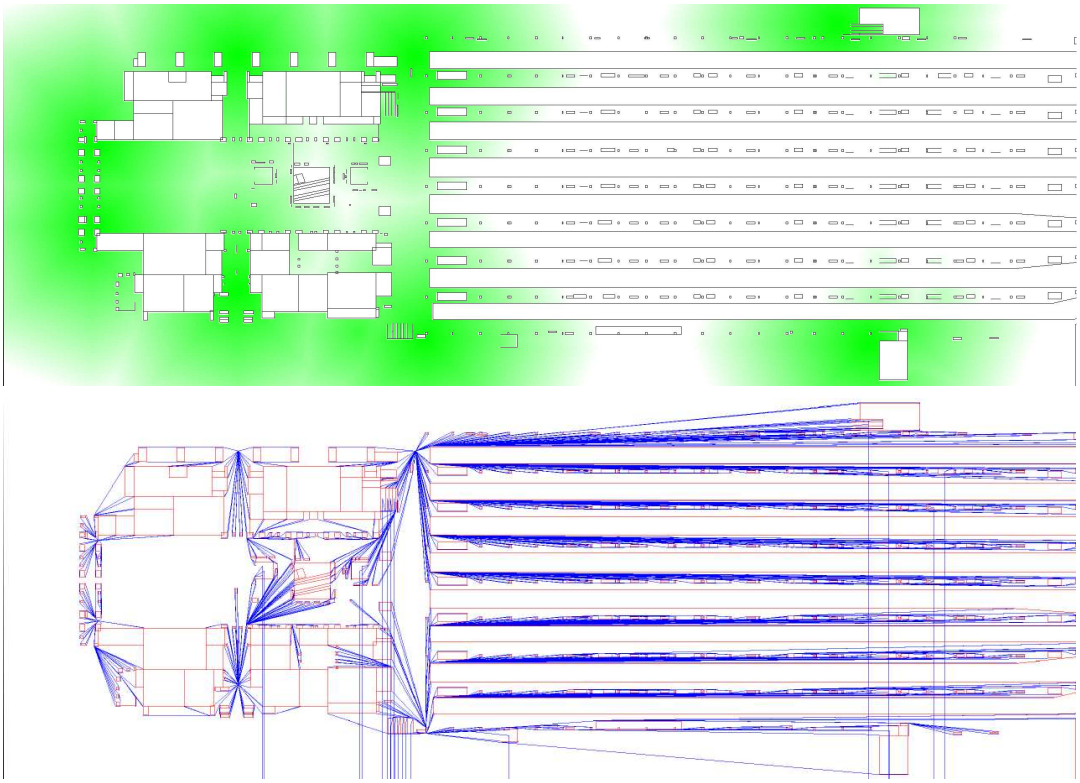


Figure 3.23: Potential field and spanning tree for Zürich Main Station, generated for the 8 potential exits. The potential field has to be generated each time the destination changes. This is also necessary for the spanning tree, but here, the underlying nodes do not change (from Stucki (2003)).

to find an node on the graph and walk into that direction first. Since in the potential field model, the field was calculated for every cell, the direction is available already. The potential field model needs considerable time to load the pre-calculated cells into memory (not shown in Table 3.1).

3.6 Pedsim, A Lightweight Version for the Public

There is need for a easy to use, but still powerful, pedestrian simulation for several reasons. There are semester and diploma theses, or projects from other departments or even other universities. Several persons asked us therefore if we could hand out our pedestrian mobility simulation. However, the mobility simulation as described in the sections above seemed to be to complex and it is bound to the data files, which we could not give away.

Therefore, PEDSIM was created. PEDSIM is a simple, microscopic pedestrian crowd simulation system. It consists, for the time being, of only a small simulation core and some functions to direct the pedestrians' desire. The output is a data stream in an XML-like events format. This stream can be used to attach external viewers in order to watch progress of the simulation in real time. PEDSIM uses

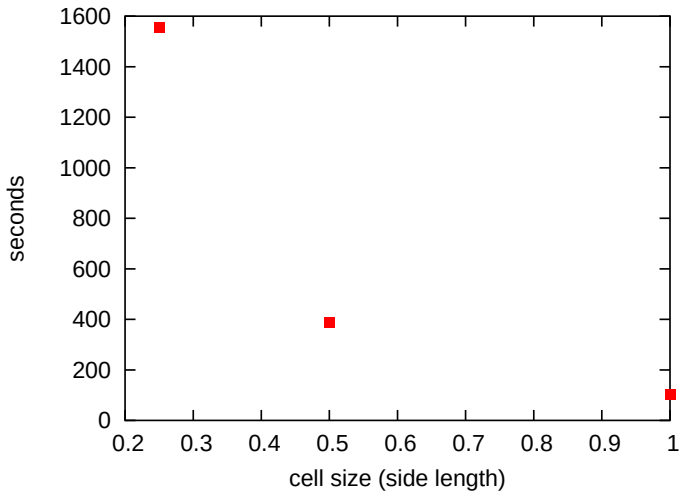


Figure 3.24: For the computation of the potential field (Section 3.5.1), different parameters influence the time needed: If we reduce the size of the ground cells, the time to precompute the potential field rises. This test was calculated using 3000 objects in the Zurich Main Station Scenario.

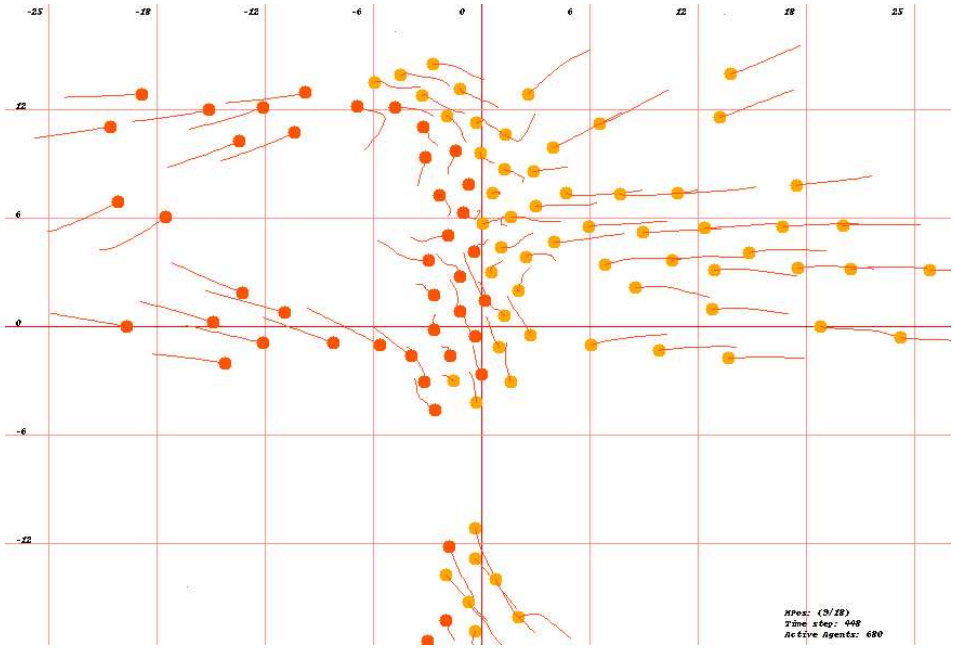


Figure 3.25: The 2-dimensional visualizer, connected to the crowd simulation "pedsim". The trails (red lines) of each pedestrian shows how it get diverted by other pedestrians.

the same physical model as the main mobility simulation, but the pedestrians are not following paths. Therefore, no street network is needed as input data.

	a)	b)	c)	d)
Pre-computation:				
Time to compute the force field or the graph	386s	386s	2134s	10582s
10 walking agents:				
Computing Time Total	81s	76s	38s	84s
Computing Time Until End of First Step	<1s	<1s	16s	66s
Computing Time Per Following Step	<1s	<1s	<1s	<1s
Walking Time	72s	116s	88.6s	80s
Max. Walking Time	132s	203s	199s	199s
Walked Distance	75m	98m	99m	96m
Maximal Walking Distance	132m	213m	237m	227m
100 walking agents:				
Computing Time Total	139s	75s	290s	525s
Computing Time Until End of First Step	<1s	<1s	131s	341s
Computing Time Per Following Steps	<1s	<1s	<1s	1s
Walking Time	67.1s	50.9s	70.6s	74s
Max. Walking Time	239s	180s	182s	182s
Walked Distance	73m	51m	80m	77m
Maximal Walking Distance	239m	217m	227m	229m
500 walking agents:				
Computing Time Total	282s	240s	1726s	4617s
Computing Time Until End of First Step	2s	1s	664s	1459s
Computing Time Per Following Steps	2s	1s	12s	12s
Walking Time	60.3s	67s	69.5s	76.6s
Max. Walking Time	234s	189s	262s	271s
Walked Distance	64m	79.4m	79m	78m
Maximal Walking Distance	236m	245m	247m	233m

Table 3.1: Time to simulate an evacuation of Zürich Main Station using a) potential field model with minimal neighbor approach (cell size: 0.5m), b) potential field with minimal distances approach, c) graph model ignoring small obstacles and d) graph model. Further the average/maximum time and distance needed to leave the station are shown. This shows that the compared models generate similar results.

PEDSIM does not use any of the advanced technologies the main simulation uses to improve the computation speed. However, this is not needed, since the purpose of PEDSIM is to be as simple as possible. Other persons interested in pedestrian and crowd simulations should be able to take this source code and start experimenting with it right away. PEDSIM was designed to be as portable and compatible as possible. It was initially developed for RedHat Linux (i686) using gcc 3.2.2, and was tested on various 32bit Linux systems, and on an Opteron 64bit machine with Fedora Core 1 as well. It should compile on every POSIX compatible operating system. However, for Microsoft Windows, the networking code

needs to be replaced (using e.g. the WinSock interface (see Windows Sockets (accessed 2005)), which basically provides the same features, but uses a slightly different syntax).

The world simulated by PEDSIM consists of a plane – infinite into all directions, and completely flat. However, it is possible to define obstacles which are avoided by the simulated agents. The basic obstacle is a point-like obstacle, which can be used for modeling e.g. trees. There is a radial force field originating at the center of the obstacle that pushes agents away. More complex obstacles can be build out of several point-like obstacles.

Every agent must have a goal, which defines the direction it walks into. There are two possibilities to define a goal:

- position based: The agent is given a position (co-ordinates), where he has to go to. It walks there on a direct line from its current position. Of course, if there are obstacles or other agents, it is deflected accordingly. If the agent eventually reaches that position, it will submit a `reached_position` event, which tells the modules of the mental layer to think of another goal. This is the mode that is used for the main mobility simulation as well, with the difference that there, agents are supposed to use hiking trails.
- Agent-based: The agent follows another agent. The agent tries to reach the other agent the same way it would walk to a position-based goal. However, it updates the position of the destination as soon as the other agent moves. Using this method, groups can be formed, where several agents follow one leader. The leader these uses a position-based goal.

At the moment, there is no configuration file. However, it is very straightforward to set the agents directly in the source code. The following examples defines 50 agents set up at random positions in a certain rectangle:

```
for (int i = 0; i<50; i++) {
   Tagent a;
   a.setPosition(random()/(RAND_MAX/40)-20,
                 random()/(RAND_MAX/100)-50);
   a.setType(0);
   agent.push_back(a);
}
```

PEDSIM uses exactly the same mechanism for the output as the normal mobility simulation does. It sends events through the events channel. This means that all the other modules can be used without modification. As an example, in Figure 6.12 the 2-dimensional visualizer is attached to PEDSIM. Two different types of agents (yellow and orange) seem to 'attack' each other. This effect is simulated by setting an agent-based goal, every yellow agent is assigned to a green agent, and vice versa. This assignment, however, is dynamic: at the beginning, every agent has a location-based goal assigned, which says basically that he should walk to the opposite side of the simulated area. As soon as an agent of a different color gets close enough (here: 5m, however, it would be better to use the visibility analyzer module presented in Chapter 2 to determine how close another agent is), it is assigned to that agent. It will follow that agent, even if another agent comes closer. The agent will follow that agent until either of the two dies.

However, in order to test the flexibility of the XML-events concept, a lot of additional parameters are sent in each `position` event, like direction of the head, agent type, and position of the arms. These can be used to render the agent more realistic, as demonstrated in Chapter 6 with the 3-dimensional and offline visualizers.

3.7 Cable Car Simulation

3.7.1 Introduction

The framework presented in Chapter 2 allows more than one mobility simulation. For example, it would be feasible to have a pedestrian simulation and a traffic simulation, both simulating the same scenario in the same area. Pedestrians and vehicles need to interact with each other. There are two kinds of interaction that is needed:

1. Pedestrians reacting to a car that is passing by, and vice versa.
2. Persons leaving or entering a car. It should be possible to simulate something like a person that arrives in an car, parks it near the start of a hiking path, and takes a hike.

To couple two simulations for interaction as in 1), the simplest choice is to use the output sent to a visualizer. Each simulation has to calculate the positions of all the agents anyway for visualizing the scenario. If this data is fed into the other simulation, it can let the agents react to those other agents.

However, this still implies major changes in the program structure: While it is not necessary to actually simulate (i.e. move) the agents of the other simulation, they still need to be represented somehow. If the agents are of a different type (e.g. cars and pedestrians), this new type has to be implemented in the mobility simulation.

This task is not that hard, but still takes time to implement. There are some problems in the mental layer as well: the agent database has to support multiple modes of transportation. And the system needs to remember where somebody has left its car when he/she is returning from a hike. In order to avoid this, we were looking for a simulation, that does not have type 1) interactions with the existing pedestrian simulation.

In the area simulated in the Gstaad-Schönried scenario are two cable cars, Horneggli and Rellerli. They transport hikers from the bottom of the valley to the peaks of two opposing mountains, and are crucial for the tourists in that region. Since the gondolas are above the hikers, they do not physically interact as in 1).

The only interaction is as in 2), when an agent enters or leaves the gondola of a cable-car. And since cable-cars are open for everybody, there is no need to remember which gondola an agent took before.

3.7.2 Implementation

A cable-car has only one degree of freedom: up and down. And since both gondolas are attached to the same wire cable, they can not run independently. The state of a cable-car can be represented in the

distance of the gondolas from their base station (summit station for one, valley station for the other). This is a single real number.

The capacity of a gondola is represented by an integer value. If there are fewer persons in a gondola, the doors are open. The doors close if either the gondola is full, or if it is time to run the cable-car.

The cable-car simulation is a simple simulation which can be used to show simulation coupling and synchronization (see Chapter 8). The only output sent by the cable-car simulation is the position of the gondolas:

```
<event type="position" agent_type="gondola"
agent="1" x="558123" y="144231"/>
```

The 2-dimensional visualizer is able to draw the position of the gondolas in a separate color.

3.7.3 Coupling

In the street network file of the simulated region, the cable-cars are represented a a single link from the valley station to the summit station. The link travel time is preset to the time it takes for the gondola to complete one run. The route generator is now able to use this link as any other when it calculates the best path. However, it does not take into account that there are discrete start times only: the cable-car does not allow pedestrian to enter this special link continuously. They have to wait at the station for the next gondola. However, the physical mobility simulations (both the pedestrian simulation and the cable-car simulation together) simulate this correctly. If the agent database listens to the events sent by the mobility simulations, it is able to figure out the actual times. If an agent is at the station early or late, it has to wait. For the agent database, this is like there is a congestion at the last link before the station, which it avoids by shifting the start times accordingly.

However, is was never investigated whether this actually works or not. The cable-car simulation was used for experimenting with synchronization (see Chapter 8) only. Based on what is visible on the visualizer's display, the coupling of the cable-car and the pedestrian simulation works fine. However, we did not look more carefully at the strategies proposed by the agent database in order to avoid waiting times at the stations.

3.8 Summary

A mobility simulation of hikers in the Alps is implemented. The ultimate goal is to have these hikers make realistic tours, where they react to visual stimuli including the presence of other hikers. This chapter presents the underlying pedestrian simulation for the project. Because of the need to allow for realistic arbitrary movement, a simulation based on continuous space was selected. The dynamic model is taken from the literature, but thoroughly tested for our purposes.

One important aspect is the computation of the forces which keep the agent on the path and make it follow the possibly winding path. Two approaches were tested: one based on a combination of location-based forces, which push the agent toward the nearest path, and attractive waypoints, which pull the agent along its route. The other model ("model B") uses a path-oriented coordinate system, with a

strong force along the path, and a weak force toward the middle of the path. For the hiking application, model B seems to be easier to handle and to generate fewer artifacts than model A.

In contrast to other pedestrian simulations, which concentrate on crowd or even panic behavior but have the advantage of relatively small spatial scenario sizes, in this project crowd behavior is less important but large scenarios need to be handled. The implementation therefore uses lazy initialization plus caching for location-based data. This means that the simulation area is segmented into blocks. Location-based data is only computed when an agent enters a block for the very first time. It is then simultaneously kept in memory and written to disk. If the simulation runs out of memory, blocks which have not been used for a long time are removed from memory but kept on disk. Such blocks can be re-loaded if another agent enters the block at some later time, or if the simulation is run a second time with the same parameters. – Since the hiking paths cover only a relatively small fraction of the whole geographical area, this mechanism makes the handling of large scenarios possible on normal desktop PCs.

The simulation system is developed for the simulation and analysis of tourist hikers in the Alps, but the design is considerably more general so that it will be possible to apply the simulation system also to related areas such as pedestrian movements in urban areas or in buildings.

Well-known cell-based models for pedestrian crowd simulations are fast, but introduce artifacts. Models based on continuous space allow the agents to move in an arbitrary direction. However, it is necessary to calculate the desired velocity $\mathbf{v}_i^o$ of the agents separately.

We presented two different approaches: the first uses a potential field in order to store the pre-computed values of $\mathbf{v}_i^o$. The second approach is based on a graph, which edges can be followed by the agents.

The two approaches were compared using a real-world example: the simulation of an evacuation of Zürich Main Station. Both fundamental different models show comparable results.

Chapter 4

Route Generator

4.1 Introduction

In (TRANSIMS www page, accessed 2005), the route generator was built as a stand-alone module. The route generator reads a file that contains *aggregated link travel times* written by the mobility simulation. In order to use the route generator to calculate shortest paths, one has to provide an *activity file* as well; routes were computed between successive activity locations. The route generator reads the link travel time, processes the activities, and writes the routes into a *plan file*. After that, it stops.

The actual calculation of routes is based on the shortest path algorithm by Dijkstra (Dijkstra, 1959). This algorithm was extended to be time dependent.

This idea was taken further for the MATSIM (MATSIM www page, accessed 2004) framework, where the route generator is file based as well. However, it is capable to read whole XML-based day plans. It figures out where complete routes are missing and fills the gaps.

The route generator in MATSIM reads a file that contains *events* written by the mobility simulation. Every agent (i.e. vehicle) writes `enter_link` and `exit_link` events, which include a time-stamp. These events are used by the route generator in order to determine the *link travel time* of each link.

In this work the route generator is a stand-alone module as well. However, it does not process files sequentially. The route generator listens on a network port for events and requests. The knowledge stored inside is updated continuously, since it listens to the events sent by the simulation in real time.

The Dijkstra algorithm was extended again, in order to calculate not the *shortest*, but the *best* path. It uses a generalized cost function instead of just the time for the route calculation.

The major differences to the route generation module from MATSIM are:

1. **Network Based Communication**: The route generation module is now a standalone application that is started as a *network daemon* (comparable to a *server* known from the Internet. To a certain extent, the route generation module behaves like classical server of a *client/server architecture*). Once started, it reads in the network definition file and waits for input from the network.
    - **Events** are received continuously from the network and are integrated into the mental rep-

resentation of the world in the moment they arrive. The next route request will be answered based on the new state.

- A **Route Request** triggers the calculation of the best path according to some parameters. These parameters are also sent within the route request. The answer is sent back over the network.

  The **cost** and the **travel time** of the path are **returned** as well.

2. By using a **Generalized Cost Function** the route generator module is now able to calculate the best route (according to the preferences given in the request) to the destination.

The route generator used here is an extension to one used by MASTIM. It is based on the same source code, which was modified and extended to provide the features presented in this chapter.

In this chapter, the internal knowledge representation is explained (Section 4.2), we focus on the temporal representation (4.2.1), as well as on the difference between individual and global knowledge (Sections 4.2.2 and 4.2.3). The algorithm which the module uses to actually calculate the route is given in Section 4.3. How the route generator module listens and processes events is shown in Section 4.4. The message interface and its syntax is demonstrated in Section 4.5.

4.2 Knowledge Representation

4.2.1 Temporal Representation of Knowledge

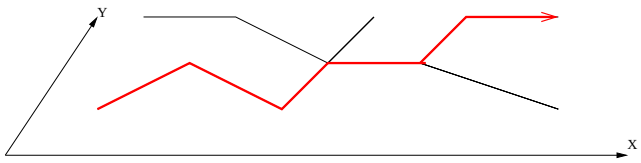
The route generator module used by MATSIM uses multiple *time bins* per link to store the link travel time. In a traffic simulation, it is obvious that the time it takes to drive along a link depends on the time-of-day. It might be congested in the rush hours, which affects the link travel time. These times are therefore stored in 15 minute intervals.

The algorithm of Dijkstra was modified in order to reflect this time dependency. While traversing the graph (street network), the link travel time which corresponds to the virtual time of the vehicle simulated is used as a link weight (Figure 4.1b).

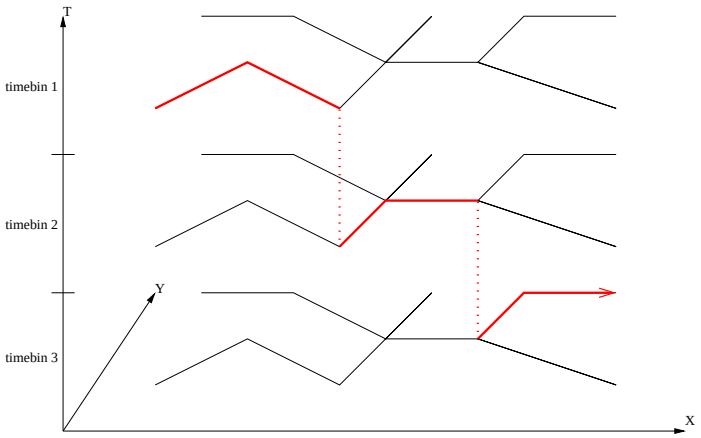
This is not uncommon for the human brain: if someone plans a trip from Zürich to Spain, he will carefully adjust the start time in order to avoid the rush hour in Lyon during the morning. This also takes into account that traveling from Zürich to Lyon takes about 6 hours.

In a hiking simulation, there is no congestion. Link travel time is (more or less) the same all the time. However, the other parameters to the *generalized cost function* might change, e.g.:

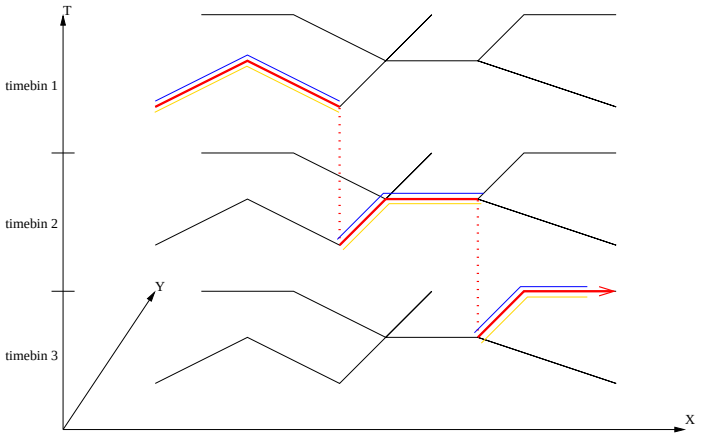
- The sun is shining during the day only.
- The sun is shining in the morning on the mountain slopes facing east, and in the evening on these facing west.
- Cable-Cars are closed before, say, 10:00 and after 20:00.
- Temperature changes during the day.



(a) Original Dijkstra



(b) Time Dependent Dijkstra



(c) Generalized Cost Dijkstra

Figure 4.1: The original algrithm of Dijstra searches the shortest route (red) on a plane graph (a). The time dependant algorithm uses a different graph (same topology, different weights) for every time bin. This graph is switched if a new time bin is reached (b). If a generalized cost function is used, the weight of the links is calculated out of several attributes (colors). However, for determinig the time bin, still travel time (red) is used.

- Weather forecast predicts rain for the afternoon.

A time bin for every attribute is needed. This multiplies the amount of memory needed by the number of attributes. In order to reduce this again, the duration of a time bin was changed from 15 minutes to 2 hours, which means that instead of 96, only 12 bins are needed per day. However, this also reduces the route generator's ability to react to show events (like to a cable-car that is closed for 15 minutes). In a production environment, this duration need to be changed again to a reasonable value.

4.2.2 Individual Knowledge

No person is able to know all the links of the street network everywhere. It is possible to build a graph representation that uses just the known links for each agent. If a destination is outside that graph, the agent has to *explore* the area.

Helena Unger shows (Unger, 2002) a way to achieve this goal for one single agent, using neural networks. In (Gloor, 2001) a implementation of that method for a large number of agents was presented. However, in order to allow the calculation of hundreds of routes per seconds, the algorithm was simplified.

Unger's algorithm gives the agents some knowledge that is not represented in the graph (e.g. sign posts in the network). It is clear that for hikers in an alpine area, *a priori* knowledge is important. For example, the hikers can consult a map of the area, or ask the tourist office for nice hike routes.

Such knowledge can be represented in a street network graph as well. The recommendations from the tourist office (or other hikers) can be represented by adjusting the weights of the graph accordingly. The hiker expects the actual streets to be like that, but his expectation could prove wrong as soon as he actually hikes on that particular street. The route generator will then learn the correct, individual weights for that link.

In the current implementation of the router, which is simpler and more straightforward, every action is based on the graph of the street network. Since each link contains N weights (of size S_{weight}) for each of T time bins (see 4.2.1), the memory consumption M is:

$$M = N_{weights} * Size_{weight} * L_{links} * T_{timebins} \tag{4.1}$$

If the number of links L_{links} is about 20'000 as used in the Gstaad/Schönried scenario, the memory M is about 15 Megabytes.

Ideally, there would be one route generator for every agent in the simulation. The same could be achieved by using one router, but a separate graph for each agent. Using this graph, which contains the experience of one single agent, the route generator would be able to calculate a route that is based on the individual knowledge of that agent.

However, with the above numbers, each agent would consume 15 Megabytes of memory. Since on a typical summer day there are more than 100 hikers in the Schönried area, this results in more than 1.5 Gigabytes, which is too much for a typical PC.

4.2.3 Global Knowledge

The opposite of the individual route generator as presented above is a *global route generator*, which shares the knowledge of all the hikers. In such a route generator, all information collected for one agent (e.g. link travel times) is accessible for all agents in the system. The knowledge is global.

Since all the link travel time information are accumulated, this assumes that all agents travel equally fast. This is the case in traffic simulations, at least in congested areas. However, with hikers, every agent can walk using the speed it wants. This could lead to inaccurate results if the link travel time changes a lot from one time bin to another (e.g. with public transportation facilities that run on a certain schedule). A possible solution to this issue would be to include the average walking speed of an agent into the route request. The route generator could then scale the link travel times accordingly. However, this solution assumes that walking speed can be scaled linearly, which is not necessarily the case (e.g. i is possible that two hikers would walk at the same speed as long as the area is flat, but one of them is faster if the path goes up- or downhill).

A global route generator uses the same input for all agents. However, the routes that are calculated still depend on the individual preferences. Every agent notifies the router how a certain link is (e.g. steepness, how many other agents are there, if it was raining). The router keeps track of those attributes *without valuating* them. This is done as soon a route request is processed, and therefore independently for each hiker.

The agents seem to learn from each other about the attributes of a certain link. One agent has experienced something, and the others can use this information in their route calculation. This is as they would talk to each other and share their knowledge. However, using this mechanism of one global graph, there is no need that two agents meet in order to share their experience. The knowledge travels infinitely fast from one agent to all the others.

4.3 Route Calculation

4.3.1 Shortest Path Algorithms

A shortest path algorithm calculates the fastest path through a graph. The way it is used here, the input is:

- the graph: edges represent streets, nodes represent intersections.
- start and end location
- the weights of each edge (link): usually travel time, or a generalized cost function (see next section)

Such an algorithm usually starts at the start node, traverses the graph and calculates the travel time from each node to the start node (using a lookup table for the weights, i.e. the *time bins*). Eventually, it will reach the destination node.

To get an route out of this, the graph is traversed backward starting from the destination node.

See Appendix A for the description of the algorithm by Dijkstra (1959). This algorithm uses a clever internal management of nodes and a better stop criteria (stops before every node is touched).

4.3.2 Generalized Costs

As said before, for a hiker, the fastest path is not necessarily the best path. Therefore, in each route request, a table of preferences is provided. This preferences vary from hiker to hiker, and can also be dependent of the time of the day or earlier experiences.

Instead of the link travel time, the following equation is used to calculate the weight of a link:

$$\left(\frac{\sum_{attr} weight_i * atan(attr_i)}{N_{Attr}}\right) + traveltime \tag{4.2}$$

This says that each $attribute_i$ is multiplied by the $weight_i$ (preference). This yields in a route that differs from the shortest path, as shown in Figure 4.2 and 4.3.

The $atan()$ in Equation 4.2 is used to level out the impact of events after a certain number of them was received. This makes sense since it matters if one or ten raindrops are recognized, but after, say, 100 raindrops, additional 10 do not make much difference anymore, it is raining anyway. The same principles applies e.g. to cows on a field or to trees. However, this is just an attempt to find a feasible formulation in order to test the implementation. This formulation is not based on any psychological background. It is also not clear if the route cost of single path segments can be added together linearly.

According to Equation 4.2, the travel time has a fixed weight In a future implementation of the route generator, it should be possible to provide a preference for the time as well. At the moment, the preference of the travel time can be changed by scaling all the other preferences by a certain value.

It is very important that the route generator internally also keeps track of the time separately. The time is needed to select the correct bin containing the attributes (see Figure 4.1c and Appendix A).

Assume that the area near the mountain peak is crowded during lunch time, and hikers do not like this. If someone starts hiking in the morning, it might take several hours to reach the mountain peak. The route generator has to calculate the absolute time the hiker is at the peak (using a certain route). If the hiker is there before or after lunch time, this route is OK. If the hiker reaches the top at noon, it might prefer another route that is i) faster, ii) slower, or iii) avoids the peak at all. Note that the fact that the peak area is crowded does not affect the travel time of the router.

In the route answer, both *travel time* and *travel cost* are returned. Using this, the agent database is able to ask for two routes to different destinations, and compare the time and cost against each other. An example using this feature is presented in Section 9.2.2.

4.4 Listening to Events

The *route generator module* (*router*) is located at the first mental layer (see Figure 5.1), which receives activity locations from above (e.g. hotel, restaurant) and sends complete routes to the simulation.

