Investigating the Technical Possibilities of Real-time Interaction with Simulations of Mobile Intelligent Particles

vorgelegt von

Dipl.-Inform. David Strippgen

aus Duisburg

von der Fakultät V - Verkehrs- und Maschinensysteme
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr.-Ing. T. Richter
Berichter:	Prof. Dr. rer.nat K. Nagel
Berichter:	Prof. P. Salvini, PhD

Tag der wissenschaftlichen Aussprache: 22.09.2009

Berlin 2009

D 83

Acknowledgments

I would like to express my gratitude to my supervisor, Prof. Dr. Kai Nagel, for giving me the opportunity to work freely on the ideas that finally culminated in this thesis. I would also like to thank him for his valuable advice, which was not limited to technical matters, and most of all for his patience during the long period it took me to make up my mind about pursuing a doctoral degree. I would also like to thank Prof. Dr. Paul Salvini for kindly agreeing to be the second advisor.

Furthermore, I am grateful to Carolin Gesing, Dr. Tomas Becker and Judith Diane Weston, who offered plenty of their valuable time to proofread sections of this thesis and help me improve my basic English skills.

Last but not least, I would like to thank my wife Franzi and my family for their support and love, which helped me to get along.

Contents

Contents **4**

List of Figures **9**

List of Tables **11**

Listings **12**

1 Introduction **1**
1.1 Roadmap of this dissertation 2
1.2 Traffic simulation 3
1.3 Visualization 4
1.4 The evolution of (graphics) hardware 6
1.4.1 A short history of consumer graphics cards 6
1.4.2 The free lunch is over 7

2 The MATSim framework for traffic generation **13**
2.1 The four-step process and beyond 13
2.1.1 Agent-based dynamic traffic assignment 14
2.1.2 Dynamic traffic assignment 16
2.1.3 Iterative demand optimization 16
2.2 The strategic layer 18
2.2.1 Agent database 19
2.2.2 Co-evolutionary algorithm 20
2.2.3 Execution of the physical layer 21
2.2.4 Evaluation of the fitness of a plan with a scoring function 22
2.2.5 Replanning modules 23
2.3 The physical layer 24
2.3.1 Gawron's queue model 25
2.3.2 Improvements to Gawron's algorithm 26
Fair intersection 26
Parallel update 28

3 Implementing the queue model on graphics devices 31
3.1 CUDA SDK . . . 31
3.1.1 The NVIDIA hardware . . . 33
Streaming multiprocessors . . . 33
Warps and half-warps . . . 34
Multiprocessor memory layout . . . 34
Accessing memory . . . 36
Coalesced memory access versus uncoalesced access . . 37
Technical capabilities of the Geforce G80 series . . . 39
Summary . . . 39
3.1.2 The CUDA SDK software API . . . 40
CUDA Computing model . . . 40
A grid of thread blocks . . . 41
Memory types of the API . . . 44
Compiler for CUDA . . . 44
Atomic operations . . . 45
OpenGL and DirectX interoperability . . . 46
Asynchronous execution . . . 46
3.1.3 Theoretical peak of calculation speed and memory bandwidth . . . 46
Some standard problems . . . 47
Benchmarking of standard problems and discussion . . 48
3.2 The queue model transferred to CUDA . . . 50
3.2.1 Parallelize the queue model . . . 50
Simulation code without node logic . . . 51
3.2.2 Activity chains . . . 52
Data structures for activities . . . 53
Data structures for Routes . . . 53
A complete plan . . . 54
3.2.3 Network data . . . 54
Links . . . 54
Nodes . . . 56
3.2.4 Administrative structures . . . 56
3.2.5 Array of structs . . . 57
3.2.6 Struct of arrays . . . 58
3.2.7 Ring buffer . . . 60
3.2.8 Vehicle data . . . 61
3.3 The simulation loop . . . 62
3.3.1 The moveLink() kernel . . . 62
3.3.2 The moveBuffer() Kernel . . . 64
3.3.3 The moveNode() Kernel . . . 66
3.3.4 The moveNode() Kernel with random selection . . . 66
3.3.5 The alternative moveNode() and moveBuffer() kernels 68
3.3.6 The insertWaiting() kernel . . . 69

3.4 Benchmark and performance results . . . 70
3.4.1 Hardware . . . 71
3.4.2 Data samples and network . . . 72
3.4.3 Results for the GeForce 8600 GT . . . 73
Results for AOS and SOA structs . . . 73
Results for the RING implementations . . . 73
3.4.4 Results for the GeForce GTX280 . . . 75
Results for AOS and SOA structs . . . 75
Results for the RING implementations . . . 77
3.4.5 Conclusion . . . 77

4 Integrating the CUDA Simulation into the MATSim framework 81
4.1 Event generation and transportation . . . 81
Data structure for events . . . 82
Agents with all activities on one link . . . 84
Memory demand for Events . . . 84
4.1.1 Mapping of internal link's and agent's representation . 85
4.1.2 Event generation . . . 85
4.2 Performance results with Events . . . 86
4.3 The CPU loop . . . 87
4.3.1 Asynchronous event handling . . . 87
4.4 JNI interface and coupling to Java . . . 91
4.4.1 Toolkits for coupling C++ and Java . . . 91
4.4.2 