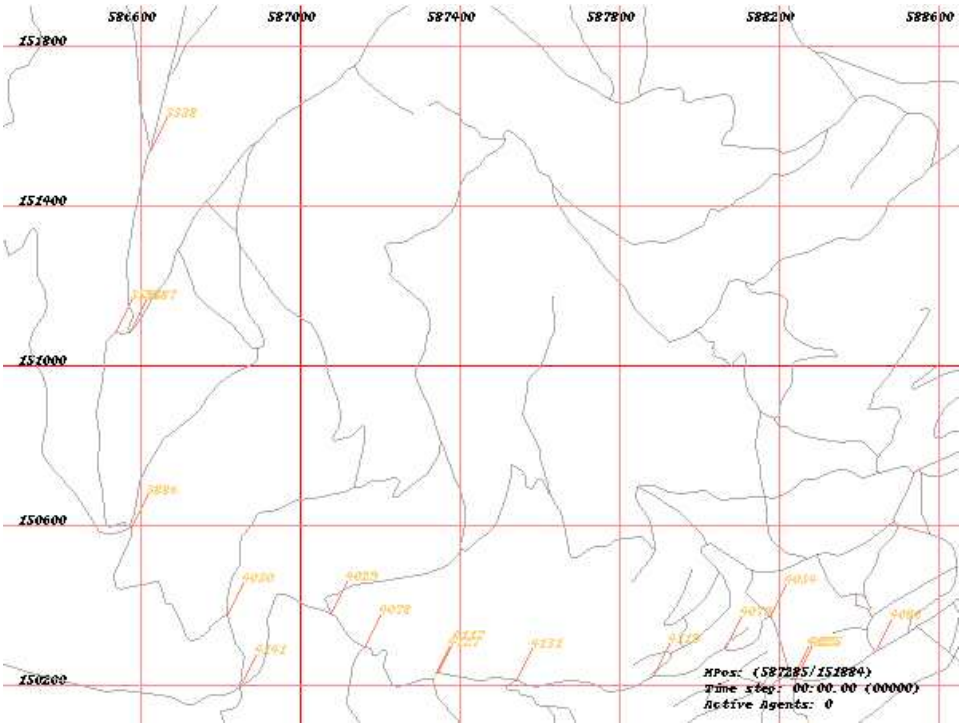


Figure 4.2: The 2-dimensional visualizer is able to display `route` *events sent by the router as an answer to a* `route_request`*, represented by tags with the node number. Here, a route from the hotel (4477) to the mountain top of Rellerli (3623) was requested. No other criteria than the shortest distance was applied.*

The router's main task is to answer `route_requests` messages for the agent database. In order to build its view of the virtual world, it has to listen to events sent by the simulation and adjacent modules first.

The router subscribes to the event channel and listens to event. It listens to

- spatial events sent directly from a module (e.g. rain, sun)
- spatial events sent from an agent in the simulation (e.g. pedpressure, view, forest)
- non-spatial events from the simulation (e.g. enterlink, exitlink)

4.4.1 Direct Spatial Events

Humans are able to *deduce* certain facts without having to *experience* them. For example, if the sun is shining on the other side of the valley, a human individual can deduce that it will be warmer there. The agent does not have to walk to the other side first to perceive the sunshine. The fact that there is sun somewhere else can be considered in route calculations without a delay.

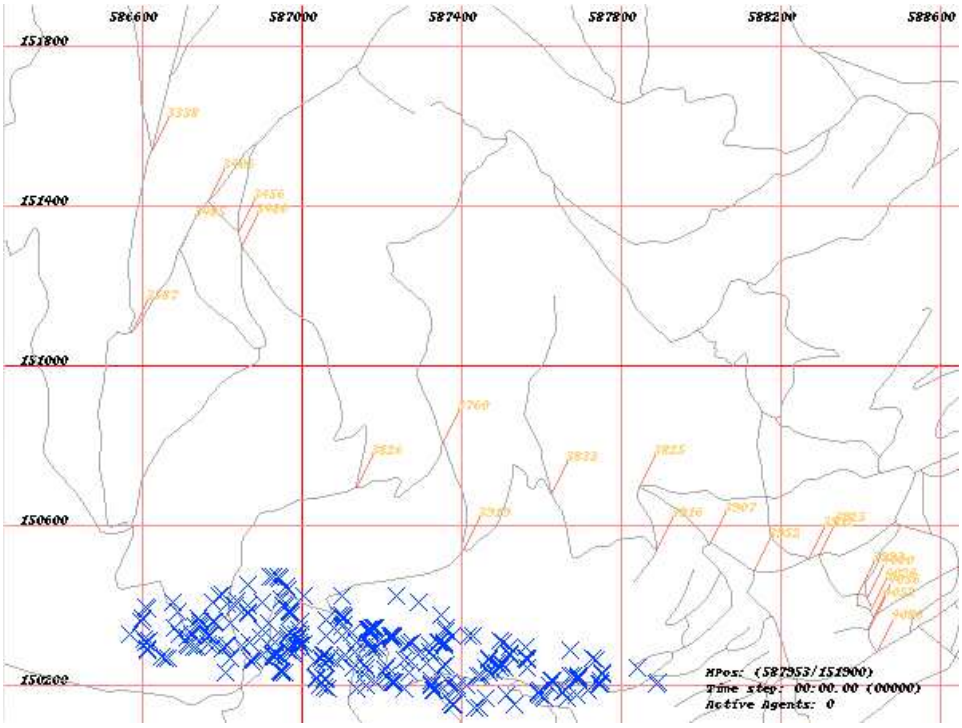


Figure 4.3: A route from the hotel (4477) to the mountain top of rellerli (3623) after it started to rain in the area where the shortest route was before (See Figure 4.2). The preferences in the request told the router to avoid rain (blue).

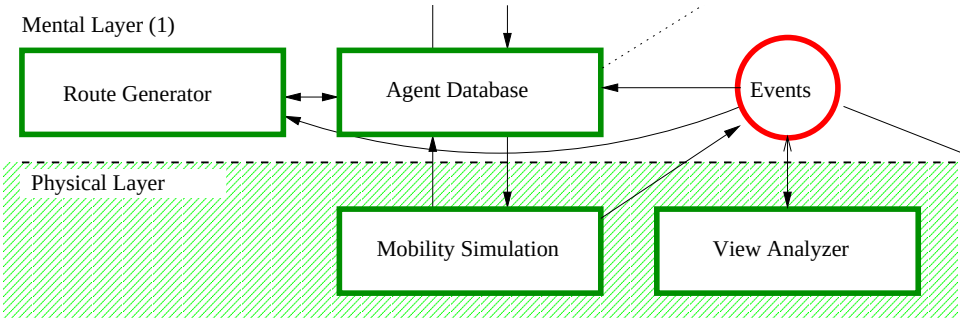


Figure 4.4: The routers main task is to answer `route_requests` *for the Agent Database. In order to build its view of the virtual world, it receives events (through the* events channel*) from the simulation, and adjacent modules, like the view analyzer module.*

This type of events are called *Direct Spatial Events*. These events are typically fed into the system from a module from the outside. One can think of a *weather module* that sends events related to the current weather.

The router receives and processes these events at the moment they are sent. This means that

- the fact is learned immediately, which means that the information will be used in the next route request,
- all agents know the information immediately, there is no need for any agent to explore the area first.

Here, as an example, a `rain` and a `sun` event:

```
<event type="rain" time="08:05.11" x="586963" y="150308"/>
<event type="sun" time="08:05.11" x="586965" y="150312"/>
```

Note that there is no a priori *valuation* stored in the event. Even a rain event reports just the fact that there is rain. Whether this is something good or not is something the modules in the mental layer have to decide.

4.4.2 Indirect Spatial Events

Some things are not known before an agent has experienced them. An example is rain inside a building. It is known globally that there is rain all over the place, but the fact that an agent does not get wet if it is inside a building is not clear from the beginning. Therefore, the `hitbyraindrop` event is sent by the simulation, if an agent was hit by a raindrop. This is the case if i) there was a `rain` event near the agent, and ii) the agent is not inside a building.

```
<event type="hitbyraindrop" id="4" time="08:05" x="586961" y="150324"/>
```

Of course one could argue that with a little more *intelligence* the agent could figure this out without actually exploring the world. But in this work, one of the goals is to replace logical reasoning by trial and error.

Also events generated by some modules adjacent to the mobility simulation (e.g. view analyzer, see Section 6) belong into this category.

```
<event type="forest" id="4" time="08:05.12" x="586961" y="150324" />
```

4.4.3 Personal Spatial Events

There are things that an agent experiences without a direct connection to the environment. Being tired or hungry are not necessarily related to the place the agent is at.

However, the fact that there are too many agents at the same place, and the agent feels uncomfortable, might be related to a geographic feature (e.g. a narrow bridge, a sharp corner). Therefore it makes perfect sense to treat these events as *spatial*.

In this implementation of the router, all knowledge is global and shared by all the agents. It would be better to keep personal experiences separate. But again, the fact that there are a lot of other agents nearby does not mean necessarily that an agent does not like the fact. The valuation is still up to the agent that *receives* the information.

4.4.4 Non-Spatial Events

Some events are not related to a certain place. For example the `start_activity` event does not specify, where this activity takes place, but which activity. To correlate this activity with a location is up to the mental layer modules.

```
<event type="start\_activity" id="4" time="08:05" activity="31"/>
```

However, the link-based events are kind of spatial. Although there is no co-ordinate in the event, the meaning is bound to a certain link, which has a known position.

```
<event type="exit\_link" id="4" time="08:05" link="31"/>
```

4.5 The Message Interface

This version of the Router is built in a way that it works as an independent module that receives its tasks over the network. However, it should be possible to create a router that can be linked to the agent database binary in the same way. The route request

```
<event type="route\_request" from="4160" to="4471" time="1"/>
```

is equivalent to this function call:

```
route\_request(4160, 4471, 1);
```

This will increase the execution speed, since there is almost no latency for function calls. However, there come some disadvantages with this approach:

- It is not simple to change an argument. A way to pass arguments as *keyword value pairs* would be very useful: `route_request(from="4160", to="4471", time="1");`.
- The router and the agent database share the same memory. Both programs use a lot of memory to store its internal representation of the simulated world. If they communicate over the network, they can easily be separated on to computational nodes in the cluster.
- It would not be possible to couple a C++ Router to a JAVA agent database. Only modules written in the same programming language can be coupled (although there are wrappers available that allow the linking if certain languages into others).

Basic Request and Answer Format

Route requests are in the events format (see Chapter 7). The minimal `route_request` event states the *from* and *to* link and a time:

```
<event type="route_request" from="4160" to="4471" time="1"/>
```

The route generator calculates the fastest route and sends the answer back via the events channel as a `route` event:

```
<event type="route" from="4160" time="1" to="4471"
       route="3826 4030 4141 4080"
       [..]
/>
```

It would be possible to use a dedicated communication channel between agent database and route generator, since usually, this is point-to-point communication. If a route generator is used that completes dayplans as the router in MATSIM does, there must be a separate communication channel. It is not possible to send the plan XML format through the events channel (see Chapter 7). However, since this route generator uses the simpler XML format, it is possible to use the events channel. There are advantages for this approach: i) it is simpler to implement, since the router and the agent database already have to listen to events, no new communication code is needed, and ii) since the events channel is a broadcast channel, every other modules is able to listen to the answers as well, which makes debugging much more comfortable (Chapter 6 shows an example how this information can be used).

A former version of the route generator module was able to send the internal representation of links to the visualizer as well. It was possible to define a threshold for a certain attribute. Of all links, whose attribute was above that threshold, the link ID was sent to the 2-dimensional visualizer. Using this mechanism, debugging the learning process was visible.

The route generator includes all the information that are in the request into the answer (from, to and time in this example). It does include all attributes into the answer, also the attributes it did neither use nor understand. This can be used in the agent database as well for maintaining the state of the request, which allows the *agent database* to be stateless, since it does not need to cache the details of every request it has submitted.

It is possible to include attributes like `agentid` or `requestid`, to keep track of the requests.

Request including `requestid="id1"`:

```
<event type="route_request" from="4160" to="4471"
       time="1" requestid="id1"
/>
```

Route answer including `requestid="id1"` as well:

```
<event type="route" from="4160" requestid="id1" time="1" to="4471"
       route="3826 4030 4141 4080"
       [..]
/>
```

The value of the attribute `type` is always changed from `route_request` to `route` in the answer.

Including Generalized Cost

In order to use the generalized cost function of the route generator, the preferences of the hiker have to be provided in the request:

```
<event type="route_request" from="4160" to="4471"
       time="1" requestid="id1"
       weight_forest="-0.2" weight_pedpressure="1"
       weight_rain="0.5" weight_sun="-0.5" weight_view="1"
/>
```

The route answer is calculated based on these preferences. The preferences as provided in the request are given back as well:

```
<event type="route" from="4160" requestid="id1" time="1" to="4471"
       weight_forest="-0.2" weight_pedpressure="1"
       weight_rain="0.5" weight_sun="-0.5" weight_view="1"
       route="3826 4030 4141 4080"
       [..]
/>
```

It is possible that the route generator has learned additional attributes from some events. This can happen for example if a module like the view analyzer module is introduced into the system. The route generator learns the new event, and would be able to answer route requests that include the corresponding preference immediately[1]. If the preference is not given, the route generator sets it to 0 internally, which means 'do not care'.

Including Travel Time and Travel Cost in Answer

As mentioned earlier in this chapter, one of the benefits from using XML as communication language is the possibility to *extend* the messages by additional attributes. This is used here to give back also the calculated travel time and travel cost, which can be used by the agent database to estimate the value of visiting a destination. The actual valuation of the links is done in the router based on the provided preferences of the hiker:

```
<event type="route" from="4160" requestid="id1" time="1" to="4471"
       weight_forest="-0.2" weight_pedpressure="1"
       weight_rain="0.5" weight_sun="-0.5" weight_view="1"
       route="3826 4030 4141 4080"
       estimated_cost="1585.200000" estimated_time="1585.998550"
/>
```

4.6 Conclusion

The route generator that was presented in this chapter is based on the route generator by MATSIM. Its code was in a first step modified in order to be network based. This was needed that it could fit into the framework used in this project.

In a second step, the generalized cost function was introduced. For this, not only the internal code of the route generator had to be modified, but also the message interface. The XML language has proven

[1] In the current implementation, one has to match the name of the preference with the name of the event. An internal table associates e.g. the event "nice_view" to the preference "weight_view". As soon as the names of the events are more uniform (i.e. "view" instead of "nice_view") this table could be dropped in favor of a general matching algorithm.

to be a flexible solution that allows such addition without breaking compatibility to older versions.

Finally, the ability to return the estimated travel time and travel cost in the route answer was appended. Again, this was possible because of the open message format provided by XML.

The route generator is now more powerful compared to his ancestor. It is, however, also slower, which is mainly because of the network based interface. It uses more memory as well, the internal representation is more voluminous because of the multiple attributes it has to store.

Chapter 5

Agent Database

5.1 Introduction

The agent database is, as stated in Chapter 2, an essential part of the learning process. The functionality of this module is to

1. connect the layers of the network,
2. provide a central interface to the layers higher up, incl. hiding the complexity of the layers underneath,
3. cache plans and their score that were simulated before,
4. make sure that bad plans are not executed again,
5. react to unexpected events from the mobility simulation.

In this chapter, two different implementations are presented. The agent database from the traffic project fulfills most of these requirements already. However, it does not fit into our framework nicely (Section 5.2). A agent database that is much simpler but fits the hiking scenario is presented in Section 5.3.

How these agent databases are used is described in Chapter 9.

5.2 Agent Database in the Traffic Project

In traffic simulations, one problem with the basic approach is that it gives rise to oscillations from one iteration to the next: If, say, route A is faster in one iteration, then many travelers will switch to it, making it slower, and some other route faster (Raney and Nagel, accepted). In order to suppress these oscillations, in every iteration only 10% of the agents ask the router for a new route. The others are stick to the route used before.

Since there is no congestion in a hiking simulation, these oscillations do no occur. However, it would be possible to use a criterion that the hikers do not like to walk near other hikers. While the route

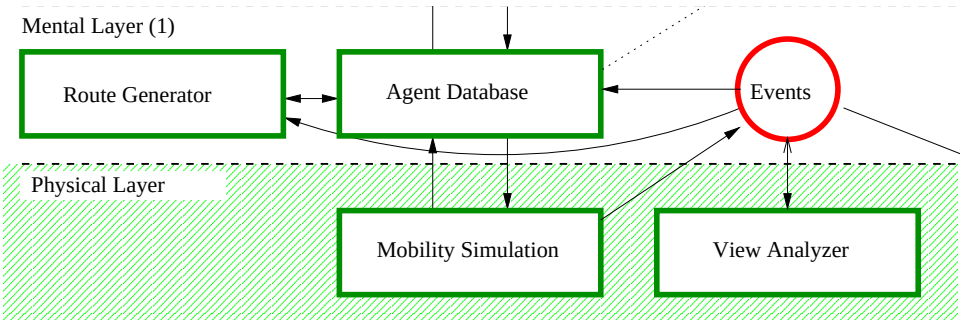


Figure 5.1: The agent database is a central part in the framework. It connects the modules in a layer (left, rigt) and adjacent layers as well (up, down).

does not get slower, it still gets a bad score based on events received. Also at the station of a public transport facility congestion could occur, at least if the density of the hikers is increased. In the current implementation of the hiking simulation, no oscillations were observed.

The agent database from the traffic project uses a file-based interface. It would possible to write an wrapper that enables the brain to perform at least the basic operations via the network: reading events, and to send routes. However, intra-day-replanning would not work. This is described in more detail by Raney (2005).

An example how the agent database taken from the traffic project can be used without modification for the hiking simulation is shown in Chapter 9.

5.3 The Simpler But Networked Agent Database

5.3.1 Functionality

The development of a sophisticated agent database was not part of this project. Originally, the idea was to take the agent database from the traffic project. However, that project decided to use a file based approach (basically because of maintenance and performance reasons, see Raney, 2005).

The core functionality of the agent database, as needed for our purpose, is as follows:

1. receiving activity plans (from the layer above, in our framework),
2. ask the route generation for a route to connect these activities,
3. pass these new route plans down to the mobility simulation, and
4. score the result and decide what to do next.

The actual collecting of executed route plans is not really necessary at this stage. However, in order to fill the hiking simulation with more realistic behavior, something like this will be needed again in future.

In addition to the functionality described above, we needed these features as well:

1. networked communication,
2. ability to use a generalized cost function for scoring,
3. a simple way to test new features, like the travel time and cost estimation of the route generator.

Since we did not want to develop a highly configurable multi-purpose agent database program, we decided to write several Perl scripts that could be modified according our need at the moment.

The skeleton of the Perl agent databases is responsible to establish the communication to the other modules in the framework. The agent database has to connect to the events channel, and establish a "braincomm" channel (see Chapter 7) to the microsimulation. According to the framework (see Chapter 2), it would be necessary to open another "braincomm" interface for the agent database in the next higher layer. Activities, however, are read from file or stored in the source code for simplicity. So there is no need at the moment to listen from incoming connections from above.

The parsing of the XML events stream was achieved without the use of an XML parser. Since the events stream uses a subset of the XML specification, it was possible to implement a simple and fast solution using regular expressions (see also Section 7.10).

The first regular expression looks for the < and /> characters:

```
($events) = $eventbuffer =~ s/<(.+?)\/>//;
```

The attributes can then be accessed using this expression:

```
($agentid) = $events =~ /agent=\"(.+?)\"/;
```

To write the XML route plan to send to the mobility simulation is possible without XML support as well, since this is just a text block formatted according to a special specification (XML).

After communication is established, the agent database script enters the *main loop* which:

- checks the events channel for incoming events. If any are available, they are put into a queue for later use,
- if there is now an event in the queue, it is extracted and parsed. Depending on the content, a further action is taken.

The module is called "agent database", because it memorizes past plans of the agents, and their performance (Raney, 2005). An additional functionality of the agent database is that it mediates between different modules, in our case between the incoming events stream, the route generator, and the higher level behavioral modules. For some functions, it is possible to reduce the agent database module to this mediating function, making it stateless in these cases.

However, the events still carry a state for each agent. According to an internal state machine, each event is followed by an appropriate action. This state machine is different for each implementation of the agent database (See Figure 8.1 for an example).

The state, which each agent is in, is sent to the other modules as well. For example, if the agent database asks the router for a route, it sends all the information related to that agent to the router. In the answer, everything the agent database has to know about that agent is included again:

```
<event type="route" from="4477" to="3623"
       time="0" agent="1"
       weight_forest="-0.2" weight_pedpressure="1" weight_rain="0.5"
       weight_sun="-0.5" weight_view="1"
       route="4086 4125 4122 4034 [..] 3886 3593 3587 3338"
       estimated_cost="1585.200000" estimated_time="1585.998550"
       request_id="routeB"
/>
```

In this case, this is the second of two requested routes (`request_id="routeB"`) for an agent (No. `1`).

5.3.2 Implementation(s)

The high-level programming language *Perl* is derived from the C programming language and from Unix tools like sed, awk, the shell. It is suited for quick prototyping and for text manipulation. The implementation of a state machine used by the agent database including the communication parts takes not more than 200 lines of code.

It is, at least for a research project, feasible to implement a new agent database for almost every test case. Based on a skeleton, which handles the basic communication and event parsing, this is a straightforward task.

One feature that was implemented in such a Perl agent database was its ability to respond to messages that indicate an “emergency” for the hiker, e.g. he detected rain nearby or a public transport facility does not work as expected.

The following excerpt from the source code of the agent database which reacts to raindrops shows the principle. As soon as an agent detects a raindrop, it heads home immediately:

```
if ($e =~ /type=\"hitbyraindrop\"/) {
    # parsing the XML message:
    $ev = $e;
    $ev =~ /agent=\"(.+?)\"/;
    $agent = $1;
    $ev =~ /link=\"(.+?)\"/;
    $link = $1;

    # request route if agent was not sent home before:
    if ($action[$agent] ne 'senthome') {
        $action[$agent] = 'senthome';
        # stopping the agent immediately
        print $brain "<stop agent=\"$agent\">\n";
        # requesting route from current position to hotel
        print $event "<event type=\"route_request\" ";
        print $event "from=\"$link\" to=\"$hotel\" agent=\"$agent\" ";
        print $event "time=\"0\" weight_rain=\"1\"/>\n";
    }
}
```

After the execution of that code segment,[1]. the agent database is polling the network for incoming events. If one of these events is the matching route answer, it is sent directly to the mobility simulation via the braincomm channel.

Note that sometimes it is necessary to keep track of the state of an agent internally. In this case the fact that an agent was sent back to the hotel already is conserved, otherwise the agent database would ask for a new route from its current position to the hotel every time it is hit by a raindrop. This state is stored in the agent database for simplicity and better performance. However, in a state-less agent database, this information could be sent to the mobility simulation, which could include this information in every event sent to events channel. This would increase the bandwidth usage unnecessarily.

During this project, besides the agent database from the traffic project, these agent databases written in Perl were used (see Chapter 9 for the related sensitivity studies):

- A similar agent database that reads in activity plans from file, but does not keep track of the scores internally. It just forwards the completed route plans from the router, i.e. learning is done in the router only.
- A version that selects one out of two possible activities according to the route answer.
- A version that reacts to events immediately, as seen before (rain).
- A version that schedules random activities for every agent. This can be used to load-test the mobility simulation or the whole framework. The agents walk from one activity location to another, without any reason. They do, however, learn the best routes between them and they explore the the region quite well.

5.4 Summary

The agent database is the central module in each mental layer. It connects the layers and does the necessary data abstraction. It is possible to use the agent database from the traffic project for the hiking simulation as well. However, it turned out that there are too many differences in these two projects.

A simple agent database skeleton was written in the Perl language. This skeleton provides the basic features as communication and XML parsing. For the parsing of the XML events stream a regular expression based approach if sufficient because of the simplified structure of the event messages.

Using that skeleton, several agent databases have been written. They support the use of a generalized cost function for plan scoring, and are able to react immediately to unexpected events in the simulation.

[1] In this example, the event is sent by three successive `print $event`. This yields usually in three packets on the network as well. However, this does not introduce problems, since the TCP Multicast Daemon correctly separates events according the < and /> characters. See Chapter 7.9

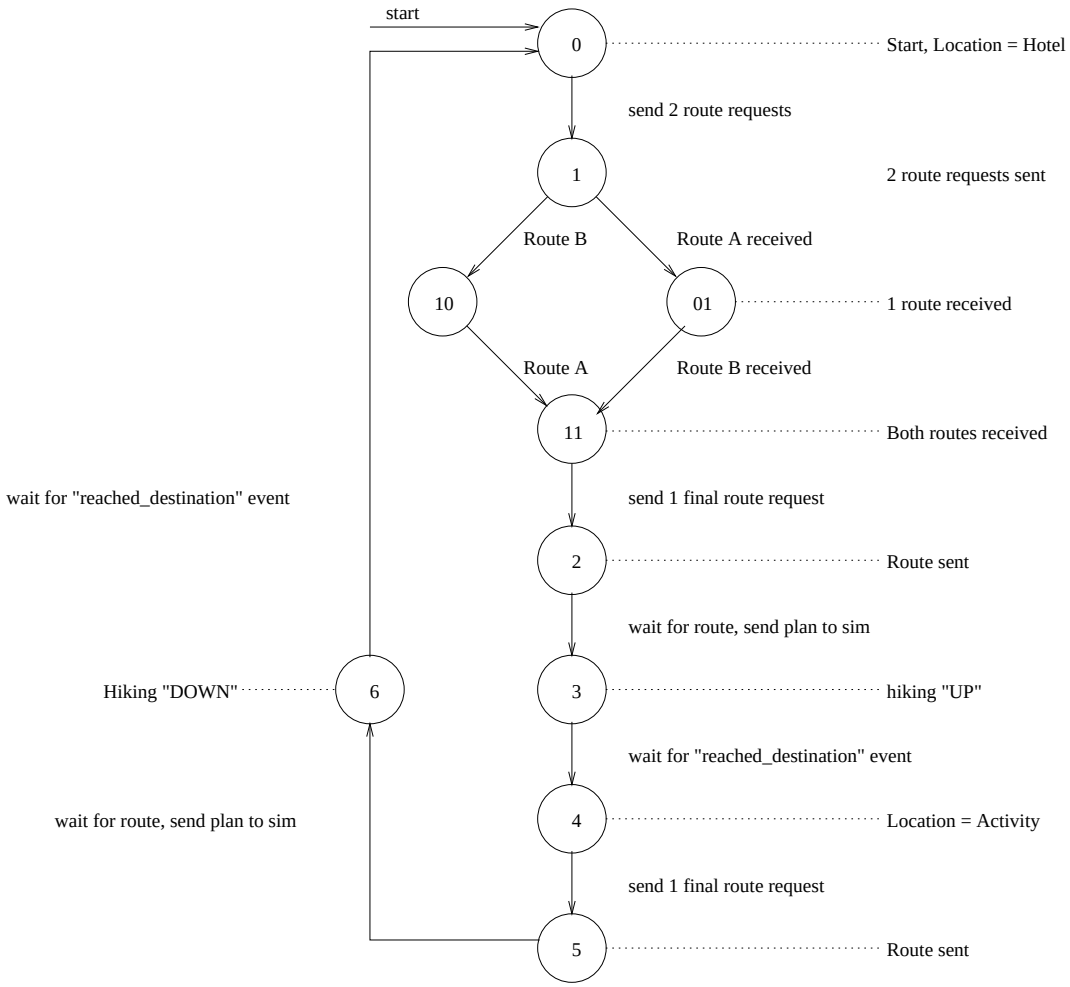


Figure 5.2: Flowchart of an agent database. It polls the route generator to determine which activity might be better. The agents hike up and down, each time they chose the activity that fits their preferences better. Note that the final route needs to be requested again because in the decision phase the agent database stores only the scores of the different alternatives. See Section 9.2.2.

Chapter 6

Visualization

6.1 Introduction

In order to visualize simulation runs, we have developed *viewer modules*, or *visualizers*. These are stand-alone applications that connect to the system via the event channel (see Chapter 7). Visualizers are built that they can directly plug into the live system. This allows the user to look at the scenario from a bird's eye view and observe how the agents move.

We have implemented two different kind of visualizer modules:

- **Real-time visualizers.** One kind of visualizer module displays a real-time view in 2D or 3D.

  The 2-dimensional real-time visualizer is suited for situations where a lot of detailed information is needed, for example while debugging. This is the main focus of this chapter, starting in Section 6.2.

  The 3-dimensional real-time visualizer has been implemented (by Duncan Cavens, Section 6.3). The user can move independently of the agents or can attach the camera viewpoint to a specific agent and see the landscape through the eyes of the agent.

- **Offline visualizers.** The other kind of visualizer module renders a more realistic image of the scene, but runs too slowly for a real-time representation (Section 6.4).

All the visualizer modules connect to the simulation using the same protocol. Internally, they share some of the data structure.

It would be possible to integrate the visualizer directly into the mobility simulation module. In fact, this is what most other simulation packages do (see e.g. de Palma and Marchal, 2002; Waddell et al., 2003; Repast www page, accessed 2003).

The arguments for our modular design from the standpoint of the visualizers are:

- It is possible to attach different visualizers to the same simulation at the same time. One can imagine a scenario using a 3-dimensional bird's eye view and at the same time a 2-dimensional view of the simulated area with additional details.

- It is possible to use multiple instances of the *same* visualizer to display different things. For example, it might be needed to show the simulation from the perspective of two agents while they interact.
- If the visualizer executable consumes too many CPU cycles, this does not affect the execution speed of the mobility simulation. To a certain extent this holds even if these two modules run on the same machine, because the available CPU time is divided evenly between the two modules.
- It is possible to run the mobility simulation on a different computing platform than the visualizers. For example, we use a Linux Beowulf cluster for the mobility simulation, but like to display the results on Windows workstations as well as on Linux workstations.
- In order to port the simulation to another computing platform, it is much easier to port one module after the other. And since there is a well defined interface, code complexity is much lower.

It is possible to connect a visualizer directly to an event stream, which allows to look at a running simulation in "real time". The visualizers are capable to connect to a `TCP_MC` channel, or listen to `UDP multicasts`. Using `TCP_MC`, the channel daemon has to be running before the visualizers start. There is no such limitation for multicast events (See Chapter 7). It is possible to mix those kind of events for use in a visualizer. Both events types are fed into the same *class* of XML-parser internally. However, they use different *instances* of the parser, which means that there is no overlapping or collision with the events.

There can be any number of visualizers connected to an event stream, each showing the same scenario from a different perspective, or displaying different accumulations of events. However, the position in time is the same for each visualizer, since this information is sent via the events channel.

If a time other than the actual simulation time is needed, it is possible to use the recorder/player modules, which act like a VCR. The recorder module records the live events steam to a file, while a player can read the file and send the data stream to the visualizer exactly the same way it would come from the mobility simulation directly.

6.2 The 2-dimensional Visualizer

6.2.1 Output

The main reason for the visualizer is to visualize the scenario we are simulating at the moment. It shows how the physical world inside the mobility simulation looks like at a certain point in time.

Visualizers are like cameras, showing the simulated world. However, not all the time a 3-dimensional, photorealistic view is expected. Sometimes, a map-like representation is most useful. The 2-dimensional visualizer presented in this section is simple, small and reasonable fast.

The 2-dimensional visualizer is able to show a large area (see Figure 6.5), but it can also be used to zoom into a very narrow area, for example to study the behaviour of a single pedestrian (See Figure 3.4).

It is able to display different kind of information:

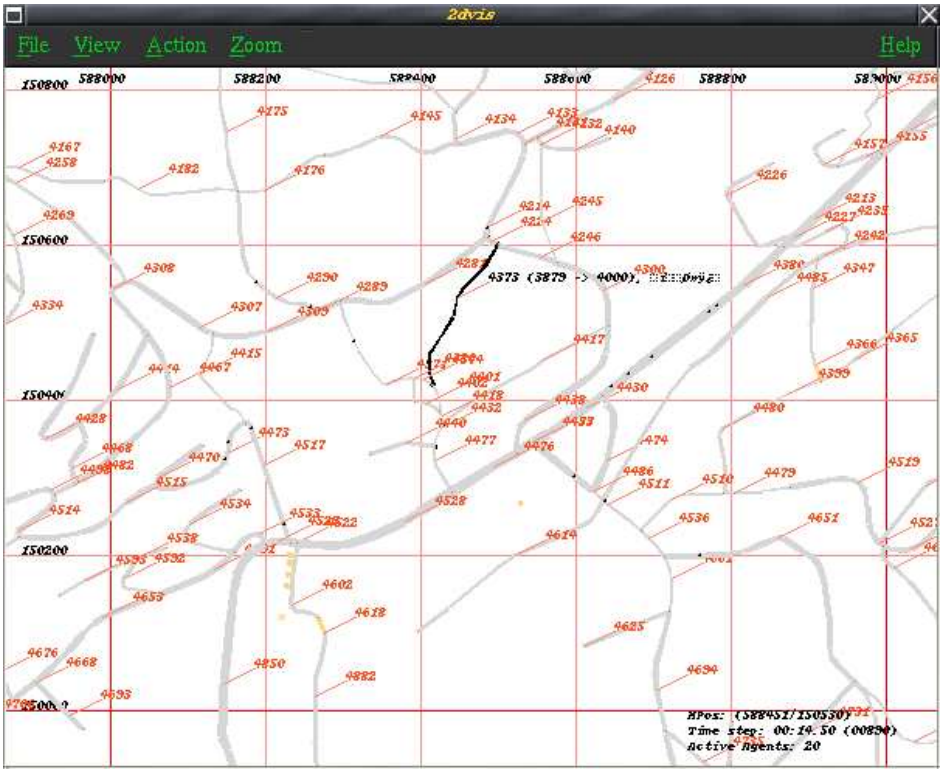


Figure 6.1: The 2-dimensional visualizer displays information related to the street network. It shows link (orange) and node IDs (not shown here), the street name, if given in the network file. The class of a street is represented as different width of the links. Also it shows additional details of the scenario like trees (green), if available.

General Information

The visualizer displays some general information that is useful in order to see what is going on in the current simulation run. This information is displayed on the right bottom corner of the window (see e.g. Fig. 6.1).

- Global simulation time, as close as we can achieve, see Chapter 8.
- The number of active agents in the system. If an agent is in the system, but has not moved yet, this agent is not known to the visualizer, since there was no `position` event yet.
- The X and Y co-ordinates of the mouse pointer in the simulated world, e.g. 588443 / 150262 for a hotel. This is useful to figure out where something happened or to look up co-ordinates of items (e.g. nodes).

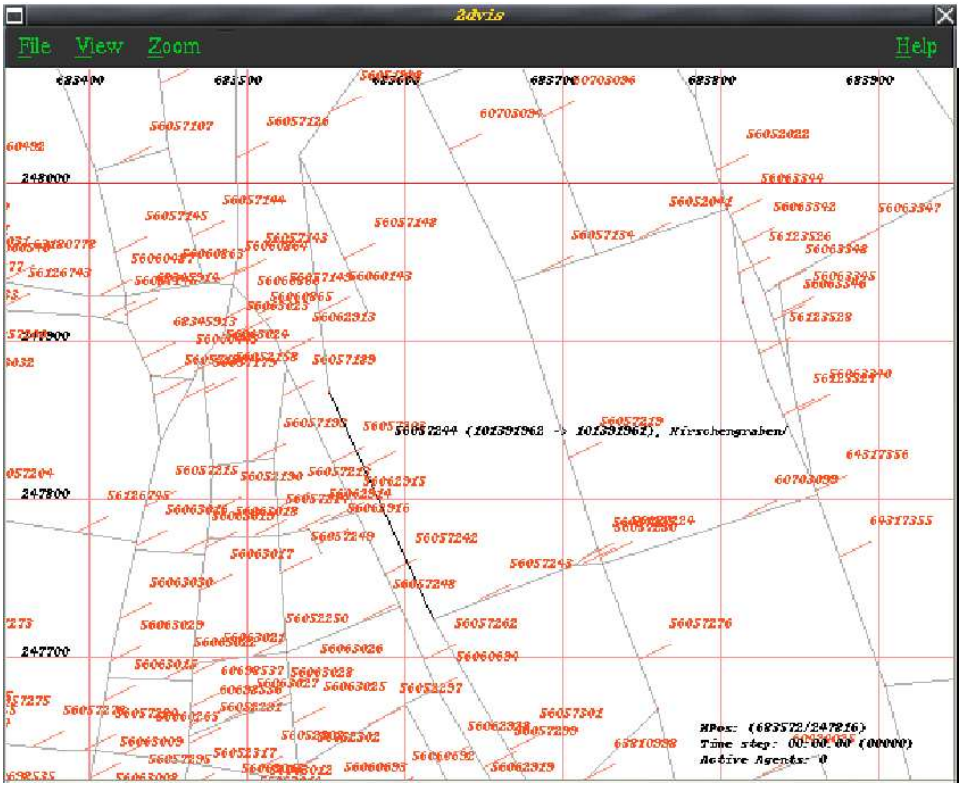


Figure 6.2: The 2-dimensional visualizer shows lots of details if configured to do so, like link ids of the network, and details like street names, if given in the network file. This is an example of the street network of Zürich.

Static Information: The Network

The 2-dimensional visualizer displays static information, that comes directly from the network file, which is the same for every module. It shows information related to the street network, i.e. link and node IDs, and street names (see Figure 6.1 and 6.2).

Also the visualizer shows additional details of the scenario like trees (green), if available.

- Link and nodes of the network. Nodes are just points with no further information (except the ID). Links, however, are almost never straight lines between two nodes, but are represented by a more or less twisted path. Since the agents are capable to follow this path in the mobility simulation, the path has to be represented exactly the same way here.
- Link and node IDs, if requested in the menu, are shown as little numbers with lines pointing to the actual link or node.
- Link details, like street names (if given in the network file), are visible only if the mouse pointer is moved over an already displayed link ID. This link is highlighted and its label is extended by

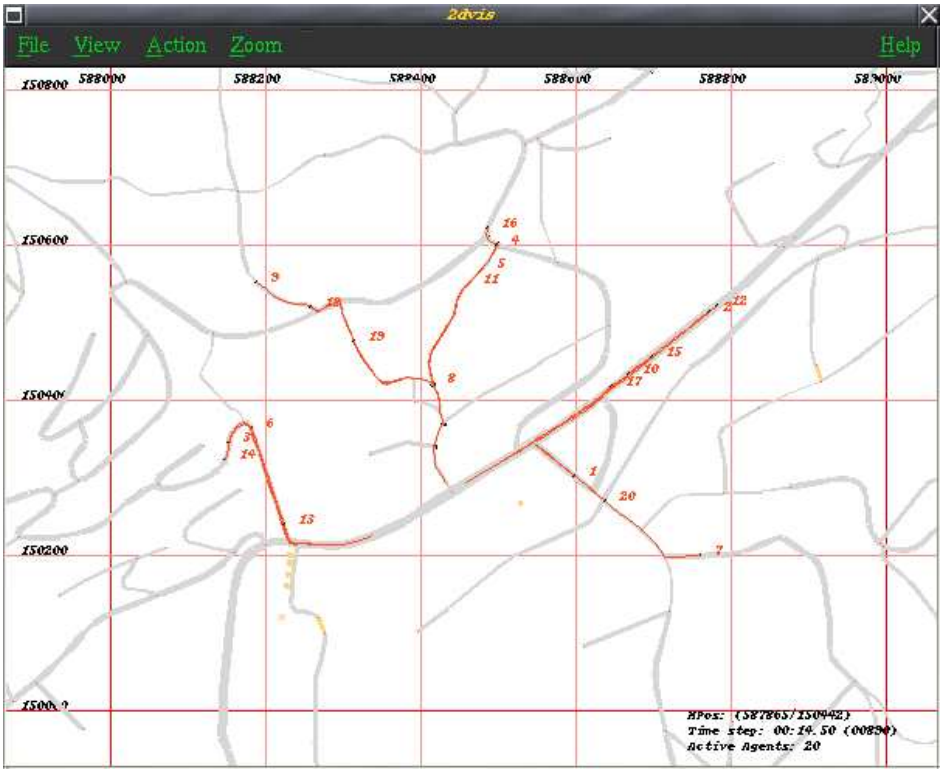


Figure 6.3: The 2-dimensional visualizer is capable to show agent related data, for example the agent's IDs, or to plot a trail of the last known positions of an agent.

further information, i.e. the street name from the network file, or any other string that is given there.

- The link type (e.g. "Hiking Path", "Street", "Highway", "Bridge") is represented by different widths or colors. The thicker a line is drawn, the higher the link's type is. Bridges are green.
- There are special objects that are used to test a scenario. As an example, the visualizer is able to display trees (green). These come from a special section in the network file that is proprietary to our project (See B)

Agent Positions

The main reason thing a visualizer has to display is information that changes during the run of a simulation. This is mainly the positions of the agents. These agents are represented as small dots (using the real size of an human, more or less, see Figure 6.3). The information comes through the events channel.

- Agents (position events) are shown as dots. The size of the dots is about 50cm, which scales down to almost a dot if the view is not zoomed in too much. The position comes from the `position events` sent by the mobility simulation.

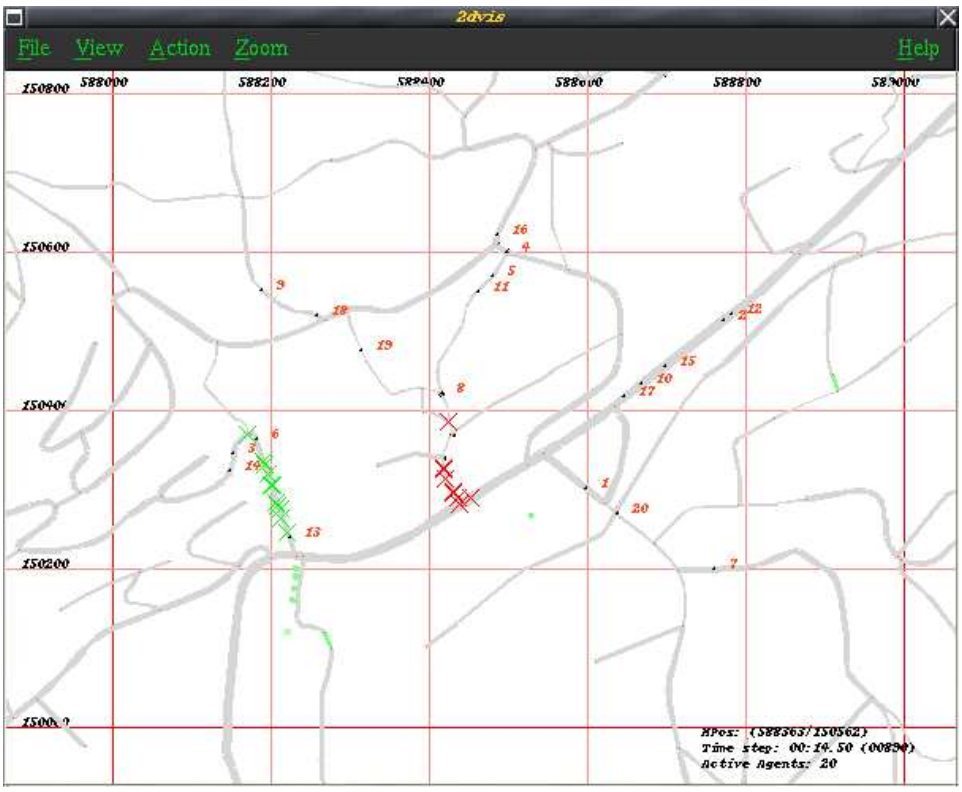


Figure 6.4: The 2-dimensional visualizer is capable to show the events sent to the event streams. Here, small green crosses are for `forest` *events, red crosses for* `pressure` *events. Note that there is no a priori valuation in these events, this is up to the modules in the mental layer.*

- If requested using the menu, agent IDs are shown.
- It is also possible to show the last 20 positions of every agent, connected by a line. This draws agent trails. However, the length of this trail depends of the number of events received from the mobility simulation and depends therefore on the number of `position events` that are sent per second or timestep.
- It is possible to send an `agenttype` tag in the `position events`. If present, it is possible to assign a different color to each type. This can be used to assign an agent type for a special demographic type, like 1 for agents that like forests and 2 for those that do not. Using this colored representation, it is easier to check if the outcome of a simulation run is feasible or not. In `pedsim`, the open source demo mobility simulation, this tag is used to color the two groups in different colors: yellow fights green (see Figure 6.12).

Events

The 2-dimensional visualizer is capable to show the more dynamical information than just the agents' positions. It listens to all events sent to the event streams. It displays all events sent by the mobility

simulation and any other module, as long as there is an entry in the visualizer's configuration file for that event type. However, these events need a X and Y co-ordinate, otherwise the visualizer is not capable to find the location where it has to draw the event. It would be possible to display events related to links or nodes as well, but this feature is not present in the current version of the visualizer.

A typical event looks like this:

```
<event type="forest" agent="42" x="588000" y="144000" />
```

which means that agent 42 has seen (or is in) a forest at this position. There is no a priori valuation in this event, so the visualizer does not know if this is something good for the agent. The demographical data for an agent comes from the `agent comm` channel, which is point-to-point based from the brain to the mobility simulation, and the visualizer has no access to.

In Figure 6.4 small green crosses are for `forest` events as seen above, red crosses for `pressure` events.

The following list of events is not concluding, since the visualizer is capable to display any event that carries a X and Y co-ordinate. It shows the events used most in the example scenarios:

- Spatial events, e.g. `view` and `forest`. These events come more or less directly from a GIS file read by the mobility simulation. If an agent is at a position that is covered by e.g. forest, the corresponding event is sent.
- Direct events, e.g. `too crowded`, which are a result from pedestrian interaction in the mobility simulation and are not stored in a GIS file.
- Indirect events, e.g. `rain` or `sun`, or the derivatives `hitbyraindrop` events, which are generated by external sources or the mobility simulation.

Special Information

Some information does not fit into the categories presented above:

- Mental map data, which can be used to see what mental map is stored in an agent's brain. This is represented of a sub-graph of the street network, which is colored blue in the displayed street map of the visualizer. This feature was implemented for router debugging.
- Block events, which show how the memory allocation mechanism in the pedestrian simulation works (See Chapter 3). If a memory block is generated for the first time, this block is drawn a a solid red square. If it is loaded from the disk cache, the outline of the square is drawn in the same red color.
- Route answer events, which come directly from the router as a response to a route request. This information is meant for the brain which initiated the request, but the visualizer is able to receive these events as well, ant uses them to display the planned route for a agent. These route answers contain just the chain of nodes, but since the visualizer has no complete graph representation

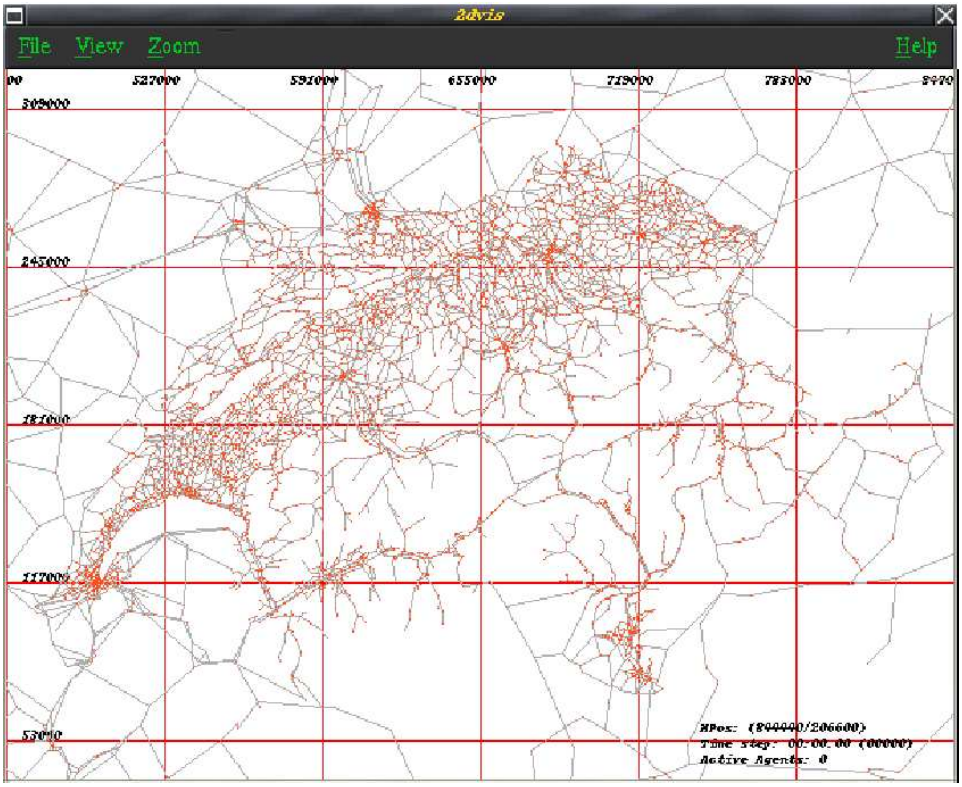


Figure 6.5: The 2-dimensional visualizer is able to display networks used for traffic simulations as well. Here the street network of switzerland is shown. However, the network is a bottleneck for simulations with a large number of agents.

of the street network internally, it is not capable to figure out which link is between two node. Therefore, the route answer is displayed as a chain of highlighted nodes.

6.2.2 Input

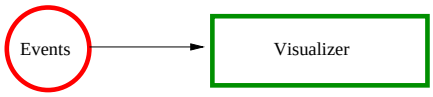
The 2D visualizers main purpose is to show what happens in the simulation. The visualizer is like a camera, and should not interfere with the system.

However, a way to interact with the system is sometimes needed, e.g. for debugging purposes.

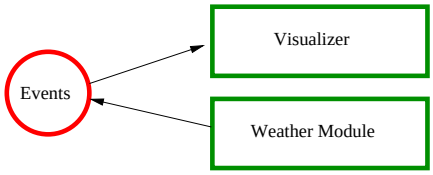
In order to maintain a sound design of the framework, input should be achieved through independent modules, that send their contribution as events or commands to the rest of the simulation system.

However, often it is faster (in terms of programming) to add such input code to an existing module. And also, if the input is meant as an user reaction to something that happens in the system, it makes perfect sense to let the user do this input at the place where he sees the output.

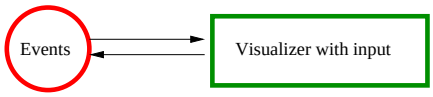
To test the ability of the system to react to certain events, we decided to implement `rain` events which show where and when it rains:



(a) Visualizer used only for output



(b) Additional Weather Module



(c) Output and input includes in visualizer

Figure 6.6: A visualizer does display the events received from the events stream (a). In order to maintain a sound design of the framework, input should be achieved through independent modules (b). However, often it makes sense to add such input code to an existing module (c).

```
<event type="rain" id="0" time="0" x="587757.000000" y="151056.000000" />
<event type="rain" id="0" time="0" x="587837.000000" y="151072.000000" />
<event type="rain" id="0" time="0" x="587941.000000" y="151072.000000" />
<event type="rain" id="0" time="0" x="587477.000000" y="150864.000000" />
<event type="rain" id="0" time="0" x="587453.000000" y="150856.000000" />
```

By pressing the right mouse button inside the visualizer's window, rain drops can be "painted" into the scenario.

Every module could send such events, it does not matter where they are from. In a fully developed simulation system, a special *weather module* could handle this.

For our purpose, a direct way to input those events into the system and a way to see how the agents would react was needed. The code to display a 2-dimensional representation of the simulated area and react to user input was already there: the 2D visualizer.

This output device therefore now has the ability to send `rain` events, and lets the user interact directly with the system. `rain` events can be drawn with the mouse (like in a graphics program, see Figure 6.7). The visualizer the sends events via the event channel, and all modules can catch these and do an interpretation, if needed. Also any visualizer is able to receive these events, in order to display them. If on one visualizer rain events are entered, the appear instantly on every other visualizer as well.

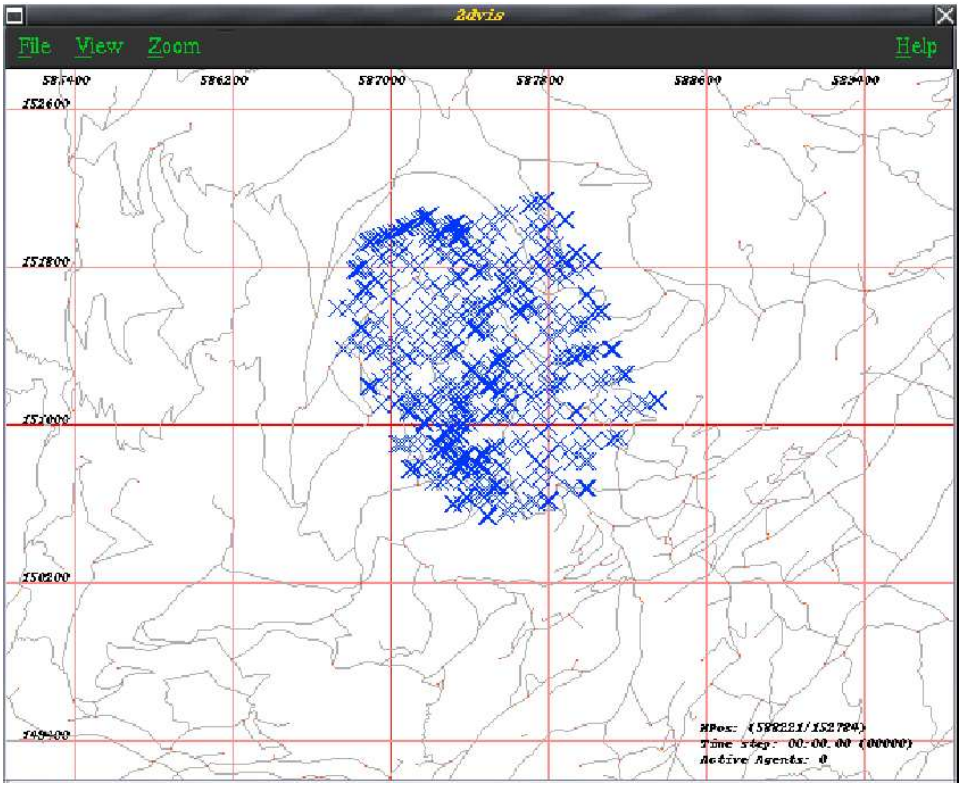


Figure 6.7: Since the code to display a 2-dimensional representation of the simulated area and react to user input was already here, this 2-dimensional visualizer now has the ability to send `rain` *events, and lets the user interact directly with the system.* `rain` *events (blue) can be drawed with the mouse (like in a graphics programm).*

The visualizers are used to interact with the simulation system. However, they are not bound any harder to the simulation as before. That a module sends events is no guarantee how the rest of the system reacts to them. Events are just an stimuli, how to treat them is up to the individual module.

As a side note here: it is possible to keep the separation of input and output modules. A visualizer can be configured to not to show events (e.g. `posiotion` events), and act as an 2-dimensional input module only.

6.3 The 3-dimensional Visualizer

One of the main aspects of the project was to build a multi-agent simulation, whose agents are capable to react to visual changes in the landscape.

There were two fields of application for the 3-dimensional visualizer:

1. To look into the simulation system from a bird's-eye view. This is useful to check the overall quality of the modeled landscape or if the agents behave reasonable.

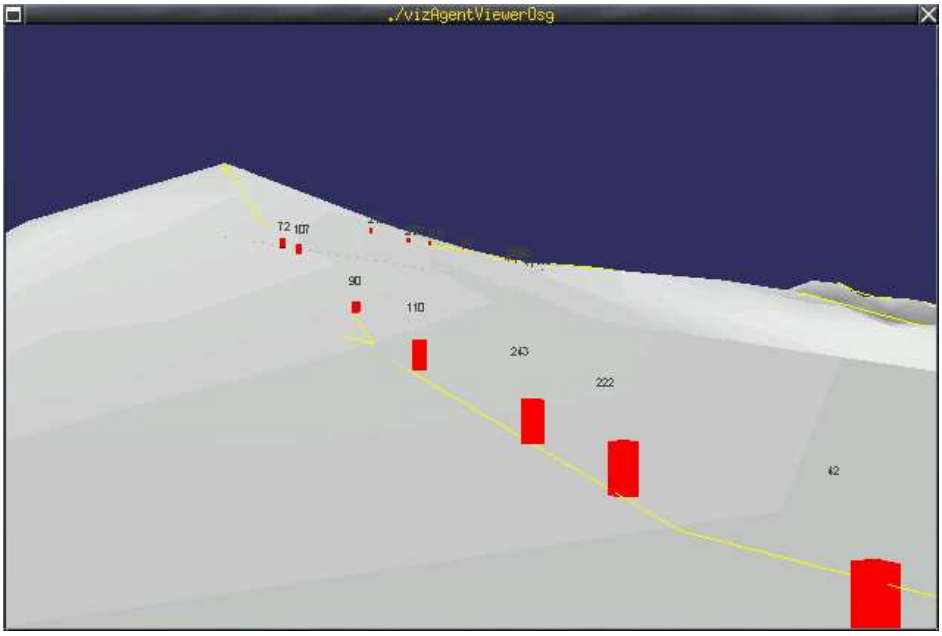


Figure 6.8: The 3-dimensional visualizer by Duncan Cavens. In this picture, a early version can be seen. Agents walk on a trail to the Hugeligrat *(Rellerli).*

2. To look through the eyes of an agent, in order to see what the agent field of view is at the moment.

However, it turned out that the 3-dimensional visualizer can be used as a module to determine what the agent see automatically (see Chapter 10), without a human interaction. The 3-dimensional visualizer was developed by Duncan Cavens (in preparation).

It is possible to attach as many 3-dimensional visualizer as needed to an event stream. Even 3D and 2D visualizers can be mixed at the same time.

6.4 The Offline Renderers

The 2-dimensional and 3-dimensional visualizers mentioned above are able to connect to any event stream that supplies `position` events. This can be e.g. a traffic simulation, where the agents represent cars, or the massive crowd simulation *pedsim* (see Chapter 3 and in this chapter, Section 6.12).

Since these visualizer provide a real-time view into the simulation, the actual drawing of the scene must not take longer than a fraction of a second. Otherwise, delay or flickering is noticeable. If superior graphic quality is needed, e.g. for presentations or a short movie, it is necessary to abandon the feature of real-time rendering.

A recorded stream of events can be feed into *off-line* renderers. These take as much time as they need to render a snapshot of the scene, called a *frame*. Since there is no time limitation anymore, advanced graphical concepts as ray-tracing can be used.

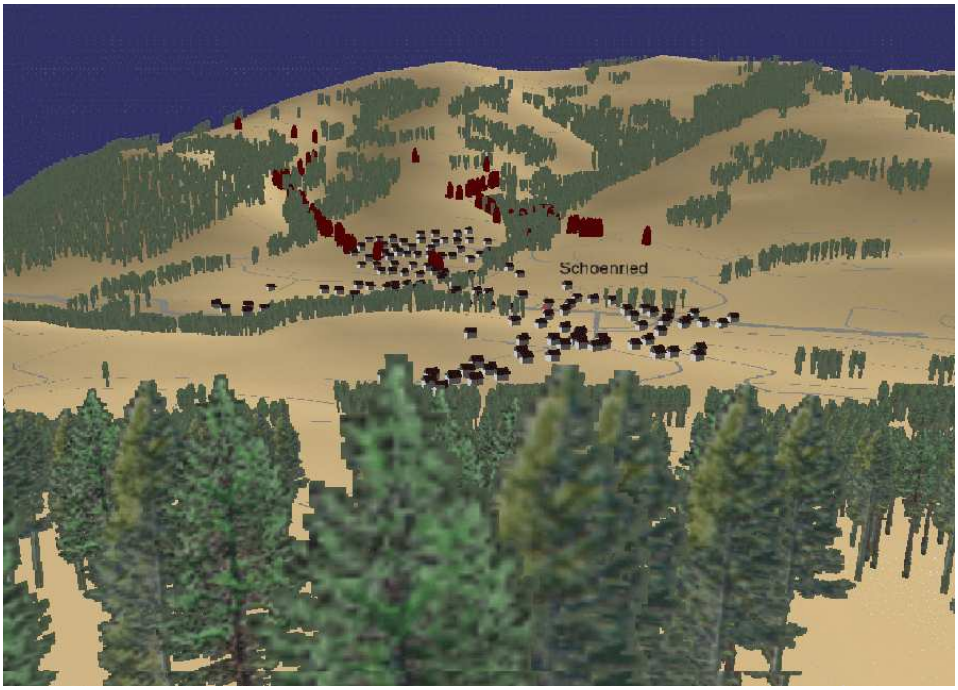


Figure 6.9: The 3-dimensional visualizer by Duncan Cavens. A view of the village of Schönried*, with* Rellerli *and* Hugeligrat *in the background. This is exactly the same scenario as presented in Section 9.2.3. The agents are rendered at ten times their natural size for better visibility.*

This calculates an image of a scene by simulating the way rays of light travel in the real world. Ray-tracing programs start at the location of the observer and trace rays backwards out into the scene. For every pixel in the final image one or more viewing rays are shot from the camera, into the scene to see if it intersects with any of the objects in the scene.

Every time an object is hit, the color of the surface at that point is calculated. Special features like atmospheric effects make it necessary to shoot a lot of additional rays into the scene for every pixel.

The off-line renderer used to generate the picture in Figure 6.13 needs up to 5 minutes per frame, depending on how many pedestrians are visible, and how close to the scene the virtual camera lens is.

A offline-renderer completes these tasks:

1. It takes a stream of events and converts these into snapshot-like scene descriptions readable by a renderer. This event stream can be recorded or even a real-time stream directly from a simulation. This step does not consume a lot of time, since it consists of re-formatting text only. However, in each timestep, the *whole scenario* needs to be written again, since these description files are processed independently afterward.

2. These scene description are then fed into the actual renderer, which turns them into picture files. Each timestep of the scenario description yields in one high-resolution image.

3. In order to generate a movie sequence out of these images, they need to be combined. Since the

Figure 6.10: The offline renderers allow to generate a photo-realistic image of the scenario. However, to produce this image of high quality, a decent computer has to calculate the traces of light rays for about 10 minutes. Here, the view an agent has when approaching a wood glade is shown. Since each tree consists of more than 20000 polygons, even a close up view looks realistic.

sum of all the images yields in a huge amount of disk space needed, the final sequence should be compressed.

6.4.1 PovRAY

The Persistence of Vision Raytracer (Povray www page, accessed 2005) is a open-source high-quality renderer. The input is given by a script which describes the scene. Since this script can be generated automatically, PovRAY is perfectly suited for our purpose.

A Povray example scene:

```
#include "objects.inc"

background { color rgb <0.0, 0.2, 0.5> }
camera { location <10, 10, 15> look_at  <24,  4, 10> }
light_source { <-110, 100, -300> color White}
plane { <0, 1, 0>, 0 pigment { color rgb <0,0.2,0> } }

object { agent rotate <0,42,0> translate <15,0.8,12> }
```

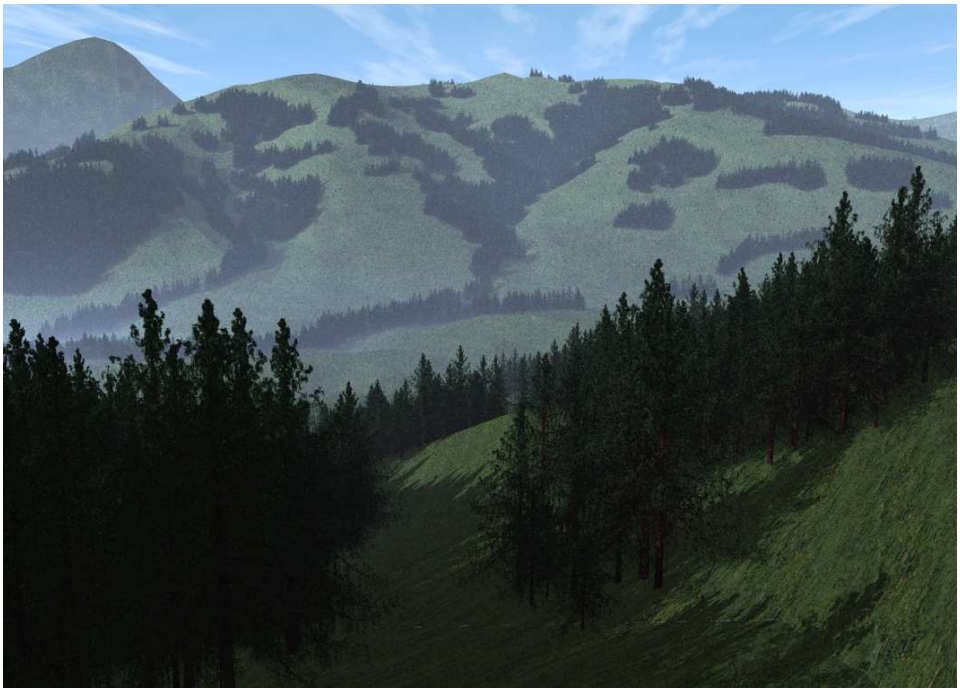


Figure 6.11: The same view as shown in Figure 6.9 and Section 9.2.3 created by using the offline renderer and PovRAY: a view toward Rellerli *and* Hugeligrat. *No agents are visible from this distance. The village of* Schönried, *which sits on the bottom of this valley, has not been modeled for this visualizer yet.*

```
object { agent rotate <0,31,0> translate <43,0.8,15> }
object { tree rotate <0,133,0> translate <42,0.7,13> }
[...]
```

For the efficient representation of trees, PovRay offers a excellent mechanismus: it is able to store a single copy of such an object in memory and use the same copy again. A basic version of a tree can be constructed using a tool like Arbaro (Arbaro www page, accessed 2005), which allows to save the result as a PovRAY mesh.

The position of each tree in the Berneroberland is not known. However, the GIS tools are capable to provide the polygons which describe the borders of the forests. Using these polygons and a simple Perl script, the scenario is populated by trees that are inside these polygons, but at a random position. For the pictures shown in Figures 6.10 and 6.11, about 100000 trees are used. There is almost no difference regarding rendering time and memory consumption for 1 or 100000 trees. Depending on the size of the picture and the quality settings (such as anti-aliasing, jitter, and athmospheric effects), it takes a decent computer (i.e. a 64bit AMD Opteron, 1.8 GHz) about 1 to 30 minutes to render. Memory consumption for the whole scenario is about 450 Megabytes.

In an second step, the scene description of each frame is rendered individually, since they are now independent. This make the task of distributing the rendering onto different machines simple: each computing node takes care of one frame, as soon as it has finished, it fetches another description and

starts with the next frame until all frames are rendered. The quality of a frame in such a movie can be lower than for a single picture, since the encoder also blurs the input because of compression and the motion. It takes about 5 minutes to render typical frame of a movie.

In the end, those separated images are combined into a movie sequence using an external program. Each minute of a movie requires 1440 frames (24 Hz), or about 120 hours of computing time.

6.4.2 Renderman

Povray is used often to generate a single picture out of a complex scene description. The Renderman (Renderman www page, accessed 2005) specification, since developed by a animation studio company, offers more features in the direction of real-time rendering. For example, it is possible to encapsulate scene descriptions of multiple frames in a single file. There exist multiple implementations of the Renderman Specification.

A Renderman compliant rendering program is able to read such a scene description directly from the network, exaclty as the 2- and 3-dimensional visualizers presented above do. All that is needed is a conversion utility that takes a stream of events and outputs a Renderman compatible scene stream. However, the output of the Renderman program will still consist of single images that need to be combined later.

At the moment, a prototype of such a conversion utility exists. Since most Renderman compliant renderers use a faster algorithm than ray-tracing (scan-line rendering), much less computing time is required to generate a frame. However, using this algorithm, global illumination calculations are not as realistic as by using ray-tracing.

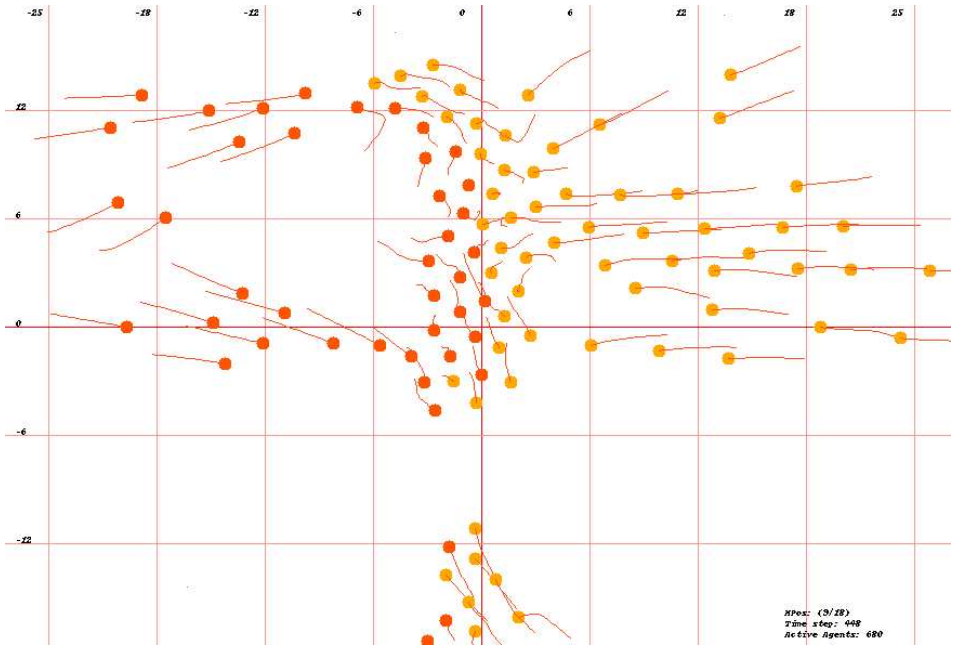


Figure 6.12: The 2-dimensional visualizer, connected to the crowd simulation "pedsim". The trails of each pedestrian shows how it get diverted by other pedestrians. This is part of the scene rendered for Figure 6.13.

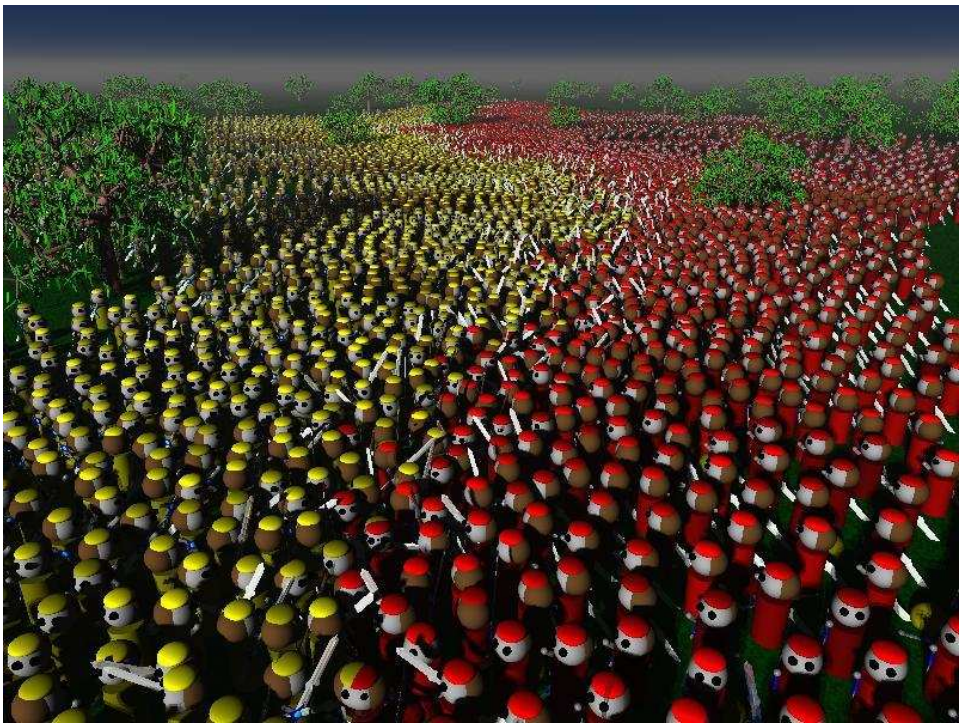


Figure 6.13: A frame of a "pedsim" scene rendered by the off-line renderer using Povray. To render this frame, it took a single xibalba node (1GHz PIII cpu) for about 3 minutes. This is the same scene as displayed in the 2-dimensional visualizer in Figure 6.12.

Chapter 7

Communication

7.1 Introduction

Since the communication needs between these modules is substantial, several methods for message passing are evaluated. Traditional approaches, such as MPI or PVM, are too inflexible for what was intended for this project. For that reason, specialized protocols based directly on the operating system are evaluated. These protocols have trade-offs in terms of ease-off-use, bandwidth consumption, and potential message loss. The overall result is that, albeit at the expense of having to use a variety of protocols for different purposes, even with existing technology simulations with thousands of agents running hundreds of times faster than real time are possible with our approach.

Traditional implementations of transportation planning software, even when microscopic, are monolithic software packages (Rickert, 1998; Babin et al., 1982; PTV www page, accessed 2004; Salvini and Miller, 2003). By this we do not dispute that these packages may use advanced modular software engineering techniques; we are rather referring to the user view, which is that one has to start one executable on one CPU and then all functionality is available from there. This is discussed briefly in Section 7.2.1.

The traditional way to couple the modules of a simulation system is to use files (Section 7.3) or to use a database management system to store the agents state (E.g. Waddell et al., 2003), see Section 7.4.

Our approach which is presented in this chapter is to couple the modules by messages over raw TCP/IP sockets (Sections 7.6, 7.7 and 7.8). In this way, each module can run on a different computer using different CPU and memory resources, which overcomes the memory bottleneck of the monolithic approach. The approach also avoids the bottleneck of file I/O, since data is not written to file at all while the simulation is running.

It is even possible (as is with file-based interfaces) to make the modules themselves distributed; for example, we have mobility simulations (for traffic) which run on parallel Myrinet-equipped clusters of workstations (7.5).

Some of these network protocols do not guarantee that the message arrives at the receiver. In Section 7.9 a mechanism is presented that is both flexible and reliable.

On simulations with many of agents, issues such as bandwidth usage, packet loss, and latency become increasingly important. As a result, we use different network protocols and implementations tailored to specific requirements of inter-module communication. Section 7.10 will discuss some of these protocols, and the diverse purposes they serve in a distributed multi-agent simulation.

7.2 Subroutine Calls

7.2.1 Locally Linked

The first step into the direction of modularization is to write the modules independently, but to define a proper interface between them. This can be a simple description of how the functions of a module are called by the main program. The modules are then compiled independently but are linked into one program that runs on a single CPU as one task. There come limitations but also advantages with this approach:

- Modules can be developed by different persons. However, they have to agree on a single computer platform (e.g. Linux).
- It is not possible to distribute the program among multiple computers or CPUs. This also means that it is straightforward to run the executable, the setup is simple.
- Function calls are very fast, compared to the other mechanisms later. If one huge fast computer is available, this is the way to go.

The fact that such a program can not be distributed among multiple machines may be acceptable as long as the problem size is small. The limiting factor is the number of agents. As a estimation, it is possible to run up to 5000 agents on a single CPU without problems. But since the goal is to simulate 10000 and more agents, the single-CPU approach is a dead end.

7.2.2 Remote Procedure Call (RPC) and Common Object Request Broker Architecture (CORBA)

Remote Procedure Call (RPC) is a protocol that a program can use to call a function of a program located in another computer in a network. RPC uses the client/server model, where the requesting program is a client and the service-providing program is the server. RPC makes it easier to develop an application that includes multiple programs distributed over a network.

By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism.

RPC behaves like a usual function call. Using RPC, the calling arguments are passed to the remote procedure and the caller waits for a response from the remote procedure. The client makes a procedure call using a special RPC library that sends a request to the server and waits. The client thread is blocked from processing until a reply is received. When the request arrives, the server calls a routine

that performs the requested function, and sends the reply back to the client. After the RPC call is completed, the client program continues.

Using RPC is simple and usually a good method to distribute existing programs. However, one looses the ability to use different programming languages for the different modules. There exist PRC solutions that work in heterogeneous environments (e.g. XML-PRC (UserLand Software, Inc. www page, accessed 2005) which uses HTTP as transport protocol).

There exist also object-oriented approaches to RPC. One of the most successful specification is the Common Object Request Broker Architecture (CORBA) (www.corba.org, accessed 2005). Instead of executing a function call on a remote machine, a object (i.e. the data stored in the object. The definition of the object has to be known to the remote machine in advance) is sent to the remote machine.

However, both RPC and CORBA were developed with commercial applications in mind. There is some overhead used for security (i.e. authentification).

RPC and CORBA are good solutions if there is a request with a responding answer. An example are the *route request* and *route answer*: the agent database is the client, the route generator the sever. However, as soon as there is one-way communication (broadcasting the events), this model seem not to fit well. If the requesting program (i.e. the client) is supposed to do something else in the meantime, a multi-threaded solution is required. The program design becomes more complex quickly.

An alternative method for client/server communication that is similar to RPC and CORBA is *message passing* (see Section 7.5).

7.3 Files

A further step to overcome these problems is to make all modules completely standalone, and to couple them via files. Such an approach is for example used by TRANSIMS (TRANSIMS www page, accessed 2005). The two disadvantages of that approach are:

1. The computational performance is severely limited by the file I/O performance.
2. Modules typically need to be run sequentially. Each module needs to be run until completion before starting the next module. For example, the routing module can only be run before or after the mobility simulation. This implies that agents cannot change their routes while the mobility simulation is running.

However, since reading and writing files is something that is possible on every computer platform and using every programming language, this solution is available everywhere.

Probably the biggest advantage is that these files can be achieved and processed later. A transient step is available for examination later, which helps during debugging.

In research projects, disadvantage 1) might be less a problem, as long as the scenario size is relatively small. However, in the framework we had in mind in this project, disadvantage 2) rules out intra-day replanning, which is something we wanted to support.

7.4 Databases

Another possibility is to use a database system for information exchange between modules. This is easiest to imagine if modules still run sequentially. Then each module changes the state of agents in the database, and some central schedule decides which module to run at which time. This approach is, for example, used by URBANSIM (Waddell et al., 2003).

This approach is also possible for modules that run in parallel. The state of each agent is stored in the database, and each module sends notifications to the other modules as soon as something has changed in the agent's state. This would be a good way if the agent state consists of a huge amount of data, since only the notification is sent over the network. The other modules then reads in the database only what is really required. A centralized database can also be a bottle-neck, if all processes try to access data at the same time.

However, in our scenario, the state of an agent is relatively small, and it makes sense to transmit the complete state together with a notification.

Similar to the solution using a database is a *blackboard system* (e.g. DARBS, see Nolle et al., 2001). The blackboard has the appearance of a database. The processes can only communicate through the blackboard. DARBS uses client/server technology. The blackboard acts as a server and the other processes act as clients.

In DARBS, a single blackboard and a number of processes co-operatively solve a problem. The processes observe the blackboard constantly and activate themselves when the information interests them. They all run in parallel. Whenever the content of a partition of the blackboard is changed, the blackboard server will broadcast a message to the processes.

This blackboard infrastructure looks similar to what we use as an agent database. However, the blackboards system is only a communication system: it is passive, like the events stream in this work. Our agent database can interact actively if necessary.

7.5 Message Passing: MPI/PVM

When a *single module is distributed* across multiple computational nodes, one often uses MPI (Message Passing Interface (MPI www page, accessed 2005)) or PVM (Parallel Virtual Machine (PVM www page, accessed 2004)). For example, our traffic simulation module (Cetin, 2005) is distributed to 64 hosts or more.

Message Passing is comparable to RPC and CORBA. However here, messages are sent to another host. These can be of arbitrary content and do not have to be arguments to function calls. The communication is symmetric, there is no dedicated server or client. Of course it is possible to use RCP in this way as well (define dummy functions that just collect data), but then there is no reason to use RPC at all.

PVM seems to have lost the race against MPI, and there is no support for PVM using the latest network technologies (like Myrinet (Myricom www page, accessed 2005) or InfiniBand (InfiniBand Trade Association www page, accessed 2005)). We decided to not follow this alternative.

MPI is the *de-facto* standard for parallel computing both in industry and academics. It was designed

for high performance on parallel machines and on workstation clusters, and most of the network infrastructure is supported natively.

However, the moment this project started (February 2002), there were limitations to the available implementation of MPI ((MPICH www page, accessed 2005) implementing MPI standard 1.1):

1. Sending a message would block the execution of the sender task until it receives an acknowledgment from the receiver.
2. Each task that participates in this communication has to be the same binary executable.
3. Therefore, it was also impossible to use MPI in a heterogeneous environment.
4. MPI communication is reliable and does not support advanced techniques as multicasting.

With the latest version of MPI (2.0), at least items 1) - 3) are possible. It is therefore thinkable to couple a C++ mobility simulation to a agent database written in JAVA (see (Cetin, 2005)). It becomes now possible to use MPI for the communication between *different modules* as described in this paper. That approach has, however, the disadvantage that one is bound to the options that MPI offers. For example, options to add or remove modules during runtime have only recently been added to the MPI standard, and multicasting is not possible at all.

7.6 Transmission Control Protocol (TCP)

The TCP/IP suite of protocols was designed to tolerate a unreliable network (e.g. the Internet). TCP/IP has two modes of communication:

- **TCP** (*Transmission Control Protocol*) is connection-oriented, bi-directional and reliable
- **UDP** (*User Datagram Protocol*) is connectionless (packet-oriented), uni-directional and unreliable

TCP breaks a message into chunks and makes sure that they reach the destination without errors and in the correct order. Since TCP is connection-based, a virtual connection has to be set up between one program and the other before they can communicate. Through this connection, both sides can send their messages as the connection is symmetric.

On clusters of workstations, MPI is often implemented on top of TCP.

The connection bases approach of TCP brings also problems:

- If two modules want to communicate, the channel between these modules has to be opened. This introduces more complexity to the modules.
- It is not possible to open a channel to multiple receivers.

- If one end closes its site of the connection, the other end has to recognize this and act accordingly.
- There has to be a “server”, which listens for incoming connections, and a “client”, which tries to establish a connection. This differentiation is also reflected in the order one has to start the modules.
- It is not possible to build a circle in the communication path, unless a modules is both server and client.
- If the receiver is not able to read the incoming data buffer fast enough (e.g. a slow visualizer), also the sender has to wait, since no messages are allowed to get lost.

TCP offers reliable communication, but also introduces these problems. Without a sophisticated method to take care of these problems, the complexity of the communication part in the modules would require too much work. A interesting alternative to TCP is UDP (presented in the next section) or the TCP Multicaster, an add-on presented in Section 7.9.

7.7 User Datagram Protocol (UDP)

UDP offers, in comparison to TCP, no control for packet loss. This means that there is no guarantee that the sent packets will arrive. The advantage is that there is considerably less overhead. Also, as will be described later, this offers (via multicast) the option to send events only once, even when many modules listen to them. In a TCP-based implementation, events need to be sent to each listener separately.

A message that arrives is guaranteed to be error free, since the UDP protocol includes a checksum.

We use UDP to transmit the agent positions to the visualizers. If the network is down for a few seconds, the simulation does not need to slow down because of lost packets. Once the viewer is back on line, it will receive the latest positions.

There are other situations in which one may accept the loss of messages. For example, if an agent reports that it is blocked in unexpected congestion (e.g. waiting for a cable-car, traffic jam), it needs a new route instantly. If its request is lost or delayed, it makes no sense for the system to buffer its request, since the agent has moved on, and the location in the original request might now be invalid. A new route computed based on the old information will be invalid as well. It is the agent’s responsibility to restate its position again if it does not receive a new route after a certain time has elapsed (Gloor, 2001).

The amount of packet loss is strongly dependent on the overall number of packets in the network. In state-of-the-art networks, which today are often 1 Gbit Ethernet, there is hardly any packet loss in the network itself. Losses occur mainly in the sending and receiving network interface cards (NICs), due to overflowing buffers. This is the case, for example, if the CPU is busy so that it cannot read the packets from the buffer quickly enough. The more packets that are sent, the higher the chance that one is lost.

With Gbit Ethernet communication, up to 180’000 raw data packets can be sent per second without any losses. Using a naive approach, which is packing one event into one network packet, one obtains 180’000 events per wall-clock second. Since the mobility simulation runs more than 100 times faster

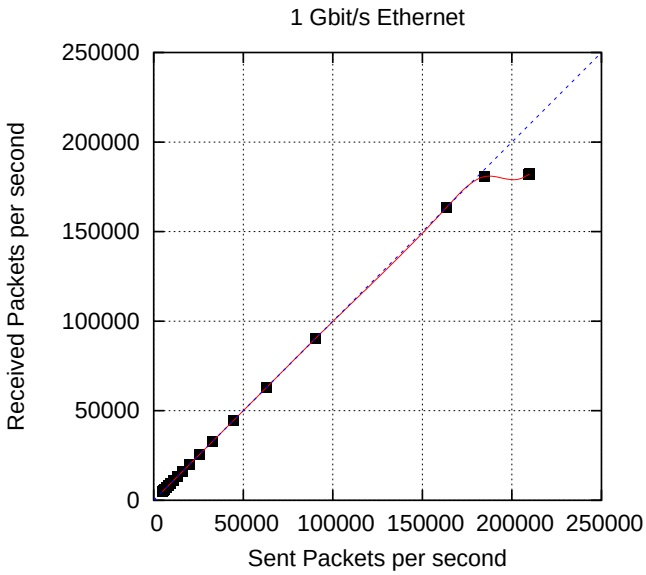


Figure 7.1: UDP packets sent at different rates using an 1 Gbit/s network, plotted as sent vs. received messages. Some packets are lost in the network, but most of them are lost in the overflowing buffers of the NICs. Each UDP packet can carry one event from the mobility simulation, e.g. a new position of an agent.

than wall-clock time, this results in 1’800 events per simulated second. This is not very much for a simulation of 1000 or more agents that report their perceptions.

Recently, new networking infrastructures have been developed which allow to send packets reliably with UDP. An example is the TNet Hardware (SCS www page, accessed 2005), used in computer clusters. The TNet hardware assigns a 16 bit CRC (cyclic redundancy check) to each packet. This checksum is used for error detection. After every transmission over a network link the packet is checked for correctness, and retransmitted if an error is detected. The 16 bit CRC is generated in the NIC and not changed in the network, therefore bit errors in the switches themselves can be also detected. This link-level protocol guarantees that no packets are lost in the network.

7.8 Multicasting

Often there is a need for sending the same packet to more than one receiver. This can be achieved by opening multiple TCP connections or by sending multiple UDP packets to the receivers. However, on large simulations, the network interface card (NIC) of the sending host quickly becomes the bottleneck, as it is unable to send out enough packets to keep the receivers fully occupied.

On Internetworks, it is possible to use *multicasting* to send a single message from one computer to several other computers, instead of having to send that message once for every destination. Because multiple machines can receive the same packet, bandwidth is conserved. Multicasting is particularly useful for any kind of streaming data such as radio or television broadcasts over the network. Its

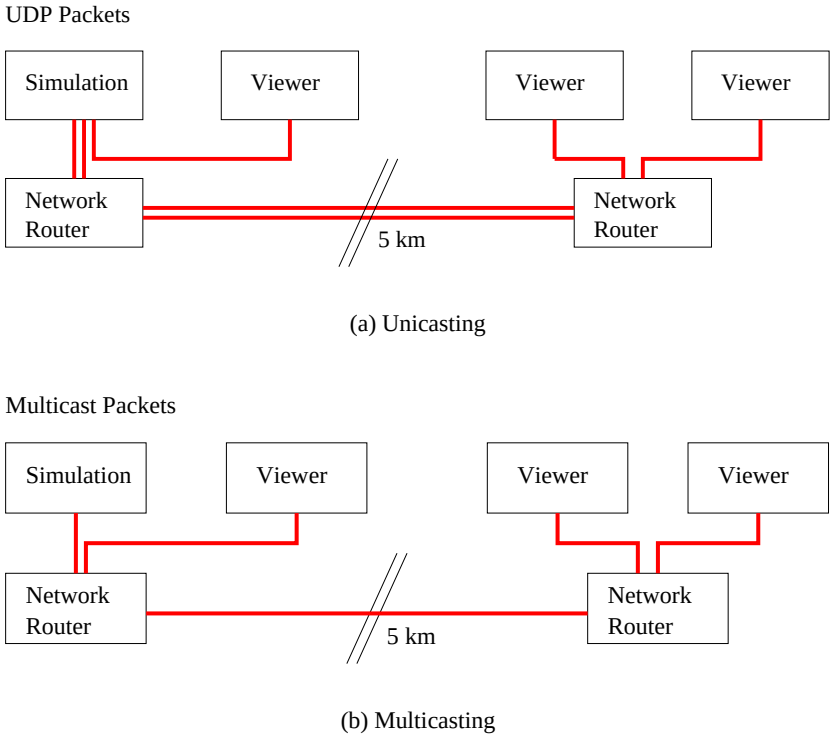


Figure 7.2: Multicast uses one data connection, even if there are multiple receivers subscribed to the multicast channel.

advantage is that the multiplication of the packets for multiple receivers is not done by the NIC, but by the network itself. This allows to avoid the NIC bottleneck.

Multicasting provides groups of hosts, that are referenced using special IP addresses (224.0.0.0 – 239.255.255.255). The sender chooses one of these groups and sends a single packet to this IP address. A receiver must explicitly join a group first, telling its NIC and the operating system to listen for packets sent to this group.

An advantage of this addressing scheme is that the sender does not need to know the IP address of the receiver. This simplifies the configuration of the system substantially. The Internet routers ensure that the packets find their way from the sender to the receivers, once they are registered to the multicast group.

A drawback with multicasting is that it has, similar to UDP, no arrival control. There is no feedback to the sender if all packets arrived at all destinations, or even at any destination at all. In consequence, this is not useful when message arrival needs to be guaranteed. To live with this problem, it often is possible to implement some sort of flow control into the application (see, e.g. Floyd et al., 1997). This, however, is a hard task and does introduce performance issues under certain circumstances. But often there is no need for the full flow control available in TCP, and a lightweight solution can increase the performance substantially. This task is simplified by the fact that in most local networks packet loss is almost zero

if you do not saturate the network. An implementation using flow control is beyond the scope of this paper.

In order to visualize simulation runs, we have developed *viewers*, which are stand-alone applications that connect to the simulation system via the network. Our project is a collaboration between two institutes at ETH Zürich. One of them is located more than 5 km away from where our computational cluster is. For every viewer that is connected to the simulation, extra bandwidth is needed. Using multicast, we cannot reduce the bandwidth used for one viewer. But as soon as there are multiple viewers looking at the same general area, the bandwidth remains constant (Fig. 7.2).

Sending agent data to multiple viewers is an instance where multicasting is extremely effective. Using multicasting, we were able to allow multiple viewers at the remote institute without saturating the network.

A typical *position update message*, sent every timestep by the simulation for each agent to announce its new position to the other modules, has a size of 100 bytes per agent. Since our mobility simulation is able to simulate more than 100 real-time seconds per second, 100×100 bytes/s need to be sent per agent. A simulation of 1000 agents therefore needs $100 \times 100 \times 1000 = 10'000'000$ bytes/s, which is already close to the theoretical limit of a 100 Mbit NIC (100 Mbit/s = 12.5 Mbytes/s). By using multicasting, this bandwidth can be effectively shared between viewers, especially when they are viewing approximately the same location. As we build our datasets to a realistic scale (thousands of pedestrians), this bandwidth saving will become increasingly important.

However, as soon as we will simulate more than 1000 agents, we need further reduction of the bandwidth usage.

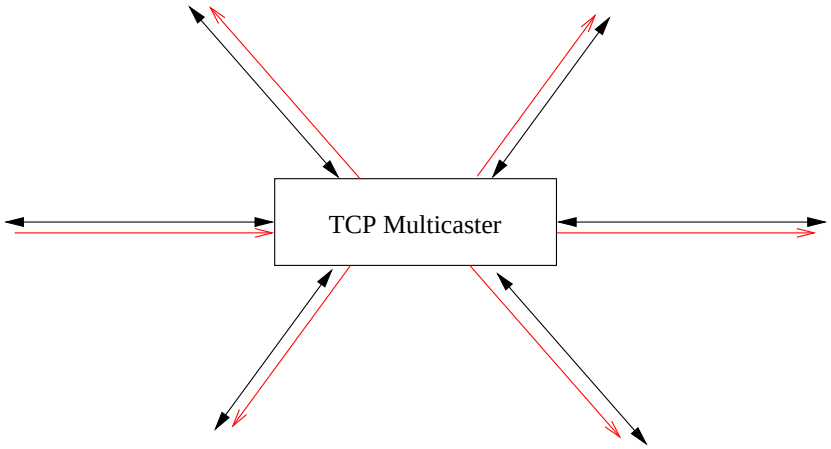
7.9 TCP Multicast Daemon

The multicast solution presented above is fast and efficient. However, not in all cases unreliable communication based by packet-loss can be tolerated. It would be possible to implement a error-correction protocol using UDP and multicasting. This would add much more complexity to all modules.

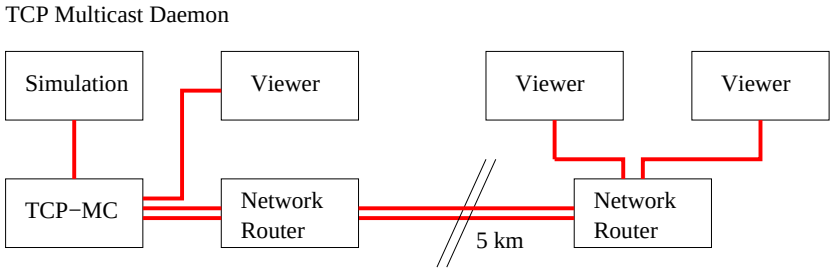
The TCP solution is reliable, but it is also a point-to-point and connection-based communication. It turns out that it is possible to build a broadcast-like infrastructure out of TCP connections using a *Broadcasting Daemon*. This is another module which accepts incoming TCP connections. If some data is sent through one of those connections, the data is broadcasted into each connection (Figure 7.3). The TCP Multicaster module must be running before any other module is started.

One advantage of packet-based communication (e.g. Multicasting and raw UDP) is that packets are presented in one piece to the application. This means that if there are two senders communicating with one receiver, packets are shuffled but not cut into pieces. If each packet contains one (or more) complete XML event message, the resulting mixture is still a valid XML data stream. Using TCP connections, it is possible that only part of a message arrives at the receiver. This is usually not a problem, since with the next `read` command, the missing part is received.

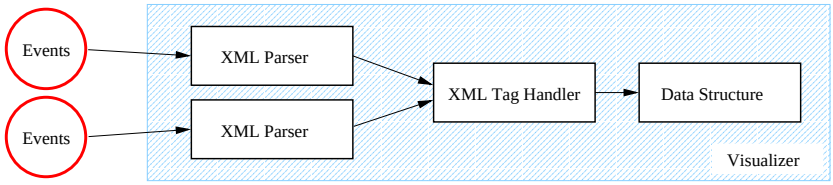
However, in the TCP Multicaster module multiple connections come together. If from one connection only a part of an message was read, there is no guarantee that not a part from another connection is



(a) TCP Multicaster functionality



(b) TCP Multicaster connecting modules



(c) Mixed XML streams inside a module

Figure 7.3: a) The TCP Multicaster is a communication module which accepts incoming TCP connections (black). If some data (red) is sent through one of those connections, the data is broadcasted into each connection. b) The TCP Multicaster uses multiple data connections (red) if there are multiple receivers c) It is possible to mix different kind of XML event streams in a module, since the parser is separated. Hoewever, it does access the same handler and data structure.

read before the second part from the first message. This yields in messed up XML messages, and the resulting data stream is no longer valid XML.

Therefore, a mechanism that sorts the messages in the TCP Multicaster is needed. This mechanism has to understand XML in order to determine when exactly an message ends. The simplest solution would be to attach an XML parser to each incoming connection, and only trigger the sender after the parser has fully recognized an XML tag. This, however, would be very slow.

A much faster alternative is to just look for matching `<` and `/>` pairs. Using full-blown XML, this would not help, because the `/>` inside a text section would also match. Luckily, we are using a specialized XML subset that does not use tags in the text section (see Section 7.10). A typical event message looks like this:

```
<event type="position" agent="42" x="588440.1" y="150281.4" />
```

which is parseable using a simple regular expression. Each incoming connection needs to have a data buffer, which hold incomplete messages. After each `read` on the connection, the data is appended to the buffer, which in turn is searched for a complete message. As soon as such a message is found, it is sent to all other connections, and deleted from the buffer. Note that the Multicaster does not have to understand the messages, it only has to recognize the end of a message. The part between the `<` and `/>` tokens is treated as ASCII text.

In the current implementation, we use this reliable multicast mechanism for events that really need to reach the destination, like the ones from and to the agent database. However, both this mechanism and the presented multicast using UDP method can be used together. For each message, the sender can chose one of the methods. For the receiver, it makes no difference how a message is received, since the module uses internally the same XML handler functions (see Figure 7.3c).

7.10 XML

7.10.1 Extensible ASCII Messages

Viewers are built so that they directly plug into the live system. The simulation sends agents' positions to the viewers, which allows to look at the scenario from a bird's eye view and observe how the agents move. As mentioned before, at this point there are two different viewers, one in 2D and mostly intended for debugging, and on in 3D.

For the 3D viewer, more data has to be sent, since it needs also the *altitude* of an agent. One needs the ability to *add* this value to the data stream in a way that there is no need to change existing viewers.

The Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.

The main advantage of XML is that there is no need to enforce a mandatory file format between the sender and the receiver.

Tags (e.g. `event`) and attributes (e.g. `agent="42"`) are not defined in any XML standard *per se*. These tags are introduced by the author of the XML document. This makes XML a perfect choice as a format for data exchange among different modules of a simulation. Whenever a module introduces a new kind of message, there is no need for a change in any of the modules that listens to the message stream—unless of course, a module wants to handle this new information specifically. This is also the case for a attribute introduced additionally.

It is, however, not possible to *change* the name of existing tags or attributes.

A typical *position update message* used in our simulation system encoded in a XML fragment looks like this:

```
<event type="position" agent="42"
x="588440.1" y="150281.4" />
```

XML is capable of handling more complicated structures, which are used for example in our *network definition files*:

```
<link id="2" from="49" to="50">
  <coord x="588446.30" y="158000.00"/>
  <coord x="588447.60" y="157995.60"/>
  <coord x="588455.60" y="158000.00"/>
</link>
```

The section between the start tag (<link>) and the end tag (</link>), called text section, is not needed in the position update messages, since all information is encoded as tag attributes directly.

Frequent changes in messages, something that happens often in research and development, are possible. Since the receiving modules search for "keyword=value" pairs, an additional attribute is simply ignored. As a result, there is no need to adopt existing modules. Further, due to the fact that messages are transmitted in plain text, debugging of communication is possible without further tools.

The example message is perfectly understandable by all modules if another attribute is added:

```
<event type="position" time="23"
agent="42" x="588440.1" y="150281.4" />
```

Since it is possible to attach existing XML parsers to any I/O stream or buffer, there is no difference between reading messages from a file or receive messages over the network. We tested XML over UDP and TCP, both versions are very flexible. Using UDP, however, if a packet containing a piece of a message is lost during transmission, the resulting stream is not necessarily still a valid XML document. This problem is circumvented by always sending complete messages in a packet. Recall that packets are guaranteed to be error free. UDP packets arriving with an error are discarded; TCP packets arriving with an error are re-transmitted.

Problems with XML are that (i) searching for "keyword=value" pairs is slow, (ii) if part of a XML stream is lost, the whole XML context may become invalid, and (iii) XML is plain text, so we have to

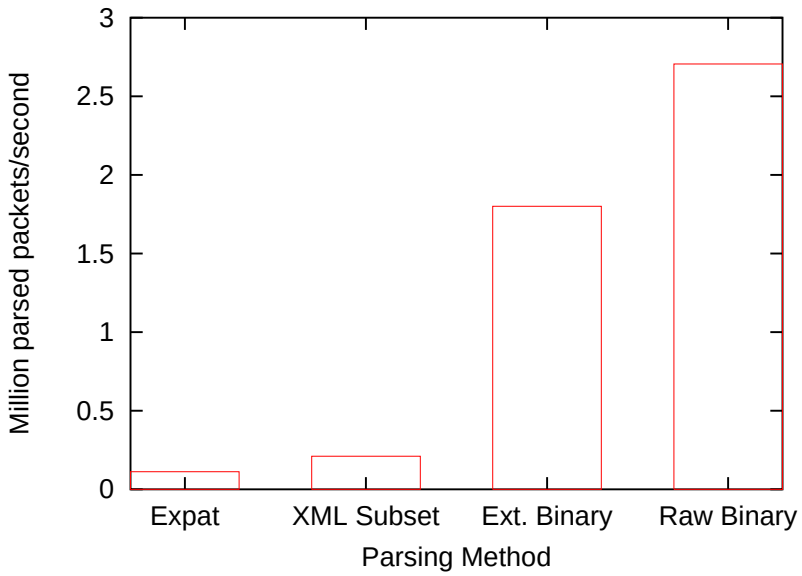


Figure 7.4: Comparison of parsing speed: Expat, XML Subset Parser, Extensible Binary Protocol vs. raw binary messages.

convert binary numbers into ASCII characters and back. To overcome problems (i) and (ii), we have developed a parser that is specialized in parsing a certain subset of the XML standard. This subset consists of simple tags with no nested tags inside:

```
<tag {attr="value"} />
```

A tag can have more than one attribute. The number of attributes is only limited by the amount of memory on the receivers machine. Note that all information is inside the attributes, and therefore inside the tag as well. Since there exists no text section as in XML, no nesting of tags is possible. However, it is possible to have multiple tags in one buffer that is given to the parser.

This subset parser is written as a substitute for existing XML parsers, such as Expat (Expat www page, accessed 2004). Therefore, the programmer's interface is exactly identical.

We measured the performance of our subset parser by parsing messages of different size for one second. These measurements were done on a 700MHz Pentium III. The messages were taken directly out of the machines main memory, so no network or disk access was involved. The results are that the off-the-shelf Expat parser parses 110'000 messages per second, while our specialized XML subset parser parses 210'000 messages per second (see Fig. 7.4). Thus, our XML subset parser is faster by a factor of 2.

Further, it is very important not to send small messages over the network. The overhead for each packet sent is 78 bytes (for Ethernet), and there may be a gap between packets for a proper aligning or collision detection. Whenever possible, combine multiple messages into one packet (see Figure 7.5).

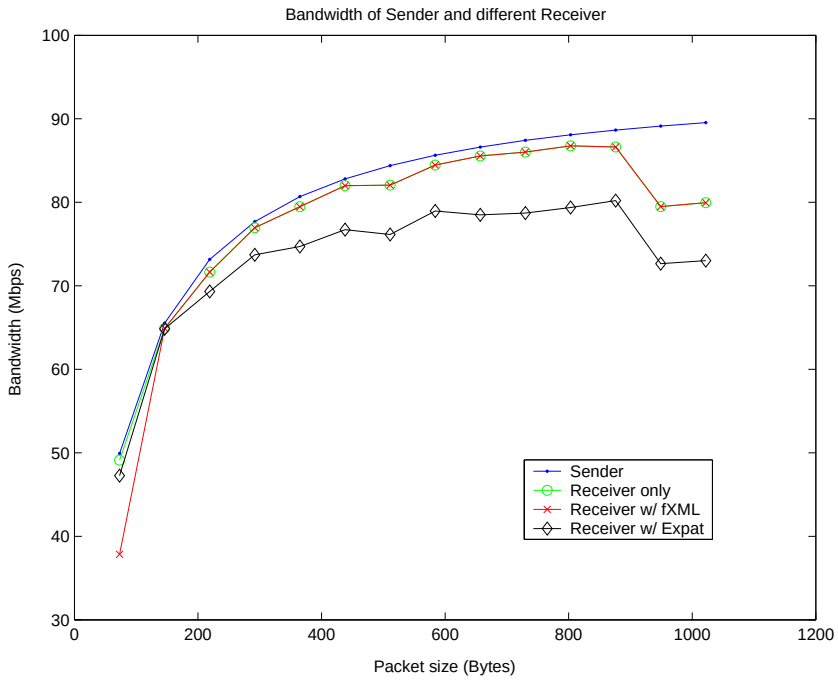


Figure 7.5: Number of messages the receiving module is able to parse. The more CPU cycles are used by the parser, the more messages are lost in the network (NIC buffers). Our fast XML subset parser (fXML) is able to parse almost as many messages as our binary reference parser.

7.10.2 Extensible Binary Protocol

The flexibility of XML is achieved at the cost of parsing ASCII files for tags and well-formatted entries. In multi-agent simulations, millions of individual agents are simulated and have to report to external applications such as graphical viewers or storage managers.

Our experiments have shown that the exchange of multi-agent data in the form of XML streams is very flexible but can reduce the bandwidth substantially due to the overhead of parsing the streams and due to converting numbers represented binary internally into ASCII and back.

In terms of CPU time, parsing the same type of streams data all the time is a waste. Often the structure of a message is known once the simulation is running.

A potential strategy is: initially, the consumer module connects to the producer module and receives a description of the available data. Then the consumer specifies which part of the data and which formatting it needs. Based on that specification, the producer begins to output binary streams formatted according to the consumer specification.

For instance, if agents are agents in a cable-car, a 2D viewer will only require x/y coordinates, while a 3D viewer will require x/y/z positions. The whole process can be seen as a user-defined on-the-fly serialization of the producer's objects (i.e. the agents) and their transfer over a high bandwidth binary channel.

We have implemented a protocol that follows these steps:

- A receiving module asks for certain data values to be transmitted in data messages. This is done by specifying the XML tag names, e.g. “agent” or “time”.

  An example request, as sent over the network, looks like

  `<request request_name="event" time="" agent="" x="" y="" />`.

- The sending module transmits a description of future data messages, e.g. `<description name="event" d_version="1" time="i" agent="i" x="d" y="d" >`. The field `d_version` is the sequence number of the description. It is unique for each new description sent.

- The sending module from then on sends plain data, packed according to the description into a binary buffer.

As soon as another receiving module asks for additional data values, they are included into the message as well, after a new description is sent to all the receiving modules. A description is sent every second as well, in case a receiving module lost a description or joined the system without requesting a new message format.

Our measurements have shown that this strategy yields in a factor of 17 faster than by using Expat, and in a factor of 8 faster than by using our XML subset parser (Figure 7.4, 3rd bar).

There exist approaches to develop a generic library that supports binary XML. Basically, two possibilities are present at the moment:

1. To place a stream compression algorithms. E.g. the deflation algorithm (used by gzip, zip and zlib) is a variation of LZ77 (Ziv and Lempel, 1977). It finds duplicated strings in the input data. The second occurrence of a string is replaced by a pointer to the previous string. XML usually is highly compressible, this saves a huge amount of bandwidth. However, this makes the consumption of CPU time even worse, since in addition to parsing the stream, is has to be unpacked before. This solution is useful only in networks where bandwidth is the major concern, which is not the case in computer clusters, and also becomes a less important problem in the Internet.

2. There are sophisticated libraries that provide an interface similar to an XML parser. However, in the background the XML stream is replaced by a binary data stream (XBIS XML Information Set Encoding www page, accessed 2005). According to the performance measurements on the project’s website, they were able to speed up parsing of documents up to a factor of about 7 for large documents, and to about 3.5 for small documents (100 bytes). This is comparable to our solution using the *extensible binary XML protocol*, which yields in a performance gain of factor 17. However, the performance measurement methods were different and one should not compare the results directly.

7.10.3 Sending raw binary data

For comparison, also the exchange of raw, binary messages was measured. It communicates the same amount of information as the above, but does nothing with it. This results in the 4th bar in Fig. 7.4. As one can see, the maximum possible speed as measured by this method is only about a third faster than our extensible binary protocol.

Since the values in the message are already binary and in correct order, there is nothing to parse. Limiting factors are the overhead to call a function, and to iterate over the values. This code fragment was used to get the values for Figure 7.4, 4th bar. It corresponds as close as possible to the other parsers. However, slight changes in this code result in different numbers.

```
void handler(void *values) {
  sclass *ptr;
  ptr = (sclass*)values;
  numer_of_parsed_messages++;
  for (int ii = 0; ii < 32; ii++) {
    // iterate over the values,
    // but do nothing with them
  }
}
```

7.11 Conclusions

Using the Extensible Binary Protocol, it was possible to visualize scenarios containing more than 1000 agents running more than 100 times faster than real time. An agent number and the x and y co-ordinates of a position update message consume 20 bytes, when packed binary. It is therefore possible to transmit up to 75 agent positions per packet. If the viewer is able to receive 200’000 packets per second, we are able to display 15 million agents per second. Since the viewer uses its host’s CPU cycles for drawing the graphical output as well, the actual number drops again substantially.

Our currently largest street network scenario, Greater Zürich, consists of 166’000 nodes and over 200’000 links. On a busy rush-hour simulation, there are up to 100’000 agents moving on these links. Since the traffic simulation is able to run at 800 times real-time speed, displaying the agents position in every timestep is still not possible.

7.12 Discussion

It is important to note that the task of the mobility simulation is simply to send out events about what happens; all interpretation is left to the mental modules. In contrast to most other simulations in the area of mobility research, the simulation itself does not perform any kind of data aggregation. For example, link travel times are not aggregated into time bins, but instead link entry and link exit events are communicated every time they happen. If some external module, e.g. the router, wants to construct aggregated link travel times from this information, it is up to that module to perform the necessary aggregation. Other modules, however, may need different information, for example specific progress

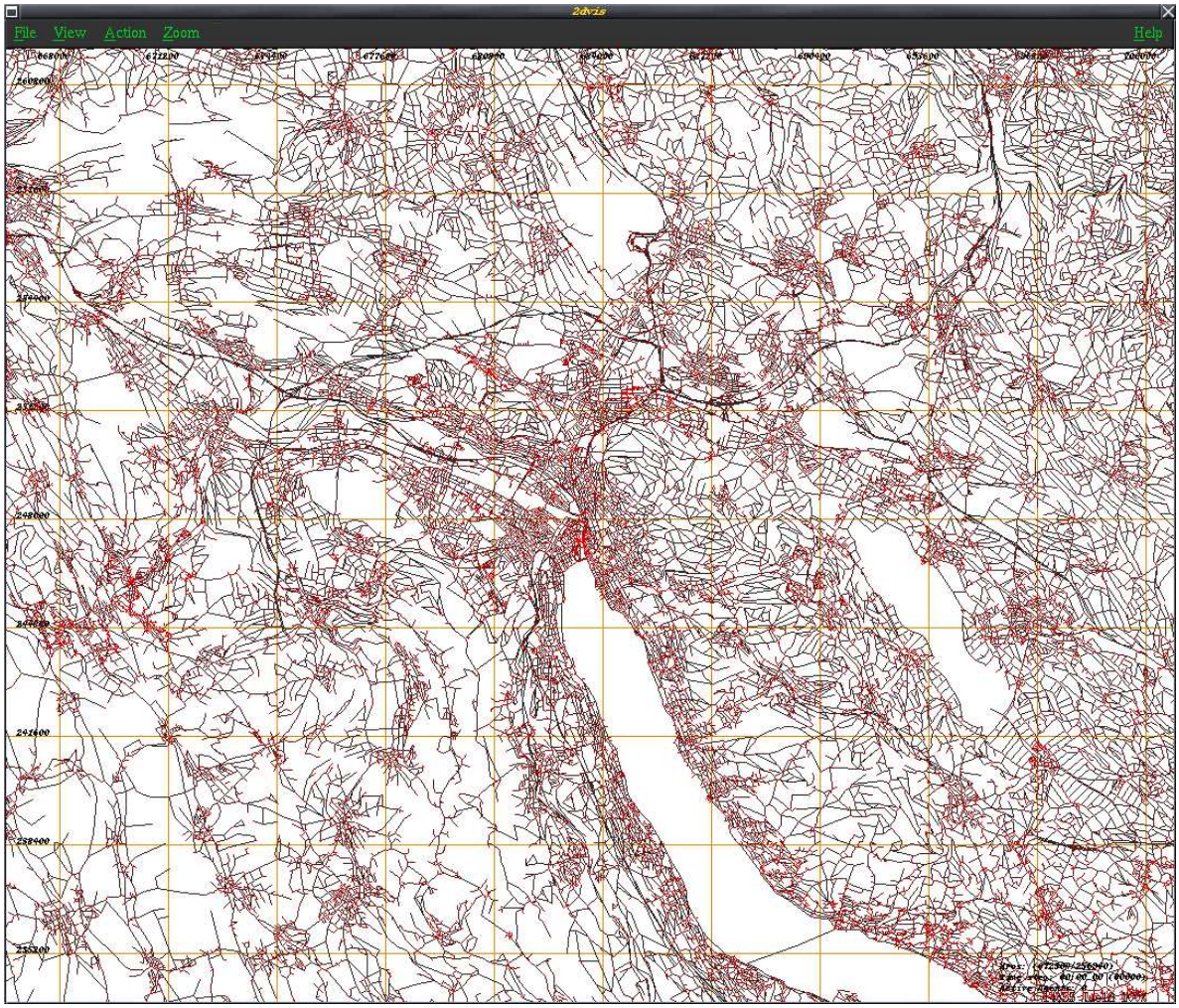


Figure 7.6: Our largest street network scenario, Greater Zürich, consists of 166'000 nodes and 205'000 links. On a busy rush-hour simulation, there are up to 100'000 agents moving on these links.

reports for individual agents, which they can extract from the same stream of events. This would no longer be possible if the simulation had aggregated the link entry/exit information into link travel times.

Despite this clean separation – the mobility simulation and the modules in the physical layer compute "events", all interpretation is left to mental modules – there are conceptual and computational limits to this approach. For example, reporting everything that an agent sees in every given time step would be computationally too slow to be useful. In consequence, some filtering has to take place "at the source" (i.e. in the simulation), which corresponds to some kind of preprocessing similar to what real people's brains do. This is once more related to human intelligence, which is not well understood. However, also once more it is possible to pragmatically make progress. For example, it is possible to report only a random fraction of the objects that the agent "sees". Calibration and validation of these approaches will be interesting future projects.

Chapter 8

Synchronization

8.1 Introduction

We need a way to synchronize the modules. There are on one side multiple mobility simulations like car (traffic), cable-car and pedestrian simulations. Also there are higher level modules like the agent database module or the route generator module.

The goal was to couple these modules as loose as possible. This is because

- modules should be able to run as independently as possible
- third party modules (that we cannot modify) can be *wrapped* inside a hull module that handles the communication and also the synchronization.

For the synchronization we wanted to use the same kind of communication channels as for the other events. This can, but does not have to, be the same instance of the channel. The synchronization events are clearly separated from the other events by the `type` tag. However, it could yield a better performance if these channels are separated completely. It is important that synchronization messages arrive at the receiver. If one module sends too much data for the others to process, there is no sense in sending a notification to stop using the same channel, since that message will be delayed in the network for too long. In the current implementation, everything uses the same events channel. With the current scenario size, this does not cause any problems.

There are modules that do not need synchronization. For example the route generator module should answer route requests all the time, even when the simulation itself is stopped. Also it builds its internal view of the world based on events that all contain a time-stamp. The route generator modules therefore does not need to track the global time. However, causality is still preserved, the router cannot use information that was not reported by an agent yet.

We use modules that behave like *time driven simulations*, e.g. the pedestrian mobility simulation. However, there are as well modules that do not follow a continuous time and belong therefore to the category of *event driven simulations*, e.g. the agent database. Also the framework, seen with the modules as smallest entities, can be seen as event driven.

In this chapter we do *not* look at the synchronization that takes place *inside* separate modules, if parallelized itself. These modules need synchronization, but this is achieved using internal and independent mechanisms (e.g. using MPI barriers, see Cetin, 2005).

Since there exist parallel algorithms and specially parallel simulations, the need for synchronizing these is recognized. Most of the algorithms proposed seem to be solutions to a special problem. And also the solution presented in this chapter is not the best for every purpose: we have to deal with 3rd party modules that we cannot modify.

Lamport (Lamport, 1978) shows that real-time temporal order, simultaneity, and causality between events in a distributed system use the same concepts as *special relativity*. He shows that, because the transmission speed of messages is limited, things that happen at the same time at different locations cannot influence each other. It takes some time before the message that something has happened reaches all parts of the simulation.

The Flip-Tick Architecture (FTA, Veit and Richter, 2000) is used for *intelligent systems*, which are mainly control programs for intelligent robots. The functionality of FTA is available in a C++ Library. Every modules hast to post a message to a global tag-board as soon as it would be ready to simulate one more timestep. The tag-boards sends a broadcast message, as soon as every module is ready. This approach guarantees that every message was received before the next simulation time step starts. However, it is rather inefficient if the time-steps are small. How long a timestep can be depends on the problem: basically this is the time any simulation module can run without input from any other module.

There exist sophisticated solutions that make sure the time is consistent, but do not consume too much resources. Time Warp (Jefferson, 1985) is based on *rollbacks*, which can be ruled out for our system, since we want to embed 3rd party modules that may not support rollbacks. However, it would be possible to mix modules that support rollback with some that do not. Those that can make use of this functionality can try to optimize the CPU usage, while others just wait. In our implementation, none of the modules support rollback.

If one module sends a message, it takes some time for the message to reach the receiver. As long as this message is traveling, there is no way to stop it. A system that wants to synchronize communicating modules has to take care of these *transient messages* as well (Mattern, 1993). For our purpose, these messages can be neglected, since the network delay is much smaller than the time a human brain would need to react to an event (see Equation 8.4).

There exist other frameworks for distributed simulations, but it turns out that they are i) complicated, or ii) not flexible enough for our purpose.

The *High Level Architecture* (Dahmann et al., 1997; Riley et al., 2000; Fujimoto, 1998, 1999) is a general purpose architecture developed under the leadership of the Defense Modeling and Simulation Office (DMSO) of the U.S. Departement of Defense. It is used to support reuse and interoperability across the large numbers of different types of simulations. It is mainly used for battlefield simulations. (Nicol and Fujimoto, 1994) give a introduction what is the state of the art in synchronization of parallel simulations when HLA was developed.

The disadvantage of HLA is its complexity, the description of the protocol is described in books containing several hundred pages. It would be possible to use HLA or most of the other approaches for our framework – as long as it is possible to integrate 3rd party modules. For our simulation, a much simpler

approach is sufficient.

In this chapter, two simple methods to keep time in our framework are described. Section 8.2 shows a simple algorithm and its problems, while Section 8.3 presents a more sophisticated algorithm that solves that problem.

8.2 A Simple Mechanism: Suspend-On-Demand

8.2.1 Description

Every module runs with its own speed. If a module needs more time in order to achieve a task, it can suspend all other modules by sending a `suspend` message in the event stream. After it is done with its task, it has to send a `continue` message to all modules.

An example for the Suspend-On-Demand algorithm, we look at the scenario where the agent database module needs to calculate a new route:

- All modules run on their own speed. A hiker encounters a situation that needs replanning, and asks the agent database module for a new route.
- The agent database module looks up in its internal tables if there exists already a route that matches the new situation. If there is such a route, the agent database module sends this information instantly to the simulation. However, if there is no route available, it has to request a route from the route generator module. Since generating a route takes some time, the agent database suspends all simulation modules. After receiving the `suspend` message, all modules halt.
- The agent database module asks the route generator module (which does not listen to `suspend` messages) for a route.
- The route generator module answers the route request.
- As soon as the new route is available to the agent database module, it sends this route to the simulation.
- Finally, the agent database module sends a `continue` message to all modules. All the modules continue to run.

However, due to network latency $T_{Latency}$, there is a short time during which the simulation (or any other module) can keep running even if the `suspend` message was already sent (see Fig. 8.1).

Because a `suspend` message is usually a result to a message sent over the network as well, the latency has to be multiplied by two. The latency needs to be multiplied by the speed the simulation runs, since the simulation is able to simulate more than one second per wall-clock second:

$$T = S_{Process} * 2 * T_{Latency} \ . \tag{8.1}$$

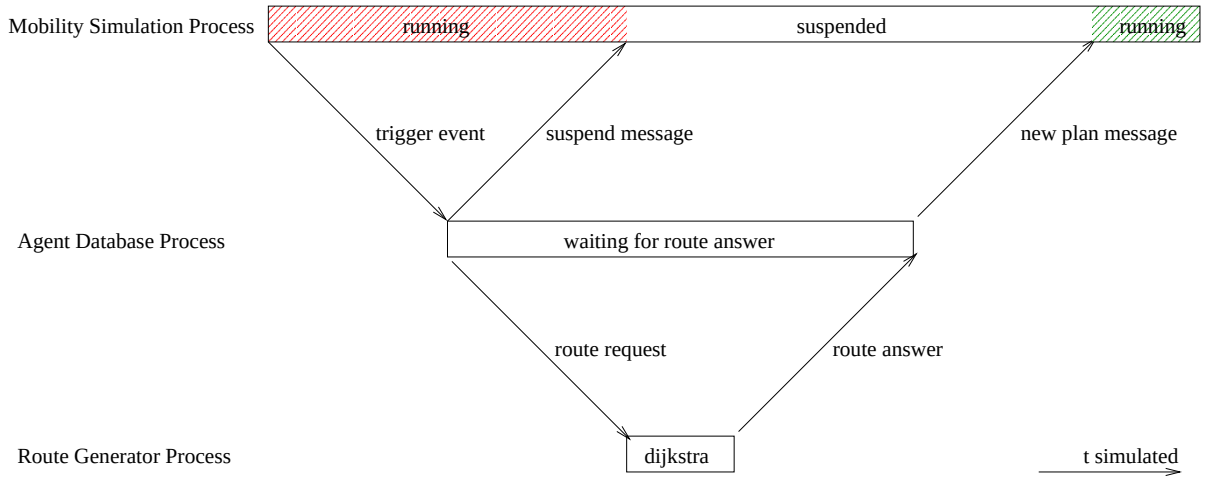


Figure 8.1: A problem with the Suspend-On-Demand algorithm is that even if the agent database module decides infinitely fast that a suspend is needed, the mobility simulation has plenty of time to run further (red area), due to the delay introduced by the network. If a module runs very fast (worst case: infinitely fast, i.e. an empty simulation), it is not possible to suspend this module before its time is too far advanced.

Using an average network latency of 0.1 ms (e.g. inside Xibalba cluster) and a process speed ($S_{Process}$) of 100 times wall-clock time (in 1 second, 100 seconds can be simulated), we get

$$T = 100s/s * 2 \times 0.1ms = 20ms \ .$$

A delay of $20ms$ is not much, given that we model human behavior. However, it turns out that the agent database module process cannot answer immediately to an event: it has to parse the received data packet (done in network layer) and to parse the actual message as well ($T_{ToReact}$).

$$T = S_{Process} * (2 * T_{Latency} + T_{ToReact}) \tag{8.2}$$

In chapter 7 it was shown that, depending on the transport protocol used, this time varies. The slowest parsing method, expat, is able to parse about 110'000 messages/s, which is about 9 μs or 0.009 ms per message. This is negligible, compared to the time around 0.1 ms (wall clock time) introduced by the network.

This could lead to the assumption that in the case that a modules is able to react sufficiently fast to an incoming message, this algorithm is good enough. The mobility simulation cannot interrupt the execution of a timestep if a message arrives. As long as the T is smaller than the duration h of a timestep ($0.5s$, see Eq. 3.2), this does not make any difference. Specially in socio-physical simulations, this small delay reflects the delay $T_{ReactionTime}$ the human brain also needs if it was provided with new information.

However, if a module runs much faster than the $S_{Process}$ assumes above, it is not possible to suspend this module before its time is to far advanced. This could be the case with a simulation that has finished

to simulate all plans for all agents.[1] For example, if for dinner all agents are inside the hotel and have no previously scheduled activities later in that day, the agent database module would not be able to assign an activity later in the evening, since the mobility simulation would have proceed to midnight already.

This yields to the maximal speed $S_{Process}$ a process can run with, in order that causality is not disregarded. Eq. 8.2 gives the number of time steps that the system needs to react. As long as the reaction time is larger than this, we are ok:

$$T_{ReactionTime} > S_{Process} * (2 * T_{Latency} + T_{ToReact}) \tag{8.3}$$

From there, one gets an upper limit on the process speed:

$$S_{Process} < \frac{T_{ReactionTime}}{2\,T_{Lat} + T_{ToReact}} \; . \tag{8.4}$$

8.3 A Slightly More Sophisticated Mechanism: Advance-To

8.3.1 Description

Like with the first mechanism, all modules run on their own speed and do not share a global time. However here, every module reports to a special *controller module* the time for which it could run without external information. For the pedestrian simulation module, this could be the time when the first plan seems to end, for the agent database module this could be some reasonable time that reflects the time a human needs to react to external influences (e.g. 1 – 10 minutes).

A *controller module* then calculates the minimum of these times, and distributes this time in an `advance to` message to all other modules.

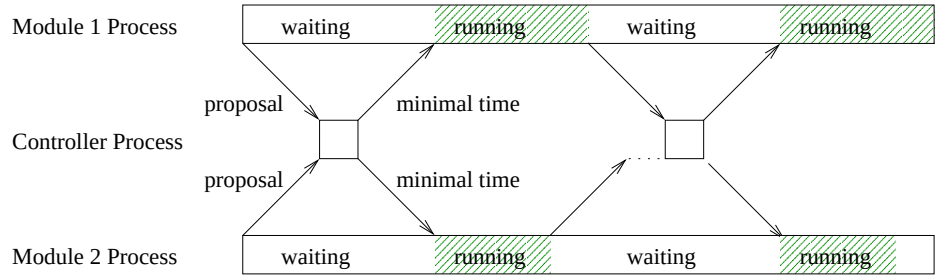
Every module is allowed to run as far as the global advanceto time is set.

This approach solves several problems at once:

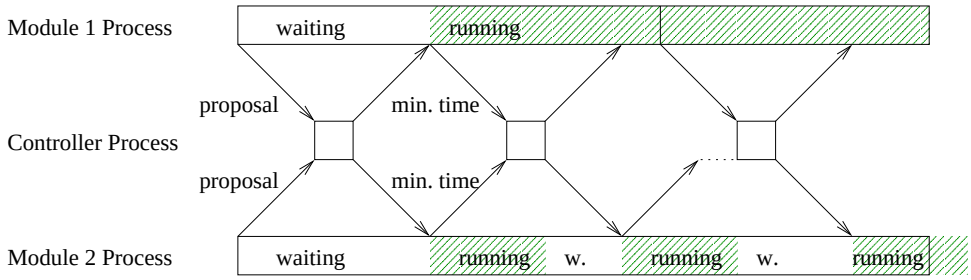
1. The modules are synchronized as accurately that for a observer, everything looks fine.
2. Every module is able to stop the whole simulation system, as soon as it detects that it needs more time than usual for a task.
3. On average, there is almost no overhead.
4. The implementation is simple.

This approach is similar to the Time Warp (Jefferson, 1985) Algorithm, however here, instead of using rollback if one process run for too long, the damage is limited by the advanceto time. Still, our algorithm guarantees a correct causality in the whole system, as long as every module choses the correct advanceto time.

[1] Since no forces between agents have to be computed, the mobility simulation is able to run infinitely fast. This is also the case with empty queue simulations for traffic simulations, if they are implemented in a way that keeps track of occupied links (Cetin, 2005).



(a) requesting an new advanceto time after reaching time



(b) requesting an new advanceto time immediately

Figure 8.2: Depending on when each module proposes a new advanceto time to the controller, it is possible that one modules runs without discontinuation. In (a), the module 2 process returns before the module 1 process, and the controller is then waiting for the second message to arrive. In (b), the module 1 process is still busy working on the first "advanceTo" message when the second such message arrives.

Ideally, the advanceto time is never reached by the slowest module. If every module submits a new proposal for the advanceto time before the slowest module is at that point, the new time broadcast by the controller is just a bit farther in the future (Figure 8.2).

If all modules propose an advanceTo time that makes sense, there are no problems with this approach. For example, if the agent database module uses advanceto times of 10 minutes, and wants to communicate with a very fast simulation (e.g. empty pedestrian simulation or a cable-car simulation), there is no way that the agent database module can influence the simulation, say, 2 minutes after it has sent the `advanceto` message. In the worst case, the mobility simulation has run 8 minutes too far, and a hiker potentially missed a cable-car (Figure 8.3). However, if the agent database has chosen an advanceTo time of 10 minutes, this also means that whatever happens, the agent database needs 10 minutes to react to an event. For example: the cable-car simulation decides to jump 10 minutes forward, because it is empty. An agent walks 2 minutes later near the station and, because he can see the cable-car, decides to step in. If the agent sees the cable-car, it stopps immediately and has to submit an event, which is received and processed by the agent database. The agent database, however, proposed an advanceTo

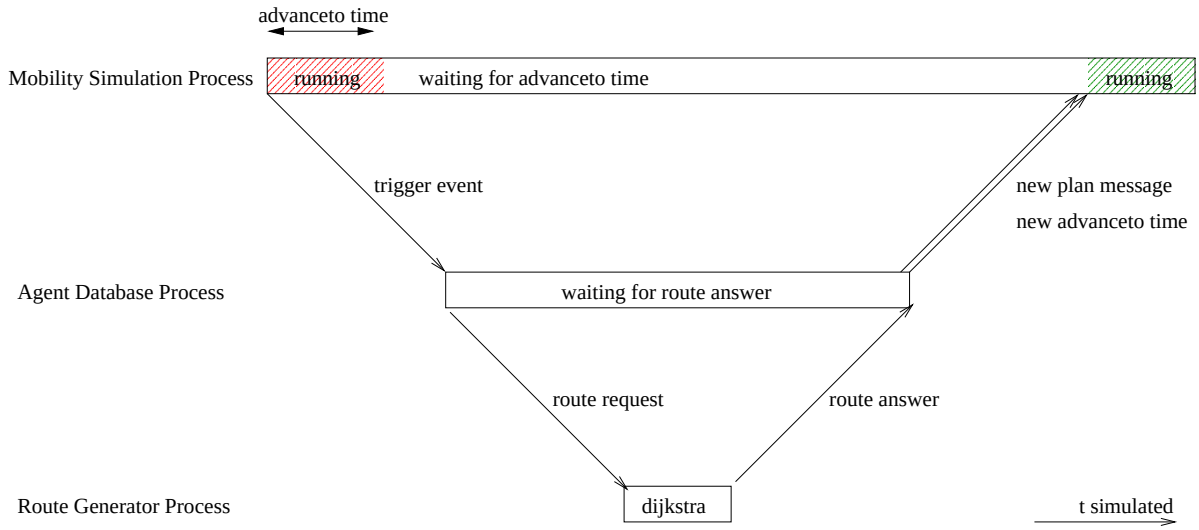


Figure 8.3: The problem with the Suspend-On-Demand algorithm is less serious with the Advance-To algorithm: even if a module runs very fast, it will wait after it has reached the next advanceto time.

time of 10 minutes, which means that the decision making process also takes at least 10 minutes. The agent therefore waits in front of the station for 10 minutes, and catches the next cable-car.

In consequence, advanceTo times need to be set to values that are behaviorally plausible. For example, it is plausible to assume that a hiker needs a couple of minutes to orient him-/herself at the cable car station, buy a ticket, etc.

8.4 Discussion

There exist already a lot of algorithms to synchronize modules of a simulation. Some trivial approaches (e.g. using fixed time barriers on each timestep) introduce a large overhead. Others are part of a large simulation framework that is too complex to be implemented in a small experimental framework as we have here.

In this chapter, two algorithms for synchronizing the modules were presented. The first example was a simple *Suspend-On-Demand* Algorithm, which could be sufficient for simulations that tolerate a slow reaction time. However, if the modules do not require more or less the same time to simulate a timestep, it could happen that one module has simulated much too far, and a `suspend` message is too late. In the worst case, when a module is able to run infinitely fast (e.g. the cable-car simulation), synchronization is no longer possible at all.

A improved version called *Advance-To* takes the different simulation speeds into account. This method uses dynamic barriers, which are, in the best case, shifted forward with the same speed as the slowest module of the simulation proceeds. In that case, the system is not slowed down at all.

Part III

Application

Chapter 9

Sensitivity Studies

In order to validate that the framework and the individual modules work as expected, sensitivity studies have been designed. Out of these, six examples have been selected in order to demonstrate the most important aspects of this work,

The following studies are presented in this chapter:

One Agent	*Multiple Agents*
Intraday Re-Planning (9.1.1)	Same Destination, Different Paths (9.2.1)
Generalized Cost Function (9.1.2)	Two Destinations to Chose From (9.2.2)
Time Dependant Paths (9.1.3)	“Exploration Scenario” (9.2.3)

9.1 Single Agent Scenarios

9.1.1 Intraday Re-Planning

Feature to Demonstrate

As stated in Chapter 2, one of the advantages of this framework is that an agent is capable to react faster to changes in its environment than file-based simulation systems. There is no need to wait for the next iteration for plan changes to become active.

The agent database (or any other module, theoretically) is able to monitor the agents in real time, and, if something happens, can react immediately by sending new orders to the agent.

How Can This Be Shown?

One of the simplest ways to show that this feature works in this framework is to let the agent react to a rain drop falling nearby.

The agent reports raindrops as events, and the agent database is able to react to this events directly. Using a special event handler in the agent database (see below), it is possible to make it react specifically

to this reported event. One possible reaction is to send the agent home immediately. Assuming that it wants to stay dry, it also avoids further rain.

Description of the Scenario

One agent is hiking somwhere in the simulated area. Preferably it should hike *away* from the hotel, otherwise it is difficult to show that it changes its direction after it was hit by a raindrop. Using the 2-dimensional visulaizer's ability to send rain events, some drops are generated near the agent's current position.

This event is sent over the events channel:

```
<event type="rain" time="100" x="588161" y="150664" />
```

The origin of this event does not matter. It could be sent by a special *weather module* as well.

The mobility simulation receives these events and processes them. It checks if an agent is near the position of that rain event. If so, it sends a `hitbyraindrop` event back into the events channel. This event contains the agent's ID and its current position, which can vary a little bit from the position of the raindrop:

```
<event type="hitbyraindrop" id="42" time="100" x="588162" y="150664"/>
```

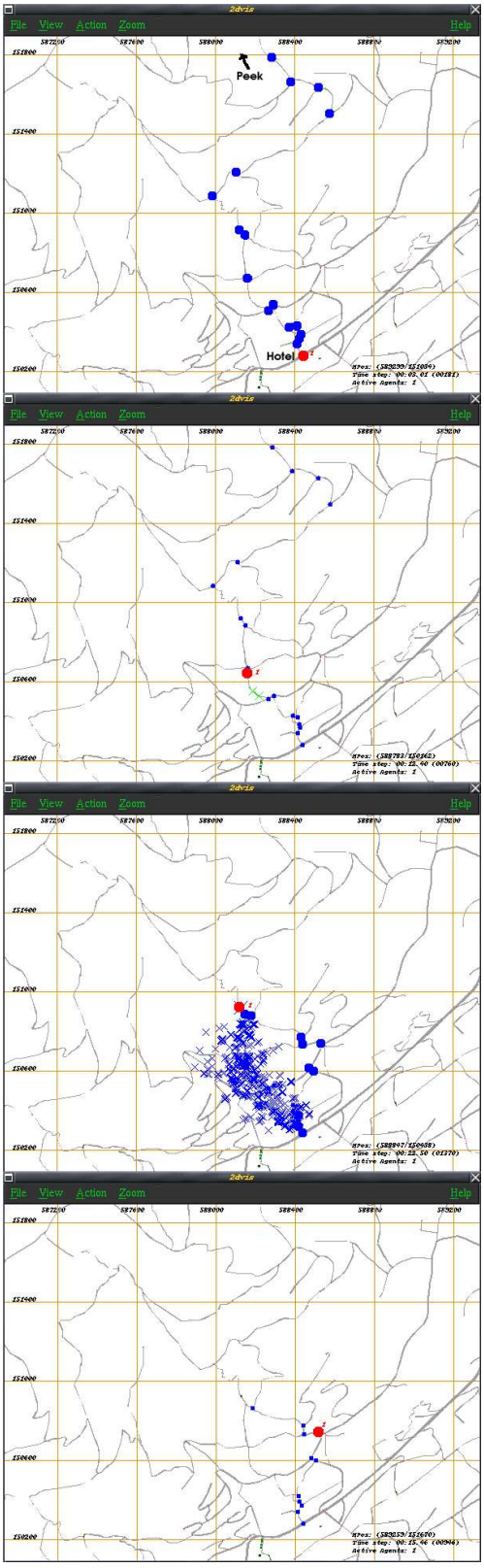
The agent is just reporting the fact that there happened something, it continues to walk on the route assigned to it earlier. However, the agent database receives the event sent by the agent, and decides that there happened something which is worth to do something. It therefore stops the agent immediately by sending such an order through the braincomm channel:

```
<stop agent="42">
```

The agent stops immediately. It might be possible that the agent walked further since the agent database had decided to stop the agent. Its location is not known exactly. The agent sends therefore an event that includes its current position.

The agent database then requests a new route from the route generator. The preferences of the hiker are all set to 0 ("don't care", the default), except for `weight_rain`, which is set to 1 ("dislike"), unregarded the original preferences of the hiker:

```
if ($e =~ /type=\"hitbyraindrop\"/) {
  [..]
  if ($action[$agent] eq 'senthome') { # send home only once per hike
    print "[brain] agent $agent is heading home already\n";
  } else {
    $action[$agent] = 'senthome';
    print "[brain] agent $agent has been hit by a rain drop\n";
    print $brain "<stop agent=\"$agent\">\n";
    print "[brain] requesting route from curr. position to hotel\n";
    print $event "<event type=\"route_request\"
                         from=\"$link\"
                         to=\"$hotel\"
                         agent=\"$agent\"
```



The agent (red dot) received a route from the hotel to the mountain peak - visible through the blue dots (slightly enlarged) along the path.

The agent (red) is hiking along the route it has received (blue). It reports no rain on its way up.

The agent is hit by the first raindrop (blue crosses). However, this is not the only raindrop, there were more in the area where the agent just hiked through. The new route (blue dots, enlarged) that leads directly to the hotel avoids this area, beacause the router learns the location of the raindrops instantly (see Chapter 4).

The agent (red) is on the way back to the hotel.

Figure 9.1: After an agent was hit by a raindrop, it is heading home on the shortest path immediately, but avoids areas where rain was detected.

```
                        time=\"$time\"
                        weight_rain=\"1\"
                    />\n";
    }
  }
```

Of course it would be possible to keep certain preferences of the hiker even for this kind of request – for example those that come from physical constraints, like the ability to walk over uneven terrain. However, since all the preferences supported by the system at the moment are of social nature (“forest”, “view” etc.), this is not implemented yet.

The router generator calculates the “best” route from the current location to the hotel. For the router there is no difference from this request to any other: it can not tell that this request is a response to an special event. The calculated route is sent back to the agent database, which threats is as a normal route answer and forwards it to the mobility simulation.

The mobility simulation receives the new route, assignes it to the stopped agent, which starts to execute it.

The agent will eventually reach the hotel. The mobility simulation then sends a `reached_ destination` event. It is up to the brain if the agent should stay in the hotel or try another hike (may be there is no rain on the other side of the valley).

Results and Discussion

After an agent was hit by a raindrop, it is heading home on the shortest path immediately, but avoids areas where rain was detected (see Figure 9.1). The new route is visible through the white dots at some nodes. The shortest path would be slightly more to the left, directly through the area with rain. For a more detailed explanation why the router has chosen that route, see Section 9.1.2 or Chapter 4.

It can be said that intraday re-planning works fine, agents react to something that happens during an iteration immediately.

9.1.2 Generalized Cost Function in Router

Feature to Demonstrate

The route generator was designed to use other criteria than just the fastest path for the route calculation. E.g. rain and sun can be liked or not. This is achieved using a generalized cost function (see Chapter 4).

How Can This Be Shown?

Since this is a feature of the route generator and its internal representation of the world only, no simulation is needed. A route request is sent, requesting a route from one activity to another.

Before the route generator has learned anything from the events, the only criterion it can use is the shortest path – as the route generator from the traffic simulation project does. Therefore, a route request

without any further initialization of the route generator should yield in the shortest path. Also the preferences of the hiker should not make any difference.

If `rain` events are detected on that shortest path, the route generator can use this information. If the hiker does prefer to walk without rain, the new route should lead slightly around the region with rain. If the preferences are neutral or the hiker even likes rain, the route must not change, since the route is already the best choice.

If `sun` events are detected slightly outside the shortest path, the route should bend a bit so that the agents can walk inside the bright area – as long as they like the sun.

Description of the Scenario

As said before, no simulation is needed. We need the router and a way to request routes and provide it with the events to build the internal map. This can be either a simple Perl script, or, since the events stream is XML and ASCII based, an telnet session to the TCP Multicast daemon.

The TCP Multicast daemon is started, then the route generator and the visualizer. Using telnet to port 10002 of `localhost`, a connection directly into the events channel is made. A route can now be requested by typing:

```
<event type="route_request" from="4477" to="3623"
       agent="1" time="0" weight_rain="0"
/>
```

The answer is visible directly in the telnet window. Additionally, the visualizer caught this message as well, and displays it (Figure 9.2a).

```
<event type="route" agent="1" from="4477" time="0" to="3623"
       weight_rain="0"
       route="4086 4125 4122 4034 4079 [..] 3587 3338"
       estimated_cost="1585.200000" estimated_time="1586.142442"
/>
```

The `estimated_cost` and the `estimated_time` both have the same value (the difference is an artefact from the small randomization, see Appendix A).

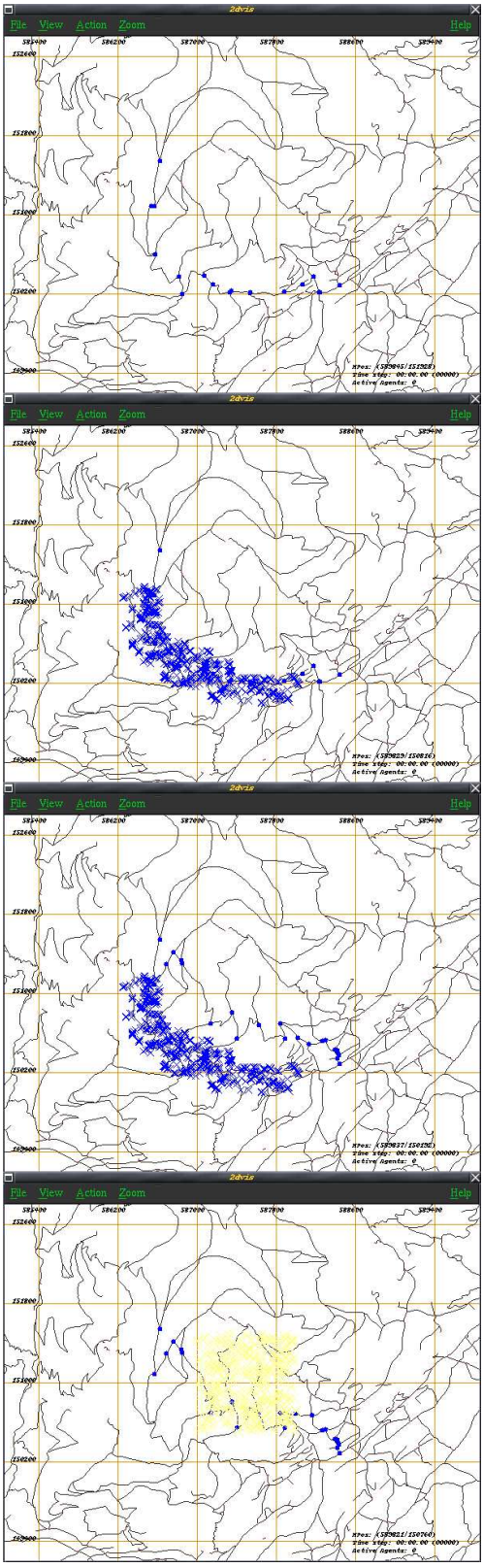
If we chose a different weight for rain, e.g. 1, the route does not change.

```
<event type="route_request" from="4477" to="3623"
       agent="1" time="0" weight_rain="1"
/>
```

Also, the retured time and cost are the same as before. The router has no additional information than estimated link travel times.

Now, some rain is generated using the visualizer's feature. The rain covers the fastest route. During the rain generation, a lot of these rain events can be seen in the telnet window:

```
<event type="rain" id="0" time="0" x="587909.000000" y="150040.000000"/>
```



The first route request gives back the fastest route (blue dots), since no additional information is known to the router.

After rain was “painted” using the visualizer, the answer stays the same – as long as the hiker does not care about rain (weight_rain=0).

If the hiker *does* care about rain (weight_rain=1), the route avoids the rain.

If a hiker likes areas with sunshine (yellow), the route leads directly throug that area.

Figure 9.2: It is possible to use other criteria than just the fastest path in the route generator. E.g. rain and sun can be liked or not.

If there was another 2-dimensional visualizer attached to the events stream at that moment, rain events would show up there as well.

If we request once more exactly the same route (using the same weight 0 for rain – do not care), the returned route is still the fastest route (Figure 9.2b). However if we provide a weight higher that 0, e.g. 1:

```
<event type="route_request" from="4477" to="3623"
       agent="1" time="0" weight_rain="1"
/>
```

the route changes (Figure 9.2c):

```
<event type="route" agent="1" from="4477" time="0" to="3623"
       weight_rain="1"
       route="4086 4052 4036 4024 4000 [..] 3587 3338"
       estimated_cost="3330.993614" estimated_time="1586.219246"
/>
```

For the next experiment, the route generator was restarted in order to clean up its mental representation. We use the visualizers feature to let the sun shine on a certain part of the scenario, lots of sun events show up in the telnet window:

```
<event type="sun" id="0" time="0" x="587897.560266" y="150636.074612" />
```

A new route request, this time using a *negative* weight for the sun (-0.3), yields in a route that leads directly through the area with sunshine (Figure 9.2d).

```
<event type="route_request" from="4477" to="3623"
       agent="1" time="0" weight_sun="-0.3"
/>
```

Results and Discussion

The results (Figure 9.2) show that the route generator is able to produce rotues according the preferences of the hikers.

9.1.3 Time Dependent Routes

Feature to Demonstrate

This experiment was done in order to demonstrate that not only the link travel times are stored in multiple time bins, but the other attributes (e.g. sun, rain) as well. If there was sun on one side of the valley in the morning but not in the afternoon, the agent will still chose that side in the morning of the other day (but not in the afternoon).

How Can This Be Shown?

Again, no simulation is needed for this experiment. The router learns from these events directly. In the route request and in the events, the time stamp can be varied.

Description of the Scenario

The set up is as in the last experiment.

First, the router needs to learn that in the morning, on one side of the valley the sun is shining. Again, this can be achieved by using the visualizer.

The sun events from the visualizer are accumulated in the first time bin, because of their timestamp (Figure 9.3a).

To pretend that there is no sun in the afternoon, we do not have to act. Time bins are per default empty (Figure 9.3b).

If we now request a route from the route generator with a preference for sunshine (`weight_sun = "-0.3"`), for a route in the morning (`time="1"`), a route through the area where sunshine is returned (Figure 9.3c).

```
<event type="route_request" from="4477" to="3623"
       agent="1" time="1" weight_sun="-0.3"
/>
```

If we request a route from the route generator for a route in the afternoon (six hours later, i.e. `time = "21601"`), the returned route is again the shortest path (Figure 9.3d).

```
<event type="route_request" from="4477" to="3623"
       agent="1" time="21601" weight_sun="-0.3"
/>
```

Results and Discussion

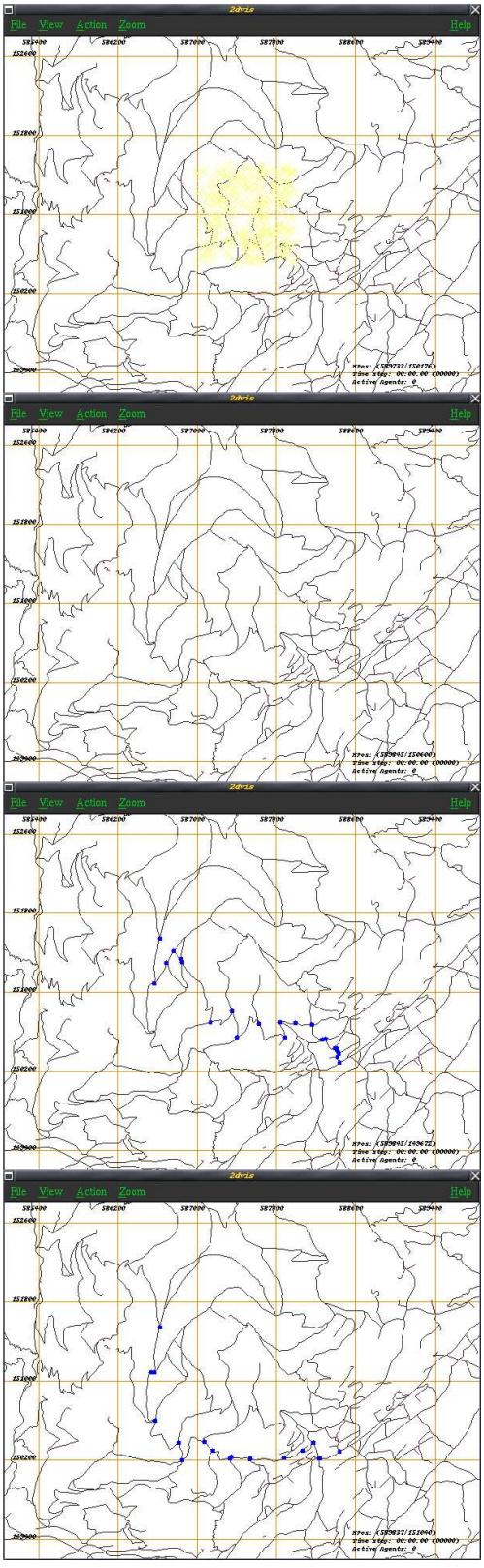
As demonstrated in Figure 9.3, the route generator does not only accumulate link travel time in multiple time bins, but all the other attributes as well.

9.2 Multi Agent Scenarios

9.2.1 Same Destination, Different Paths

Feature to Demonstrate

Every hiker has his own preferences assigned, based on the demographic data. The “best” route to a given destination might be different for two hikers.



In the morning of the first day, the sun is shining inside a certain area.

In the afternoon of the same day, the sun disappeared.

On the second day, the agents still prefere the area where the day before the sun was shining (even if there is no sun at the second day).

However, they have learned that in the afternoon usually there is no sun in that area.

Figure 9.3: This experiment was done in order to demonstrate that not only the link travel times are stored in multiple time bins, but the other attributes (e.g. sun, rain) as well. If there was sun on one side of the valley in the morning but not in the afternoon, the agent will still chose that side in the morning of the other day (but not in the afternoon).

How Can This Be Shown?

Two groups of agents are built, each group has different preferences. One group likes forest, the other does not. The same for scenaric view, but the other way round. Agents initially receive the same route from the route generator, since they have same activity and the route generator has no information to chose from other than link travel time. However, they soon should spread out as the route generator builds its mental map out of forest and view events sent by the (hikers in) the mobility simulation. The separation of the two groups should be visible.

Description of the Scenario

For this experiment, the mobility simulation is required. The agent database used here assigns a simple task to each agent: start at the hotel, hike to the peak of the mountain, and hike back. Do this as long as the simulation runs.

In the agent database module, each group of hikers is assigned different preferences, as shown in the code example blow. Note that this belongs *not* into the agent database, but into a higher layer in the hierarchy. There should be a module that assigns preferences to agents according to evaluated demographic data.

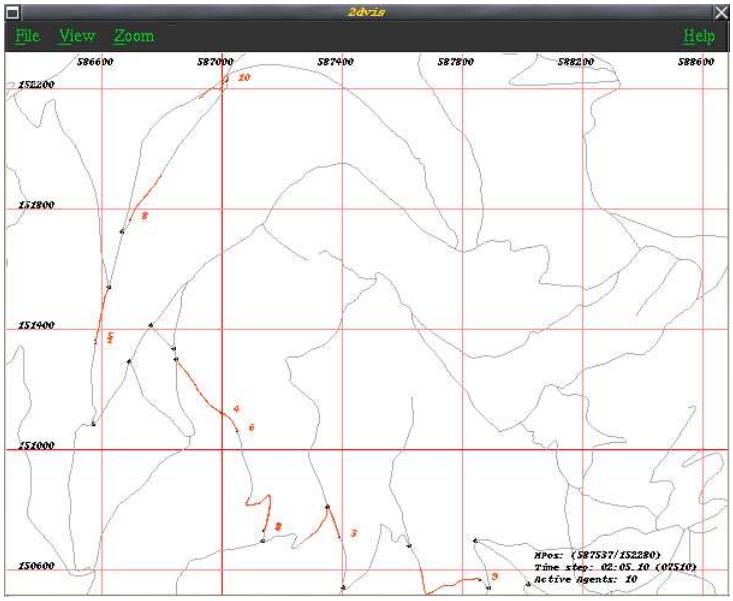
```
if (($agent % 2) == 0) {
  $weights[$agent] = 'weight_pedpressure="1"
                      weight_view="-0.2"
                      weight_forest="1"
                      weight_rain="0.5"
                      weight_sun="-0.5" ';
  $agenttype='0';
} else {
  $weights[$agent] = 'weight_pedpressure="1"
                      weight_view="1"
                      weight_forest="-0.2"
                      weight_rain="0.5"
                      weight_sun="-0.5" ';
  $agenttype='1';
}
```

This string is given to the route generator in each request. In the beginning, the route generator does not know where the forests are and where the view is great (see Figure 9.4a). This is information it has to learn from the events, which are sent by the mobility simulation. “forest” and “view” belong to the category of Indirect Spatial Events (See Section 4.4.2).

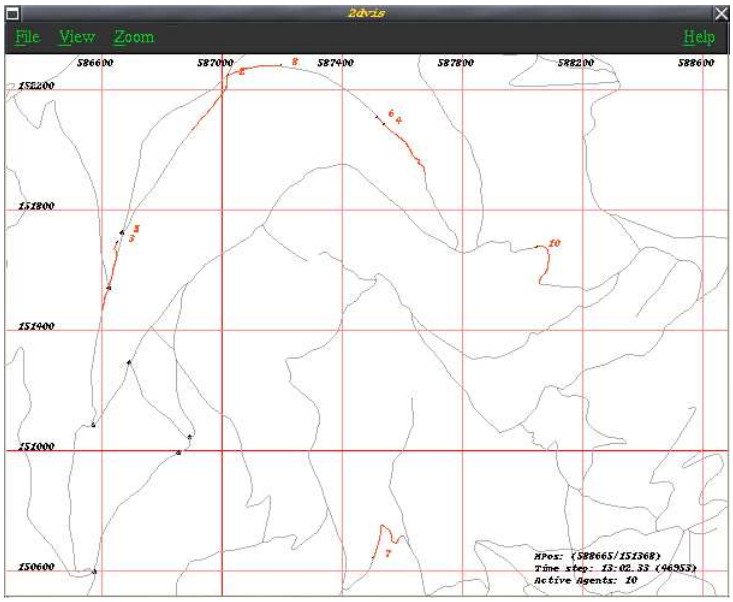
The mobility simulation sends lots of events, since every agents reports what it perceives on almost every step. An example for a “forest” event, sent when an agent has recognized that it is inside a forest, is:

```
<event type="forest" id="4" time="08:05.12" x="586961" y="150324" />
```

For every event, the route generator searches the appropriate link. The occurrence of each event for a link is just accumulated. Every time an agent asks for a new route, the router knows the area better and is able to give a route that fits better to the agent’s preferences.



(a) First run, all the agents hike on the same path, since this seems to be the "best" path for everyone.



(b) After 5 iterations, they have learned to spread according their preferences.

Figure 9.4: Same destination, different kind of hikers. In the 1st iteration (a), every agent walks on the same path, although they have different preferences. The route generator did not get a chance to build its mental map yet. After the 5th iteration, the agents have figured out which side of the hill they prefer (b) and spread out accordingly. The route generator has built its mental map out of forest and view events.

This is visible quickly: after the first agent has reached the top of the mountain, it asks for a new route in order to hike down again. Already, the route generator has learned how the route the agent took on its way up looks like, and is able to give a different route that fits better. Already on the first hike downhill, it is possible to distinguish the two groups, although the routes do overlap quite a bit.

After five iterations, the separation is visible well. In Figure 9.4b, the agents with even IDs belong to one group, the agents with odd IDs to the other.

Results and Discussion

The agents are able to learn very quickly. This is because the route generator accumulates the events received from any agent. Every agent has access to the knowledge of the other agents. The separation starts to be visible right after the first iteration.

If the departure time of the agent is different (say one agent leaves the hotel at 8:00, one at 8:15 and so on), the agents leaving later *already* chose a different route, since the route generator already has access to the events sent by the agent before. This behavior is unrealistic, the knowledge travels too fast from one agent to another.

9.2.2 Two Destinations to Chose From

Feature to Demonstrate

One of the features added to the route generator is its ability to send back the travel time and the travel cost. Using this values, the agent database is able to ask for a route to different destinations, and compare the travel time and cost of these. It can then decide which destination is better for an agent, given its preferences.

How Can This Be Shown?

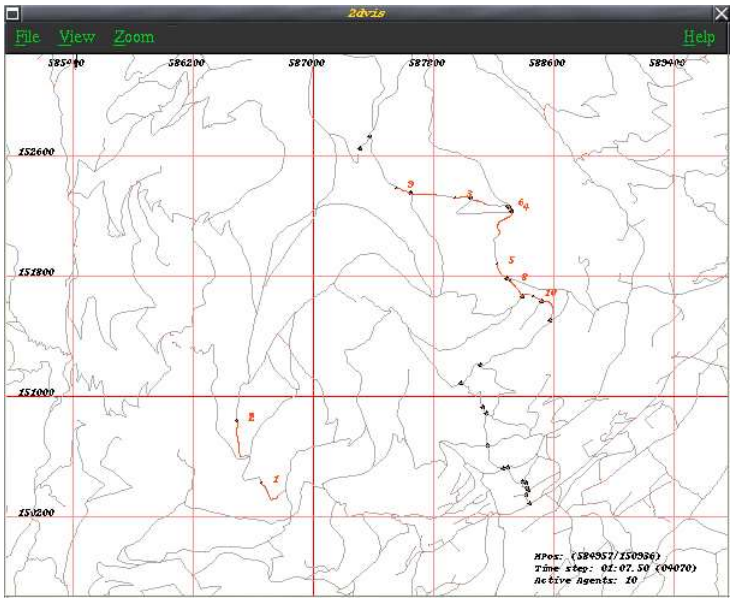
The experiment setup is the same as in the experiment before, however here, the agent database needs to compare the route answers. It makes two route route requests (one to Rellerli, one to Horneggli) and selects the destination with the higher utility (lower cost per time spent hiking).

The same two groups as in Section 9.2.1 are used again for this experiment. Hikers of the first group like forest, the others like scenaric view.

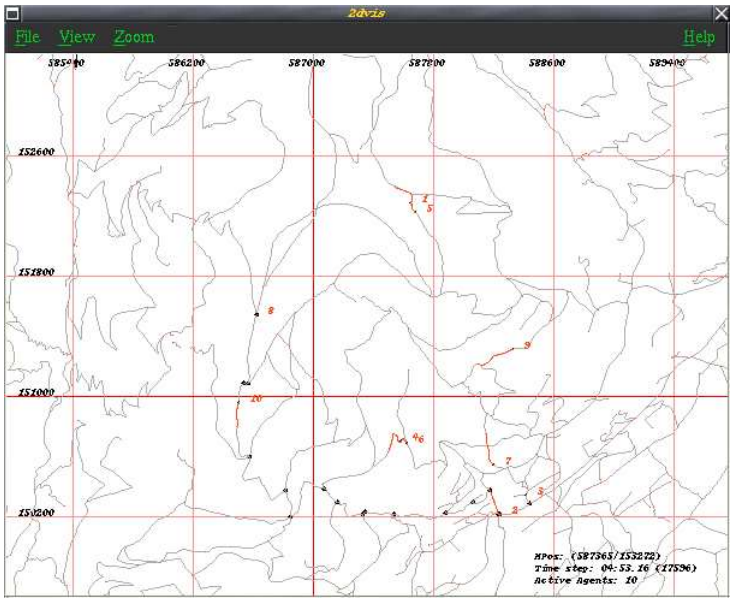
Description of the Scenario

As in the last experiment, we need the mobility simulation, the router, and the agent database. In order that the agent database is able to select out of two destinations, this functionality has to be added (See Figure 8.1 for the flow chart if that implementation of the agent database).

First, the agent database has to ask for the routes for the two destinations:



(a) First iteration



(b) After 5 iterations

Figure 9.5: Two different activities to chose from, two groups of hikers. In the beginning, they chose randomly (a). After the 5th iteration: the figured out which side of the hill is nicer and chose the destination that fits better their preferences.

```
print $event "<event type=\"route_request\" from=\"$linkhotel\"
                     to=\"$activities[0]\" agent=\"$agent\"
                     time=\"0\" $weights[$agent]
                     request_id=\"routeA\"/>\n";

print $event "<event type=\"route_request\" from=\"$linkhotel\"
                     to=\"$activities[1]\" agent=\"$agent\"
                     time=\"0\" $weights[$agent]
                     request_id=\"routeB\"/>\n";
```

The agent database uses the route generator's ability to just send back those parts of a request that it does not understand. Here, a `request_id` is assigned to each request. In the route answer, the agent database is able to distinguish the two answers and assign the values to the correct destination:

```
if ($requestid eq 'routeA') {
  $costA{$agent} = $ecost/$etime;
}
if ($requestid eq 'routeB') {
  $costB{$agent} = $ecost/$etime;
}
```

As soon as both route answers are available, the agent database compares the utilization:

```
if ($costA{$agent} < $costB{$agent}) {
  print "Agent $agent: Route A is better\n";
  print $event "<event type=\"route_request\"
                       from=\"$linkhotel\" to=\"$activities[0]\"
                       agent=\"$agent\" time=\"0\" $weights[$agent]
                       request_id=\"final\"/>\n";
} else {
  print "Agent $agent: Route B is better\n";
  print $event "<event type=\"route_request\"
                       from=\"$linkhotel\" to=\"$activities[1]\"
                       agent=\"$agent\" time=\"0\" $weights[$agent]
                       request_id=\"final\"/>\n";
}
```

In the current implementation, the agent database sends again a route request, but only for the better destination. This is because it was designed to be state-less, i.e. there should be no information related to an agent stored in the agent database. However, requesting the same route twice is generally not a good idea either. In this case, since the agent database has to store the time and cost for each of the initial routes anyway, it could store the route as well.

Here, only route answers that carry the `request_id` "final" are sent to the mobility simulation for execution.

Results and Discussion

In the first iteration, the route generator has no other information than the estimated link travel times (as in the experiment before, see Figure 9.5a). Again, it builds its mental map as soon as the first hiker starts to send events.

The travel time and cost in the routes given back do not differ, cost has the same value as time. This

yields in a cost/time-factor of 1 for both destinations. Due to the small randomization in the route generator, the factor is not exactly 1, and it looks like the agent database would chose a destination at random.

After 5 iterations, travel time and cost differ, the route generator has learned additional attributes. The agent database is able to chose the best destination, which is visible by the separation of the two groups (Figure 9.5b).

The learning and the application of the knowledge works fine. However again, as in the last experiment, the propagation of the knowledge is too fast, since the agents share their knowledge.

It would be better to give up the principle of a stateless agent database in order to increase the speed of computation. The route answers should be cached by the agent database.

9.2.3 The "Exploraition Scenario"

Feature to Demonstrate

One of the main reason for a modular system is that it is possible to replace any module. In an early stage of development, only the mobility simulation and the visualizer existed. The agent database and the route generator module from the traffic project (see Raney, 2005) were taken and placed into the pedestrian simulation framework – to check if the mobility simulation was able to run longer scenarios and that the events interface was working.

How Can This Be Shown?

The input to the mobility simulation is *plans*, sent over the braincom channel. The traffic agent database is able to generate plan files out of activity files. A simple acitivity pair (i.e. leave hotel, hike to mountain peak) was provided.

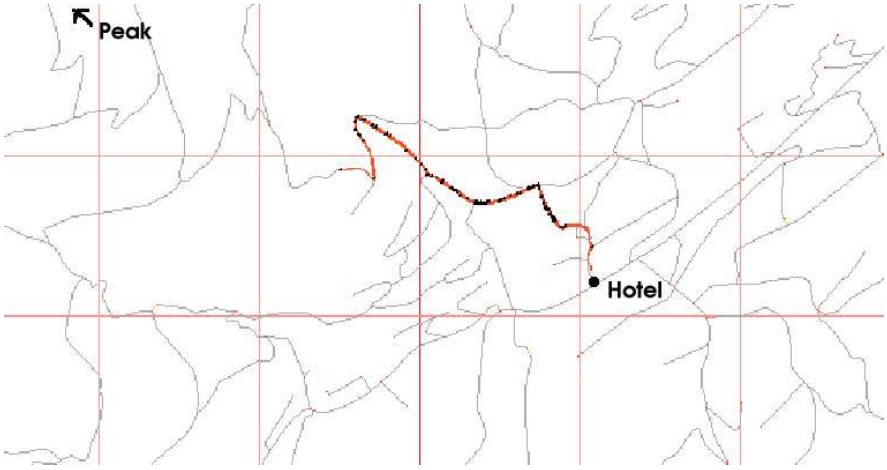
The output of the mobility simulation is sent as *events* to the events channel. These events were fed into the route generator, which was polled by the agent database for a route. Using the information from the last iteration, the route generator is able to propose a reasonable route for any agent.

The agent database from the traffic project lets the agents explore the network. They might be able to find a better route if they know the network better.

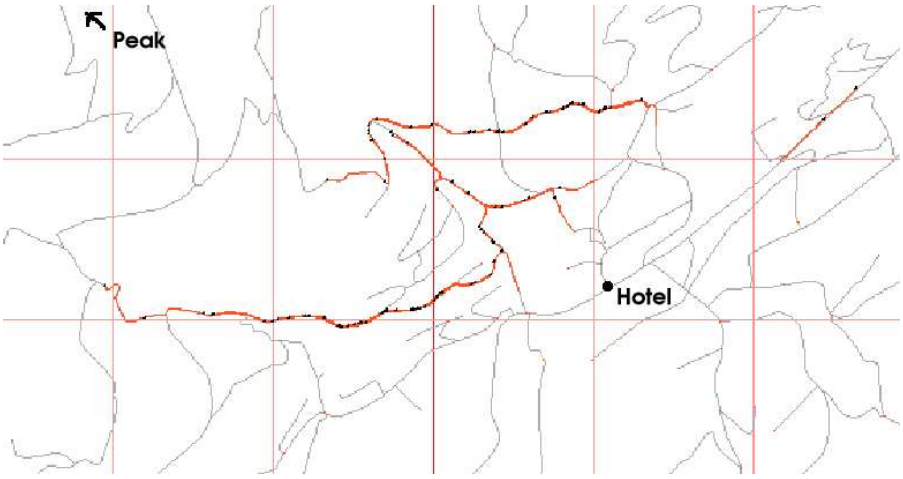
This can be used in the hiking scenario as well: the hikers spread out to explore the hiking trails.

Description of the Scenario

The route generator from the traffic simulation project proposes routes based on link travel time only. It uses the events sent by the mobility simulation to update its representation of the links by setting the link travel time of every link to the actual, observed value. It listens to `enterlink` and `exilink` events, which, if they are paired correctly, can be used to calculate the time it took for an agent to travel through a certain link.



(a) First run, all the agents hike on the same path, since this seems to be the "best" path for everyone.



(b) After 50 iterations, they have learned to spread out to avoid each other.

Figure 9.6: It is assumed that all agents leave in the morning from the hotel and hike to the same mountain peak. They start to explore the area in hope to find a faster route.

Before the first simulation run, all the link travel times in the route generator were set to an *very high value*, which says that the agents are able to travel very fast on all the links. The routes proposed by such an route generator represent still the shortest route from hotel to mountain peak. In the first iteration, all the agents followed this link (Fig. 9.6a).

However, they all reported their actual travel time to the route generator, which adapted its internal representation accordingly. The link that was the fastest is now much suddenly much slower, since the

unrealistically short link travel times were replaced by the actual values.

As a result, the next route that is calculated for the same activity pair (i.e. still hotel to mountain peak) uses all the links near to those used in the first iteration. The agents travel on this new route only to learn that also here the travel time was completely underestimated. This forces them to explore a new route in every iteration, until the proposed detour is so much longer, that it becomes more efficient to use the shortest route again.

However, in order to avoid oscillations in the case of a bottleneck in the traffic scenario, the agent database assigns new routes only to 10% of the travellers. This results in 90% of the hikers that stay on the first route in the second iteration.

After the second iterations, 10% of the travellers ask for a completely new route, another 10% use any of the routes they have experienced before, but not the last one, and the rest (80%) stays on the same route until the next iteration.

After many iterations the agents are evenly distributed among the routes that seem reasonable (Fig. 9.6a). If a route is reasonable is defined by the ratio between the actual travel time and the value that was used as the very high travel speed. If a detour of 10 times the length of the fastest route should be considered, the original value for the travel speed has to be ten times faster than the mobility simulation actually reports.

If this experiment would run forever, the hikers eventually will all switch back to the original fastest route. After all feasible route alternatives have been explored, the agents learn that the fastest is the one they used in the first run. However, since the agent database assigns a new route to only 20% of the agents, this takes many iterations.

Results and Discussion

A small proof-of-principle run is documented in Figure9.6. It is assumed that all agents leave in the morning from the hotel and hike to the same mountain peak. They however want to avoid each other because they want to hike in solitude. Fig. 9.6a shows the first run, where no hiker knew about the other hikers' intentions. Fig. 9.6b shows the situation after 50 iterations, where hikers have learned to spread out and avoid each other.

It can be said that the framework is able to produce the proposed separation of the hikers.

However, since the decisions of this agent database are based on link travel times only, it was not used further. A simple agent database for the special purpose of hiking agents was written later, and is used for the recent test scenarios.

Chapter 10

General Use of the Framework

10.1 Integration of 3rd Party Modules into the Existing Framework

One of the advantages of the presented framework is that it is possible to attach a 3rd party module into the system. As an example the connection of the view analysis module (by Cavens, in preparation, see also Chapter 2) is presented.

The events that are generated by the view analyzer module are then processed by another module: the mental map module (see Section 10.1.2). This module proposes new activity locations, which are then connected by the route generator.

Finally, as an extreme example of its flexibility, the framework is transformed into a financial application that supports stock market traders (Section 10.2).

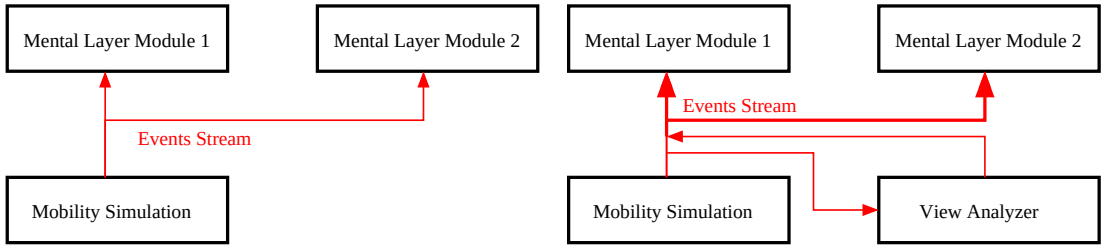
10.1.1 Adding Physical System Information into the Events Stream: Visual Analyzer

The mobility simulation sends the perception of an agent to the modules in the mental layer via the events stream (see Figure 10.1a). These events cover what the physical model in the mobility simulation is able to represent: links, nodes and other agents. However, a hiking simulation consists of more than these three items.

It would be possible to implement the representation of, say, trees into the mobility simulation directly. This would make sense if those trees would affect a hiker directly as well. Since a tree is relatively small, this is not the case. A tree mainly affects the hiker because it is nice to look at, or because it blocks the view to the rest of the landscape. However, concepts like that belong into the mental layer.

A separate module, the view analyzer, takes care of this. In order to calculate what an agent is able to see, it needs to know the position and walking direction of an agent. This information can be derived from a position event, sent by the mobility simulation via the events stream.

```
<event type="position" id="42" time="100" x="588162" y="150664"/>
<event type="position" id="42" time="101" x="588161" y="150663"/>
```



(a) Only the mobility simulation sends events

(b) The original events are extended by events sent by the view analyzer module

Figure 10.1: The events sent by the mobility simulation are received by the mental layer modules. The view analyzer module listens to these events as well, and sends again its events that are based on the original events back into the same events stream.

The view analyzer module listens to the events stream, and calculates what an agent is able to see. It then encodes this new information again in the same way as the other events:

```
<event type="vis_analysis"
       id="42" time="100" x="588162" y="150664"
       sky="10%" forest="80%" ground="10%"
       viewDirection="45" viewDistance="124"
/>
```

In this example, the agent's field of view consists of 10% sky, 10% ground, and 80% forest. The hiker, at the moment this event was sent, was looking in direction "45" and on average, the items seen are 124m away.

This new event is now sent back into the events stream (Figure 10.1b). For the modules in the mental layer, the events stream suddenly carries more events than before. The view analyzer module *added* information to an existing stream.

The modules of the mental layer can now make use of this new information – or, if they do not understand its meaning, discard it.

It would be possible to model a hiker that does not like to see other agents at all by simply attaching a view analyzer and let it send a *pedpressure* event every time it sees another agent. This event is already known to the route generator, which also knows how to connect this to the preferences of a hiker.

Using this mechanism it is even possible to attach a new module (like the view analyzer) to an *running system*. As long as there is no other module that understands the new information, this does not make sense. However, also another module in the mental layer can be added at the same time that uses the new information.

Also the visualizer can display the position from where these events were sent without modification. This can be used to gather further information about a running system, like aggregations that are not done in the mobility simulation.

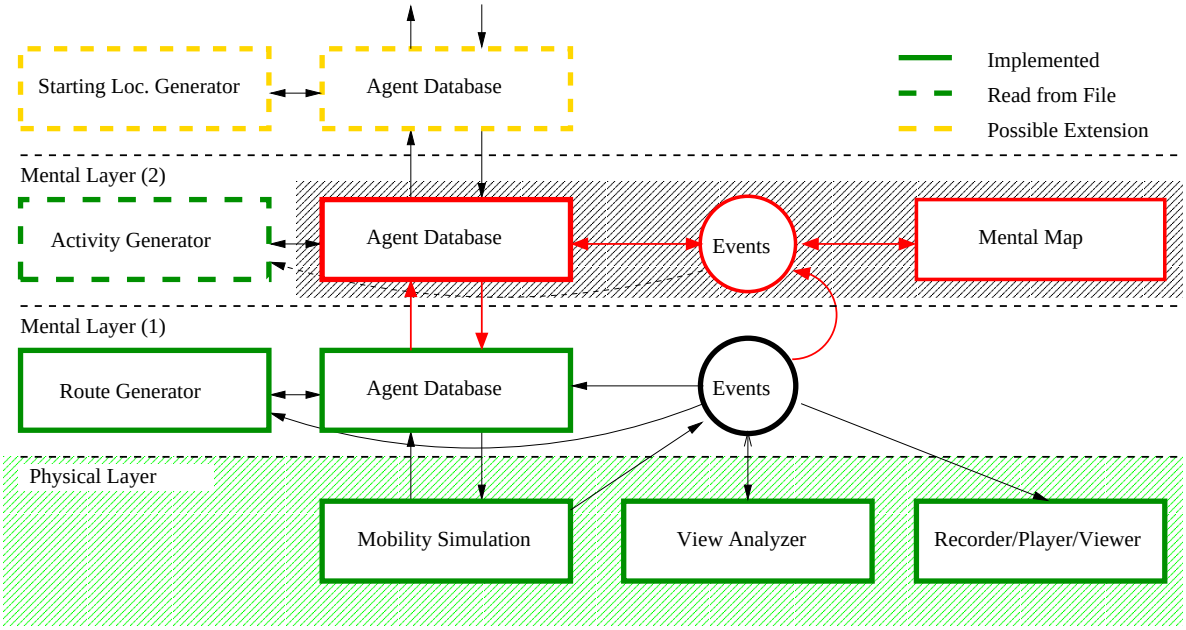


Figure 10.2: The program by D. Kistler includes not only the mental map module, but also functionality of the agent database in the second mental layer. The hatched (black) area was implemented in a single executable. The communication channels marked with red were new in that work.

10.1.2 Building Mental Maps for Individual Agents

An example of a module that uses the extended events fed into the events stream by the view analyzer module is the mental map module by Kistler (2004).

This module is in the second mental layer, since its purpose it to propose yet unknown destinations for activities. Activity plans usually are sent from the agent database in the second mental layer to the one in the first mental layer (see Figure 10.2).

It listens to events sent by the view analyzer. Specifically is searches for landmarks that are a potential new destination. It extracts from the events, for example, the direction and distance of forests:

```
<event type="vis_analysis"
       id="42" time="100" x="588162" y="150664"
       sky="10%" forest="80%" ground="10%"
       viewDirection="45" viewDistance="124"
/>
```

Every time it receives such an event, it adds a data point into its internal representation of the world (see Figure 10.3). This information is weighted according the percentage of forest seen (80% in the example). As soon as it believes that it has gathered enough information, it proposes a new activity location to the agent database in the second mental layer. This agent database uses this new information in the next activity plan, which is sent down to the first mental layer, where the activities are connected by feasible routes (see Figure 10.2).

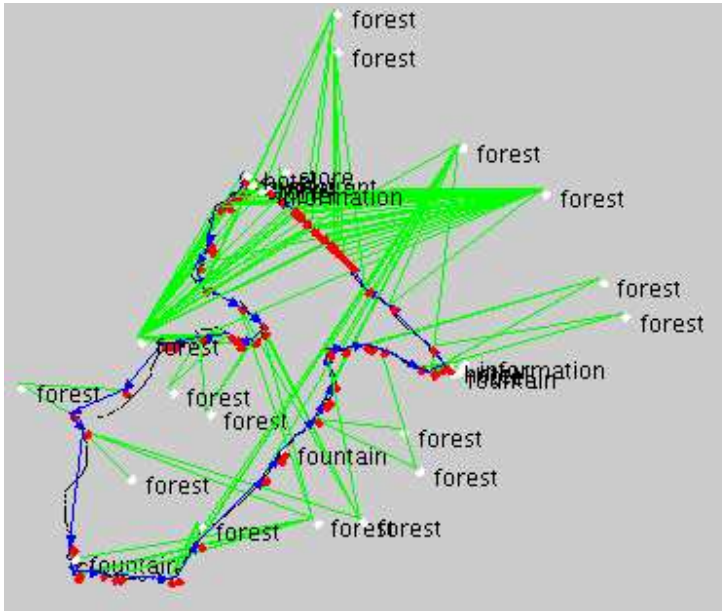


Figure 10.3: The view analyzer module reports how much forest is in the field of view of an agent. These messages include the current location of the agent (red dots) and the direction and distance where the forest is (green lines). The program by D. Kistler builds a mental map out of these information: it is able to learn the position of forests (Fig. from Kistler, 2004).

In D. Kistler's work, the part of the framework that is hatched in Figure 10.2 is implemented as one single module. It therefore also includes the agent database of the second mental layer. This does not necessariliy have to be like that, but this approch eliminates the need for an interface to propose new activity locations to the agent database.

In order that the route generator is able to calculate a path to a newly discovered activity location, it has to be located on the existing street network. It would be possible to introduce new nodes and links to the mobility simulation as well, the hikers are capable of leaving the paths if needed. However, this approach was not followed further in this or D. Kistler's work.

10.2 Using the Same Framework for a Completely Different Application: Value Added Services for Day Traders

In order to show that the framework presented here can be used for other purposes as just for a hiking simulation, an extreme case is presented.

In financial markets, traders do not look directly at the prices of a title in a stock market, they use several tools to display the information. It is impossible for a single trader to keep an eye on all the prices of a stock market. He needs a tool that aggregates the raw data and also points out a lucrative situation.

An example of such an infrastructure can be built using almost exactly the same framework as presented

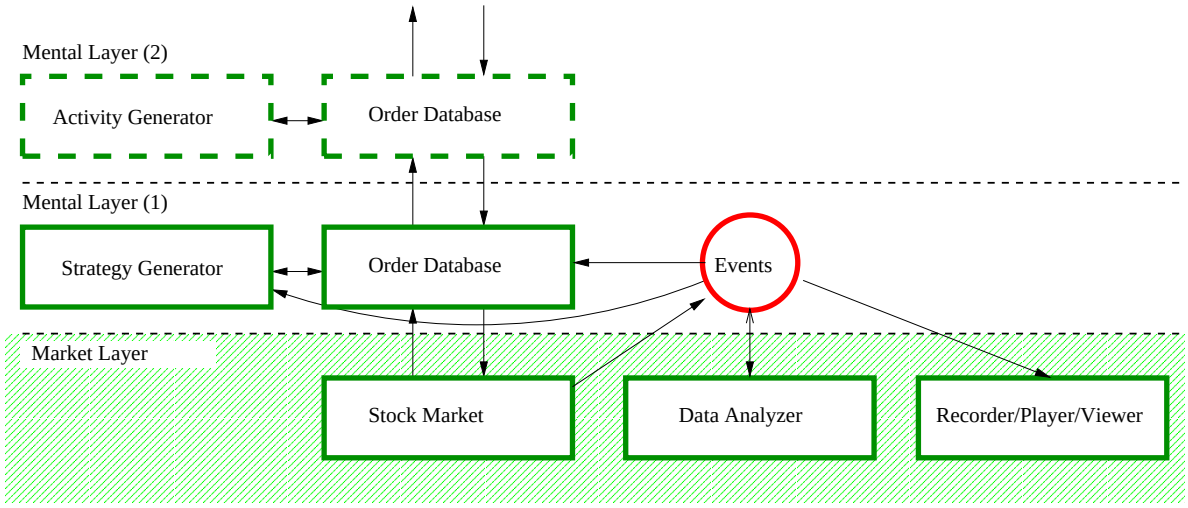


Figure 10.4: The same framework can also be used for a completely differenct application: for processing order strategies in an automated trading system.

in this work. The modules perform the same actions and are connected in the same way, but they act on different data. Here, the agents are not hikers, but *orders*. These can be simple "buy" or "sell" orders, but also "sell/buy at a certain price" or to cancel a previous order.

In the following sections, the mapping of the modules known so far to the modules used in a fictitious [1] application for traders is presented.

Mobility Simulation: The Live Stock Market

The mobility simulation can be replaced by a stock market. Usually, a broker gets a data feed from the live market that announces the price and volume change of each title in the market.

```
<price type="update" exchange="XEurex" contract="X FDAX"
  contractdate="JUN04" lastprice="3796.5" lastvolume="4"
  bidprice="3796.5" bidvolume="2" askprice="3797" askvolume="9"
/>
<price type="update" exchange="XEurex" contract="X FDAX"
  contractdate="SEP04" lastprice="3816" lastvolume="1" bidprice="3816"
  bidvolume="2" askprice="3817" askvolume="9"
/>
<price type="update" exchange="XEurex" contract="X FDAX"
  contractdate="DEC04" lastprice="3843" lastvolume="1" bidprice="3836.5"
  bidvolume="21" askprice="3838" askvolume="1"
/>
```

The provider of this data feed also offers an API to interact with the market. Using this API, a program is written. As in the mobility simulation, events (price or volume changes, but also status updates,

[1] There exist prototypes for most of the modules presented here. They were also connected to a life data stream of a stock market. Although they did not execute "real" orders, it was possible to see that the fbw of data did indeed work.

e.g. opening or closing of a market) can be sent to an events stream.

Visualizers: Visualizers

The data stream from the stock market can be picked up by a visualizer module. The output, of course, is not map based, but graphical as well in order to support the traders.

Recorder/Player: Recorder/Player

The events stream is recorded using the recorder module. Using this recorded stream, it is possible to look at the history of a market in detail later. This is useful in order to see why exactly the system behaved like it did.

Using the player module, which plays back the data as it would come directly from the market, is is possible to test a newly developed module on historical data. However, there is no feedback from submitted orders. But still, this can be very useful to *train* a module.

View Analyzer: Value-Add Module

Into this data stream, additional modules can be attached. These modules, like the view analyzer module, take the raw data and perform calculations on them. The result is sent back into the data stream for other modules.

A simple example is the calculation of a *floating average*. The module memorizes the last N price updates of a given title, and calculates a floating average out of them. In every timestep, this value is sent back into the data stream. Using different values for N, there can be more than one average for a title. These values are used in technical analysis. If this floating averages are displayed together with the original price data in the visualizer, a human trader can use this as base for his decisions. However, also modules in the mental layer can listen to this data.

Agent Database: Order Database

From above, the order database receives an order to execute. It passes this order down to the market, and waits for a feedback. This order can be filled (executed) in the market or, if it is still there after a certain time, canceled by the agent database. Again, this cancellation may work or not (it fails e.g. if someone just has filled the order).

The order database listens to the events from the market. It uses some simple criteria about when to cancel an order or place a new order. For example, one strategy might be to bid EUR 1.- below the current price. If the price goes down, the order is filled and a new order has to be placed. If the price goes up, the order has to be canceled and again a new order is placed.

There is also a module that listens to the events stream and searches for a good combination of prices actively. For example, there might be a different price in two markets for the same stock. It then might be lucrative to buy at one market and sell immediately at the other (arbitrage). This price differences

exist for a moment only, so the module has to submit two orders to the agent database quickly. However, if the price has changed in the meantime, one or both orders may fail.

The agent database can assign a score (simple: the earned money) to each order pair. If in future a similar order is proposed, it can deny it (or alert a human trader).

Route Generator: Strategy Generator

Of course there are no routes to calculate in this example. However, if in the example above one order was executed and the other failed, there is a certain number of titles that were bought but there is no further usage. The agent database can ask the strategy generator module for an order that would make the best use of this asset. This can be a simple "sell". There might exist more sophisticated solutions that include derivatives of the title, like buy or sell *options*.

Upper Mental Layer: Preferences Given by the Customer

The agent database receives as input from above the preferences of a user. Just "make more money" is not sufficient, there is also the amount of risk an investigator is willing to accept that influences the selected strategy.

10.3 Summary

The ability to connect new modules directly into the existing framework was shown. The view analyzer introduces additional value into the events stream without the need to modify the system at all.

This events can be used by a mental map module, which builds an internal representaion of the world based on what a hiker sees. It can propose new activity locations to the system. This module as well can be inserted without modification of the system.

An example for the usage of the framework outside hiking simulations was presented. Using the same framework, and using even some modules without modification (TCP Multicaster, Recorder, Player), a financial application was described.

These tree cases show the flexibility of the proposed framework. It is possible to attach new modules in the physical layer, in the mental layers. And it is not bound to pedesrian simulations at all.

Chapter 11

Outlook, Summary and Conclusions

Visual quality is hard to quantify and therefore hard to integrate into a computer model and has therefore largely been ignored in planning models and simulation. The simulation system presented in this work goes in this direction.

It tries to integrate aesthetic qualities with other factors such as availability of recreational opportunities, congestion and service levels using an agent based approach.

In this chapter, we look at further improvements (Section 11.1) and summarize what has been achieved so far (Section 11.2).

11.1 Outlook

The framework and the modules presented in this work seem to be flexible and useful. The simulation of simple real-world examples is feasible already. For larger scenarios, more work is required in order to calibrate the model.

There is also room for future improvements in the modules. Some of the ideas are presented in this section.

The pedestrian mobility simulation presented in Chapter 3 runs on a single CPU. Quinn et al. (2003) introduced an implementation that divides the simulated area in rectangular blocks. This is useful in order to simulate an area with more or less equal density of pedestrians. In our scenario, the hikers tend to stay on the trails. Between the trails are areas where no hiker will ever walk on (like dense forest, lakes etc.).

It would be possible to parallelize the pedestrian simulation using almost the same technique as used for the traffic mobility simulation (Cetin, 2005). There, the street network itself is used to find an optimal partition. The border between two processors is right after an intersection. In the traffic simulation, one CPU has to inform the adjacent CPUs about full links, that no car can enter this link. For the pedestrian simulation, a CPU would have to send the position of each pedestrian on that link in order that the adjacent CPUs are able to compute the forces between agents. Assuming the agents are capable to “feel” another agent that is at the other end of an intersection. However, it is not clear if this graph-

based approach would be better than the existing block-based approach.

The route generator uses the same graph for every agent (see Chapter 4). This leads to some artifacts in the learning mechanism, as seen in Sections 9.2.1 and 9.2.2, where the knowledge was propagated too fast from one agent to another.

In (Gloor, 2001) an approach to build a route generator that uses a separate graph for each agent was presented. Since it is not possible to store a complete graph for every agent, some kind of compression or sophisticated storage algorithm is needed. No hiker ever visits the whole graph, so for each hiker, only a small part of the graph is really filled with collected attributed (from events). It should be possible to build an representation, where a route generator uses one full copy of the graph network, and stores the additional information learned for each hiker in additional tables. Using techniques like *hash tables*, the lookup during the computation of the shortest path should be reasonable fast.

Unger (2002) presented a method where agents can explore parts the network that is completely unknown. She also introduced methods to encode the *a priori* knowledge such as guideposts and traffic signs.

An attempt to include such information was made by (Kistler, 2004, see Chapter 10). He used the information gathered by the Visibility Analyzer Module (Cavens, in preparation) to build a new mental map, and this mental map to propose new possible activities. Using a mechanism like that, it would be possible to react to street signs etc. as well.

However, these methods work at the moment for single hikers only, because they consume a lot of CPU time. Another interesting aspect is the *communication* between agents. Agents could share their knowledge if they meet each other, e.g. in a hotel or restaurant. However, for this, a simulation with multiple agents is needed.

In further sensitivity studies, it should be investigated how virtual (and real) hikers react to visual aspects, like a golf course, a power plant, or to reduction of forest etc. In order to do this, additional events and handlers in the agent database or another module needs to be implemented. A flexible approach to detect visual aspects is the Visibility Analyzer by Cavens (in preparation).

The agent databases used here are simple replacements for the one developed by Raney (2005). It would be interesting to combine these approaches, i.e. to make Raney's agent database communicate over the network, and to implement the features added here, like reaction to unpredicted events.

Also an agent database for the upper layers in Figure 2.1 would be needed. At the moment, it is unclear which form of the framework presented in Chapter 2 is the better approach.

If there are more modules that need to communicate with each other, also the need for fast communication methods increases. In the current implementation, only one XML message is sent per data packet. Throughput could be increased by sending multiple messages per packet (see e.g. Cetin, 2005).

Also usually in modern computer clusters, a faster network infrastructure exists. This work is based completely on the TCP/IP protocol stack. However, is should be possible to adopt this to other network hardware, e.g. to Myrinet (Myricom www page, accessed 2005) and InfiniBand (InfiniBand Trade Association www page, accessed 2005).

In order to make the simulation system more usable, a simpler way to run simulations is required. At the moment, either every modules is started manually, or a script is used that controls the startup. However,

using this script an user loses much of the control. A graphical user interface would be better, where a user could select input files and scenarios from a menu.

Also the 2-dimensional visualizer needs a better user interface. It should be possible to integrate the user interface used by the player module at the moment. If a single user interface would exist for the whole system, also the 3-dimensional visualizer could be integrated (using e.g. a plug-in).

Finally, in order to create formidable videos for presentation, the non-real-time visualizer can be used. For this, a detailed 3-dimensional representation of the simulated areas was created, like the one created before by Cavens for the 3-dimensional visualizer. However, the representation of each agent could be much more detailed, since the computation time spent in the rendering does not matter that much anymore. A animation of a walking agent as used by (e.g. Ponder et al., 2003) could be implemented into the non-real-time visualizers as well.

11.2 Summary and Conclusions

In this work, the known modular framework of traffic simulations was developed further. Using the same modular structure, but a new way for coupling these modules, also new functionality was developed.

The modules are connected no longer by files, but by network messages. Of course this also brings disadvantages, as the system becomes more complex, but they are outweighed by advantages: the message-based approach allows real-time interaction between the modules. This allows the agent to react to a new situation immediately.

It is also possible to attach new modules directly into the framework: It turned out that modules like the view-analyzer can be entered into the events stream without major modifications of the framework design.

This allows to use modules that process the raw data from the simulation and feds value-added data back into the events stream. The resulting events stream is then a combination of events generated by the mobility simulation, and secondary events generated by other modules. In this way, it is possible to have several modules jointly compute different aspects of the events stream, which are then picked up by the mental modules.

An extension to a well-known pedestrian dynamics model was implemented. Using sophisticated techniques which adapt the model to the rather special circumstances of hikers in the alps, it is now possible to simulate an area of $15km^2$ and more on a single CPU.

This new simulation also works close to obstacles, as are found e.g. close to buildings. Also the simulation of the inside of buildings is possible, which allows the usage of the same framework and mental layer modules for e.g. evacuation simulation.

The route generator module that was presented in this work is based on previous work. It was modified to fit into the new network-based framework. In a second step, a generalized cost function was introduced. The modification of the message interface, which was needed anyway to support the use of the hiker's preferences, allows now to return the estimated travel time and travel cost in the route answer

The route generator is now more powerful compared to his ancestor. It is, however, also slower. It uses

more memory as well, the internal representation is more voluminous because of the multiple attributes it has to store.

The framework proved to be useful, since it fulfilled the hope that it would allow intra-day replanning. It turned out that this modularization and the message based approach yield much flexibility.

In terms of simulating real-world scenarios, the framework and the single modules are a step forward. It seems that it is possible to simulate simple examples already.

Appendix A

Algorithms

A.1 Dijkstra's Algorithm

Dijkstra (1959) presents an exact algorithm for finding the shortest path between two nodes in a graph. It functions by constructing a shortest-path tree from the initial node s to every other node in the graph (if not aborted before), which also leads to the destination node d.

Every path in a (weighted) graph has an associated path weight, the value of which is the sum of the weights of that path's links.

The algorithm starts by assigning a travel time T_s of 0 to the start node s (we are already there) and *infinite* to all other nodes (we have no idea how to get there). The start node s is also added to the shortest-path tree (SPT), which at this moment, contains s only. SPT contains all nodes to which we know already the travel time.

Also there is the group of leaf nodes of nodes already in the SPT. In the initialization, the start node s is put into this group.

Loop:

The leaf node a with the lowest travel time to s is selected, and removed from the group of leaf nodes. This node a is now *expanded*, which means that the travel time T_n to all nodes n directly attached to a is calculated ($T_n = T_a + \Delta T_{a \to n}$). If a node n is already a leaf node, the new travel time T_n is only assigned if it is smaller than the T_n already stored in n. Also, into each node that is updated like this, a *pointer* to the current node a is saved. All these nodes n are put into the group of leaf nodes. The expanded node a is now automatically in the SPT because of the pointer.

The loop ends as soon as the destination node d is selected as the leaf node with the lowest travel time T_n, since then we know the shortest path to the start node s.

In order to determine the shortest path from node s to node d, the SPT has to be traversed *backwards* from d. Starting at d, the pointers to the previous node (`prev`) are followed, until the start node s is reached.

```
// initialize start node:
```

```
startNode->set_ArrTime( 0 ) ;
pending.insert( make_pair( 0, startNode ) ) ;

// Dijkstra loop proper:
while( pending.size() > 0 ) {
  RouteNode* theNode = pending.begin()->second ;
  pending.erase( pending.begin() ) ;
  if ( !(theNode->isDone()) ) {
    theNode->set_IsDone() ;
    if ( theNode!=endNode ) {
      theNode->expand( pending ) ;
    } else {
      return 0 ;
    }
  }
}
void RouteNode::expand ( NodeList& pending ) {

  Time now = arrTime_ ;
  for ( VLinks::iterator ll=outLinks_.begin();
        ll!=outLinks_.end(); ll++ ) {
    RouteNode* nextNode = (*ll)->toNode() ;

    // tTime not dependent on time (now)
    Time linkTTime = (*ll)->tTime() * ( 1.+ 0.001*myRand() );

    if ( ( linkTTime != linkTTime ) || ( linkTTime <= 0 )
    || ( linkTTime >= 999999 ) ) {
      cerr << "node " << this->id() << ": linkTTime for link "
           << (*ll)->id() << " is " << linkTTime
           << endl;
    }
    assert( linkTTime > 0 ) ; assert( linkTTime < 999999 ) ;

    Time nextTime = now + linkTTime ;
    if ( nextTime < nextNode->arrTime() ) {
      nextNode->set_ArrTime( nextTime ) ;
      assert( !(nextNode->isDone()) ) ;
      nextNode->set_Prev( this ) ;

      pending.insert( make_pair( nextTime, nextNode ) ) ;
    }
  }
}
```

A.2 Time Dependent Dijkstra

In order to keep track of time through Dijkstra's algorithm, the start time T_s is set to the time the agent starts traveling (e.g. 15:20 instead of 0).

Each time the travel time (weight) from a node a to a node n ($\Delta T_{a \to n}$) is used, it is looked up in a table according the absolute time T_n.

```
// initialize start node:
startNode->set_ArrTime( startTime ) ;
```

```
pending.insert( make_pair( startTime, startNode ) ) ;

// Dijkstra loop proper:
while( pending.size() > 0 ) {
  RouteNode* theNode = pending.begin()->second ;
  pending.erase( pending.begin() ) ;
  if ( !(theNode->isDone()) ) {
    theNode->set_IsDone() ;
    if ( theNode!=endNode ) {
      theNode->expand( pending ) ;
    } else {
      return 0 ;
    }
  }
}
void RouteNode::expand ( NodeList& pending ) {

  Time now = arrTime_ ;
  for ( VLinks::iterator ll=outLinks_.begin();
        ll!=outLinks_.end(); ll++ ) {
    RouteNode* nextNode = (*ll)->toNode() ;

    // tTime dependent on time (now)
    Time linkTTime = (*ll)->tTime( now ) * ( 1.+ 0.001*myRand() );

    if ( ( linkTTime != linkTTime ) || ( linkTTime <= 0 )
    || ( linkTTime >= 999999 ) ) {
      cerr << "node " << this->id() << ": linkTTime for link "
           << (*ll)->id() << " at " << now << " is " << linkTTime
           << endl;
    }
    assert( linkTTime > 0 ) ; assert( linkTTime < 999999 ) ;

    Time nextTime = now + linkTTime ;
    if ( nextTime < nextNode->arrTime() ) {
      nextNode->set_ArrTime( nextTime ) ;
      assert( !(nextNode->isDone()) ) ;
      nextNode->set_Prev( this ) ;

      pending.insert( make_pair( nextTime, nextNode ) ) ;
    }
  }
}
```

A.3 Time Dependent Dijkstra Using Generalized Costs

As with the time dependent algorithm, in order to keep track of time, the start time T_s is set to the time the agent starts traveling. However, additionally, the travel cost is stored for each node. This cost is set to 0 for the start node s and *infinite* to all other nodes.

The loop changes as follow:

The leaf node a with the lowest travel *cost* (instead of time) to s is selected. This node a is now *expanded*, which means that the travel cost C_n as well as the travel time T_n to all nodes n directly attached to a are calculated ($T_n = T_a + \Delta T_{a \to n}$, $C_n = C_a + \Delta C_{a \to n}$). If a node n is already a leaf

node, the new travel cost C_n and travel time T_n are only assigned if the travel cost C_n it is smaller than the C_n already stores in n. All these nodes n are put into the group of leaf nodes. The expanded node a is now put into the SPT.

Each time the travel cost (weight) or the travel time from a node a to a node n ($\Delta C_{a\to n}$, $\Delta T_{a\to n}$) is used, it is looked up in a table according the absolute *time* T_n.

In short, for calculating the next node to expand, the travel cost C_n is used, for selecting the right bin to loop up $\Delta C_{a\to n}$ or $\Delta T_{a\to n}$, the travel time T_n is used.

```
// initialize start node:
startNode->set_ArrTime( startTime ) ;
startNode->setCost( 0 ) ;

pending.insert( make_pair( startTime, startNode ) ) ;

// Dijkstra loop proper:
while( pending.size() > 0 ) {
  RouteNode* theNode = pending.begin()->second ;
  pending.erase( pending.begin() ) ;
  if ( !(theNode->isDone()) ) {
    theNode->set_IsDone() ;
    if ( theNode!=endNode ) {
      theNode->expand( pending, weights ) ;
    } else {
      return 0 ;
    }
  }
}
void RouteNode::expand ( NodeList& pending, Weights& weights  ) {

  Time now = arrTime_ ;
  double cost = arrCost_;

  for ( VLinks::iterator ll=outLinks_.begin();
        ll!=outLinks_.end(); ll++ ) {
    RouteNode* nextNode = (*ll)->toNode() ;

    // tTime dependent on time (now)
    Time linkTTime = (*ll)->tTime( now ) * ( 1.+ 0.001*myRand() );

    if ( ( linkTTime != linkTTime ) || ( linkTTime <= 0 )
    || ( linkTTime >= 999999 ) ) {
      cerr << "node " << this->id() << ": linkTTime for link "
           << (*ll)->id() << " at " << now << " is " << linkTTime
           << endl;
    }
    assert( linkTTime > 0 ) ; assert( linkTTime < 999999 ) ;

    // linkCost dependent on:
    // time (now) and the preferences of the hiker (weights)
    double linkCost = (*ll)->getCost(now, weights );

    // keep track of time and cost
    Time nextTime = now + linkTTime ;
    double nextCost = cost + linkCost;

    // use cost as criteria
```

```
      if ( nextCost < nextNode->arrCost() ) {
        nextNode->set_ArrTime( nextTime ) ;
        nextNode->setCost( nextCost ) ;
        nextNode->set_Prev( this ) ;

        pending.insert( make_pair( nextTime, nextNode ) ) ;
      }
    }
  }
```

Appendix B

Network File

In general, the input files used in this project share the same ideology with the input files used in the traffic simulation project. However, there are some differences.

The outline of the network file looks like this:

```
<network>
   <streets>
      <nodes>
         <node/>
      </nodes>
      <links>
         <link>
             <coord/>
         </link>
      </links>
   </streets>
   <objects>
      <tree/>
   </objects>
</network>
```

Additions are the `<streets>` tag, with marks the network, and the `<objects>` tag, which marks the additional items, like building, trees etc.

B.1 Links (Streets)

Inside the `link` tag, `coord` was introduced. This is to represent the bends of the hiking trails. Streets do also have bends, however, for the traffic simulation, they can be disregarded. `coord` is like a node, but without any further meaning for modules in the mental layer (e.g. the route generator).

```
<link id="1" from="38" to="39" length="21.3"
      freespeed="3.6" class="6_Klass" name="Bahnhofstrasse">
    <coord id="1" x="585158.50" y="158000.00"/>
    <coord id="2" x="585166.80" y="157997.80"/>
    <coord id="3" x="585174.30" y="157998.10"/>
```

```
    <coord id="4" x="585179.20" y="158000.00"/>
</link>
```

In addition to `id`, `from`, `to` and `freespeed`, which is also used in the traffic project, `class` and `name` can be used. However, at the moment, this fields are read by the 2-dimensional visualizer only and do not affect the result of the simulation.

B.2 Nodes (Streets)

The notation of the nodes is the same as in the traffic network files.

```
<node id="18216" x="598951.30" y="122000.00"/>
```

B.3 Objects

The hiking simulation uses the `objects` section to store the position of trees. These are displayed by the visualizer, and it would be possible to use them in the force calculation as well. In fact, the open source pedestrian simulation (PEDSIM, Chapter 3) does this already.

```
<objects>
    <tree id="1" x="588528" y="150267" radius="3" type="leaf"/>
    <tree id="2" x="588231" y="150200" radius="3" type="leaf"/>
    <tree id="3" x="588231" y="150190" radius="4" type="leaf"/>
    <tree id="4" x="588229" y="150175" radius="3.5" type="leaf"/>
    <tree id="5" x="588227" y="150160" radius="3.3" type="leaf"/>
    <tree id="6" x="588220" y="150120" radius="4" type="conifer"/>
</objects>
```

The pedestrian simulation developed by Stucki (2003) uses additional items in the `objects` section:

```
<line type="wall">
  <coord x="38.33" y="10"/>
  <coord x="38.33" y="16.66"/>
</line>

<point id="1" x="19" y="1" type="item" name="coffee machine"/>

<polygon id="1" type="item" name="table">
  <coord id="1" x="34" y="14.66"/>
  <coord id="2" x="35" y="14.66"/>
  <coord id="3" x="35" y="16.66"/>
  <coord id="4" x="34" y="16.66"/>
</polygon>
```

There is a syntax definition for `point`, `line` and `polygon`, which should cover almost every possible object. However, additional types can be defined as needed.

Appendix C

Communication

C.1 Events Channel

The events channel uses the simplest format in this work. As presented in Chapter 7, it is a subset of XML.

As an additional example is here the output of an agent walking through a simple network:

```
<event time="25200" id="1914" link="200" from="20" type="departure" />
<event time="25200" id="1914" link="200" from="20" type="wait2link" />
<event time="25246" id="1914" link="200" from="20" type="entered link" />
<event time="25246" id="1914" link="200" from="20" type="left link" />
<event time="25279" id="1914" link="500" from="50" type="entered link" />
<event time="25279" id="1914" link="500" from="50" type="left link" />
<event time="25302" id="1914" link="800" from="80" type="entered link" />
<event time="25302" id="1914" link="800" from="80" type="left link" />
<event time="25325" id="1914" link="1100" from="110" type="entered link" />
<event time="25325" id="1914" link="1100" from="110" type="left link" />
<event time="25348" id="1914" link="1400" from="140" type="entered link" />
<event time="25348" id="1914" link="1400" from="140" type="left link" />
<event time="25394" id="1914" link="1702" from="170" type="entered link" />
<event time="25394" id="1914" link="1702" from="170" type="left link" />
<event time="25440" id="1914" link="1800" from="180" type="entered link" />
<event time="25440" id="1914" link="1800" from="180" type="left link" />
<event time="25440" id="1914" link="1800" from="180" type="arrival" />
```

The typical position event used multiple times in this work looks like this:

```
<event type="position" time="0" id="0" x="588440.0" y="150275.0"/>
```

C.2 Brain Comm Channel

The communication format of the Brain Comm Channel is more complex as the events channel, since here, the text section of various XML tags are filled with meaningful data as well. This makes it

impossible to send a plan over an events channel, since the TCP Multicast Daemon is not able to deal with such complex structures. Also the parsers presented in Chapter 7 are not able to handle this.

```
<request type="Goal" agent="333">
   <plan>
      <act actid="id420815" location_type="coord"
           type="hotel" x="588200" y="150000"
           end_time="07:00.00" location="4240"
      />
      <leg mode="ped">
           <act>4255</act>
           <act>4283</act>
           [..]
           <act>3886</act>
           <act>3593</act>
      </leg>
      <act actid="id23232323" location_type="object_id"
           type="restaurant" x="586515.369072" y="151093.534088"
           location="13" dur="360"
      />
      <leg mode="ped">
           <act>3593</act>
           <act>3886</act>
           [..]
           <act>4192</act>
           <act>4205</act>
      </leg>
      <act id="id131313" location_type="coord"
           type="hotel" end_time="34:00.0" x="588200" y="150000"
      />
   </plan>
</request>
```

At the moment, the Brain Comm Channel is used only to send data from an agent database to the mobility simulation. There is nothing sent over this channel into the other direction. Every feedback is sent through the events channel.

Appendix D

Module Usage

D.1 Communication Channel

The TCP Multicast Daemon has to be started first, before any other module. A simple call to `./tcp-mc` is sufficient.

If something went wrong and a previous instance of the daemon has crashed recently, it might be necessary to wait a few seconds before a new daemon can be started. This is because the operating system keeps network ports open for a certain time after a crash.

D.2 Mobility Simulation

The pedestrian mobility simulation is started like this:

```
./alpsim configfile.xml
```

The simulation is ready as soon as it prints the following line:

```
New grid created
```

D.3 PEDSIM

The open source pedestrian simulation is much simpler, and there is no configuration file at all. Therefore, to start PEDSIM, the following command is used:

```
./pedsim
```

PEDSIM will start immediately, since it does not read a network file. The output is sent to the TCP Multicast Daemon.

D.4 Cable-Car Simulation

The cable-car simulation does not need much configuration either. Again, the simple call to the executable is enough:

```
./ccsim
```

D.5 Route Generator

The route generator needs a configfile:

```
./messagerouter.exe config.xml
```

The configuration is identical to the route generator used in the traffic project.

The directory `network` holds the definition of the network used. This can be specified in the config file. However, note that the route generator uses the syntax for the network file used for the traffic project (i.e. without `streets` and `objects` tags).

D.6 2-dimensional Visualizer

The 2-dimensional visualizer reads a configuration file:

```
./2dvis config.xml
```

In that configuration file, also the color and name of received events can be specified:

```
<module name="alpview2d">
  <param name="event_xy_max" value="5" />

  <param name="event_xy0_name" value="pedpressure" />
  <param name="event_xy0_draw_size" value="5" />
  <param name="event_xy0_draw_cleartime" value="1000" />
  <param name="event_xy0_draw_color" value="1" />

[..]

  <param name="event_xy5_name" value="sun" />
  <param name="event_xy5_draw_size" value="5" />
  <param name="event_xy5_draw_cleartime" value="500" />
  <param name="event_xy5_draw_color" value="20" />
</module>
```

D.7 Agent Database

The agent databases presented in this work do not need a configuration file. The configuration is done directly in the source code. Therefore, depending on which agent database is used, another executable is started:

```
brainstub.jan05.singleagent.pl
brainstub.nov04.betteract.pl
brainstub.nov04.randomact.pl
```

Bibliography

Akima H. A433, interpolation and smooth curve fitting based on local procedures. *Communications of the ACM*, 15(10):914–918, 1972.

AlGadhi S., Mahmassani H., and Herman R. A speed-concentration relation for bi-directional crowd movements with strong interaction. In M. Schreckenberg and S.D. Sharma, editors, *Pedestrian and Evacation Dynamics*, Proceedings of the 1st international conference, pages 3–20. Springer, 2001.

Arbaro www page. http://arbaro.sourceforge.net/, accessed 2005.

Babin A., Florian M., James-Lefebvre L., and Spiess H. EMME/2: Interactive graphic method for road and transit planning. *Transportation Research Record*, 866:1–9, 1982.

Balmer M., Raney B., and Nagel K. Coupling activity-based demand generation to a truly agent-based traffic simulation – activity time allocation. In *Presented at EIRASS workshop on Progress in activity-based analysis*. Maastricht, NL, 2004. Also presented at STRC'04, see www.strc.ch.

Beazley D., Lomdahl P., Gronbech-Jensen N., Giles R., and Tamayo P. Parallel algorithms for short-range molecular dynamics. In D. Stauffer, editor, *Annual reviews of computational physics III*, pages 119–176. World Scientific, 1995.

Bishop I.D., Ye W.S., and Karadaglis C. Experimental approaches to perception response in virtual worlds. *Landscape and Urban Planning*, 54:119–127, 2001.

Blue V. and Adler J. Flow capacities from cellular automata modeling of proportional splits of pedestrians by direction. In M. Schreckenberg and S.D. Sharma, editors, *Pedestrian and Evacation Dynamics*, Proceedings of the 1st international conference, pages 115–121. Springer, 2001.

Brent R.P. *Algorithms for Minimization Without Derivatives*. Prentice-Hall, 1972.

Cavens D. Ph.D. thesis, Swiss Federal Institute of Technology ETH, in preparation.

Cavens D. and Lange E. Hiking in real an virtual worlds. In Schrezenmayr et al, editor, *The Real and the Virtual World of Planning*, pages 145–165. 2003.

Cetin N. *Large scale parallel graph-based simulations*. Ph.D. thesis, Swiss Federal Institute of Technology ETH, 2005.

Dahmann J., Fujimoto R., and Weatherly R. The department of defense high level architecture. In *Proceedings of the 29th conference on Winter simulation*, pages 142–149. ACM Press, 1997. ISBN 0-7803-4278-X.

Dijkstra E. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269 – 271, 1959.

Dijkstra J., Jessurun J., and Timmermanns H. A multi-agent cellular automata model of pedestrian movement. In M. Schreckenberg and S.D. Sharma, editors, *Pedestrian and Evacation Dynamics*, Proceedings of the 1st international conference, pages 173–179. Springer, 2001.

Expat www page. James Clark's Expat XML parser library. expat.sourceforge.net, accessed 2004.

Ferber J. *Multi-agent systems. An Introduction to distributed artificial intelligence*. Addison-Wesley, 1999.

Floyd S., Jacobson V., Liu C.G., McCanne S., and Zhang L. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997.

Fujimoto R. Time management in the high level architecture. 1998.

Fujimoto R. Exploiting temporal uncertainty in parallel and distributed simulations. In *Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pages 46–53. IEEE Computer Society, 1999. ISBN 0-7695-0155-9.

Galea E. Simulating evacuation and circulation in planes, trains, buildings and ships using the EXODUS software. In M. Schreckenberg and S.D. Sharma, editors, *Pedestrian and Evacation Dynamics*, Proceedings of the 1st international conference, pages 203–225. Springer, 2001.

Gimblett R., editor. *Integrating Geographic Information Systems and Agent -Based Modeling Techniques*. Oxford Univerity Press, Oxford, 2002.

Gingold R. and Monaghan J. Smoothed particle hydrodynamics - theory and application to non-spherical stars. *Royal Astronomical Society, Monthly Notices*, 181, 1977.

Gloor C. *Modelling of autonomous agents in a realistic road network (in German)*. Diplomarbeit, Swiss Federal Institute of Technology ETH, Zürich, Switzerland, 2001.

Gloor C., Cavens D., Lange E., Nagel K., and Schmid W. A pedestrian simulation for very large scale applications. In A. Koch and P. Mandl, editors, *Multi-Agenten-Systeme in der Geographie*, number 23 in Klagenfurter Geographische Schriften, pages 167–188. Institut für Geographie und Regionalforschung der Universität Klagenfurt, 2003.

Helbing D., Farkas I., and Vicsek T. Simulating dynamical features of escape panic. *Nature*, 407:487–490, 2000.

Helbing D., Schweitzer F., Keltsch J., and Molnar P. Active walker model for the formation of human and animal trail systems. *Phys. Rev. E*, 56(3):2527–2539, 1997.

Hoogendoorn S. and Bovy P. Normative pedestrian behaviour theory and modelling. In *Proceedings of the 15th International Symposium on Transportation and Traffic Theory, Adelaide, Australia*, pages 219–245. 2002.

Hoogendoorn S., Bovy P., and Daamen W. Microscopic pedestrian wayfinding and dynamic modelling. In M. Schreckenberg and S.D. Sharma, editors, *Pedestrian and Evacation Dynamics*, Proceedings of the 1st international conference, pages 123–154. Springer, 2001.

InfiniBand Trade Association www page. Infiniband. http://www.infinibandta.org/, accessed 2005.

Jefferson D. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985. ISSN 0164-0925.

Keßel A., Klüpfel H., Wahle J., and Schreckenberg M. Microscopic simulation of pedestrian crowd motion. In M. Schreckenberg and S.D. Sharma, editors, *Pedestrian and Evacation Dynamics*, Proceedings of the 1st international conference, pages 193–200. Springer, 2001.

Kistler D. *Mental maps for mobility simulations of agents*. Master's thesis, ETH Zurich, 2004.

Krauß S. *Microscopic modeling of traffic flow: Investigation of collision free vehicle dynamics*. Ph.D. thesis, University of Cologne, Germany, 1997. See www.zaik.uni-koeln.de/˜paper.

Lamport L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. ISSN 0001-0782.

MATSIM www page. MultiAgent Transportation SIMulation. See www.matsim.org, accessed 2004.

Mattern F. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

Mauron L. *Pedestrians simulation methods*. Diploma thesis, Swiss Federal Institute of Technology ETHZ, 2002.

Meyer-König T., Klüpfel H., and Schreckenberg M. Assessment and analysis of evacuation processes on passenger ships by microscopic simulation. In M.S. et al, editor, *Pedestrian and Evacuation Dynamics*, pages 297–302. Springer, 2001.

MPI www page. www-unix.mcs.anl.gov/mpi/, accessed 2005. MPI: Message Passing Interface.

MPICH www page. www-unix.mcs.anl.gov/mpi/mpich/, accessed 2005. MPI: Message Passing Interface MPICH implementation.

Müller H. and Landes A. Standortbestimmung Destination Gstaad-Saanenland: Gästebefragung Sommer 2001. Schlussbericht, Forschungsinstitut für Freizeit und Tourismus der Universität Bern, 2001.

Myricom www page. Myrinet. www.myri.com, accessed 2005. Myricom, Inc., Arcadia, CA.

Nicol D. and Fujimoto R. Parallel simulation today. *Annals of Operations Research*, (53):249–285, 1994.

Nishinari K., Kirchner A., Nazami A., and Schadschneider A. Extended floor field CA model for evacuation dynamics. In *Special Issue on Cellular Automata of IEICE Transactions on Information and Systems*, volume E84-D. 2001.

Nolle L., Wong K., and Hopgood A. Darbs: A distributed blackboard system. In M.Bramer, F.Coenen, and A.Preece, editors, *Proc. ES2001, Research and Development in Intelligent Systems XVIII*, pages 161–170. Springer, 2001.

Ortúzar J.d.D. and Willumsen L. *Modelling transport*. Wiley, Chichester, 1995.

de Palma A. and Marchal F. Real case applications of the fully dynamic METROPOLIS tool-box: an advocacy for large-scale mesoscopic transportation systems. *Networks and Spatial Economics*, 2(4):347–369, 2002.

Paul M. Torrens www page. GeoSimulation. see www.geosimulation.com, accessed 2005.

Ponder M., Papagiannakis G., Molet T., Magnenat-Thalmann N., and Thalmann D. VHD++ development framework: Towards extendible, component based VR/AR simulation engine featuring advanced virtual character technologies. In *Computer Graphics International*. IEEE Computer Society, 2003.

Povray www page. http://www.povray.org/, accessed 2005.

PTV www page. Planung Transport Verkehr. See www.ptv.de, accessed 2004.

PVM www page. www.epm.ornl.gov/pvm/, accessed 2004. PVM: Parallel Virtual Machine.

Quinn M., Metoyer R., and Hunter-Zaworski K. Parallel implementation of the social forces model. In E.R. Galea, editor, *Pedestrian and Evacuation Dynamics 2003*, Proceedings of the 2nd international conference. CMS Press, University of Greenwich, 2003.

Raney B. *Large scale agent learning*. Ph.D. thesis, Swiss Federal Institute of Technology ETH, 2005.

Raney B. and Nagel K. Truly agent-based strategy selection for transportation simulations. Paper 03-4258, Transportation Research Board Annual Meeting, Washington, D.C., 2003.

Raney B. and Nagel K. An improved framework for large-scale multi-agent simulations of travel behavior. In *Proceedings of Swiss Transport Research Conference (STRC)*. Monte Verita, CH, 2004a. See www.strc.ch.

Raney B. and Nagel K. Iterative route planning for large-scale modular transportation simulations. *Future Generation Computer Systems*, 20(7):1101–1118, 2004b.

Raney B. and Nagel K. An improved framework for large-scale multi-agent simulations of travel behavior. In P. Rietveld, B. Jourquin, and K. Westin, editors, *Towards better performing European Transportation Systems*. accepted.

Renderman www page. Pixar animation studios. https://renderman.pixar.com/, accessed 2005.

Repast www page. Recursive porous agent simulation toolkit. repast.sourcefourge.net, accessed 2003.

Rickert M. *Traffic simulation on distributed memory computers*. Ph.D. thesis, University of Cologne, Cologne, Germany, 1998. See www.zaik.uni-koeln.de/~paper.

Riley G., Fujimoto R., and Ammar M. Network aware time management and event distribution. In *Proceedings of PADS 2000: 14th Workshop on Parallel and Distributed Simulation*. 2000.

Salvini P. and Miller E. ILUTE: An operational prototype of a comprehensive microsimulation model of urban systems. In *Proceedings of the meeting of the International Association for Travel Behavior Research (IATBR)*. Lucerne, Switzerland, 2003. See www.ivt.baum.ethz.ch.

Schadschneider A. Cellular automaton approach to pedestrian dynamics - theory. In M. Schreckenberg and S.D. Sharma, editors, *Pedestrian and Evacation Dynamics*, Proceedings of the 1st international conference, pages 75–85. Springer, 2001.

SCS www page. http://www.scs.ch/, accessed 2005. The T-Net Hardware.

Windows Sockets. WinSock development information. http://www.sockets.com/, accessed 2005.

Stucki P. *Obstacles in Pedestrian Simulations*. Diploma thesis, Swiss Federal Institute of Technology ETH, 2003.

Timmermans H. The saga of integrated land use-transport modeling: How many more dreams before we wake up? In *Proceedings of the meeting of the International Association for Travel Behavior Research (IATBR)*. Lucerne, Switzerland, 2003. See www.ivt.baum.ethz.ch.

TRANSIMS www page. TRansportation ANalysis and SIMulation System. transims.tsasa.lanl.gov, accessed 2005. Los Alamos National Laboratory, Los Alamos, NM.

Tsuji Y. Numerical simulation of pedestrian flow at high densities. In E.R. Galea, editor, *Pedestrian and Evacuation Dynamics 2003*, Proceedings of the 2nd international conference, pages 27–38. CMS Press, University of Greenwich, 2003.

Unger H. *Modellierung des Verhaltens autonomer Verkehrsteilnehmer in einer variablen staedtischen Umgebung*. Ph.D. thesis, TU Berlin, 2002.

UserLand Software, Inc. www page. XML-RPC. see www.xmlrpc.com, accessed 2005.

Veit H. and Richter G. The FTA design paradigm for distributed systems. *Future Gener. Comput. Syst.*, 16(6):727–740, 2000. ISSN 0167-739X.

Waddell P., Borning A., Noth M., Freier N., Becke M., and Ulfarsson G. Microsimulation of urban development and location choices: Design and implementation of UrbanSim. *Networks and Spatial Economics*, 3(1):43–67, 2003.

Weidmann U. *Transporttechnik der Fussgänger*, volume 90 of *Schriftenreihe des IVT*. Institute for Transport Planning and Systems ETH Zürich, 2 edition, 1993. In German.

Wolfram S. *Theory and Applications of Cellular Automata*. World Scientific, Singapore, 1986.

www.corba.org. CORBA: Common Object Request Broker Architecture. accessed 2005.

XBIS XML Information Set Encoding www page. XBIS. see xbis.sourceforge.net, accessed 2005.

Ziv J. and Lempel A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

Curriculum Vitae

Name	Christian Daniel Gloor
Affiliation	Swiss Federal Institute of Technology (ETH)
	Department of Computer Science, Zürich, Switzerland
EMail	chgloor@inf.ethz.ch
Date of Birth	27.12.1975

Formal Education

2002–date	ETH Zürich, Switzerland: Doctorate at the Inst. of Computational Science
1996–2001	ETH Zürich, Switzerland: Diplomstudium at the Dept. for Computer Science
1992–1996	Kantonsschule Wohlen Typus C (Scientific)
1987–1992	Bezirksschule Mutschellen

Publications and Papers Presented at Conferences

2004 Christian Gloor, Pascal Stucki and Kai Nagel: Hybrid techniques for pedestrian simulations, Lecture Notes in Computer Science, Cellular Automata: 6th International Conference on Cellular Automata for Research and Industry, ACRI 2004, Amsterdam, The Netherlands, October 25-28, 2004. Proceedings.

2004 Christian Gloor and Kai Nagel: A Message Based Framework for Real World Mobility Simulations, presented at the 17th International Conference on Computer Animation & Social Agents 2004 in Geneva.

2004 Christian Gloor and Kai Nagel: A Message Based Framework for Real World Mobility Simulations, presented at the Third International Joint Conference on Autonomous Agents & Multi Agent Systems (Workshop on Agents in Traffic and Transportation) in New York (AAMAS 2004). To be published by Birkhäuser.

2004 Christian Gloor, Pascal Stucki and Kai Nagel: Hybrid techniques for pedestrian simulations, presented at the Swiss Transportation Research Conference 2004 in Ascona (STRC'04)

2003 Christian Gloor, Duncan Cavens, Eckart Lange, Kai Nagel, Willy Schmid: A pedestrian simulation for very large scale applications, in A. Koch and P. Mandl (Editors): Multi-Agenten-Systeme in der Geographie. Klagenfurt. (Klagenfurter Geographische Schriften)

2003 Christian Gloor, Laurent Mauron, and Kai Nagel: A pedestrian simulation for hiking in the Alps, presented at the Swiss Transportation Research Conference 2003 in Ascona (STRC'03)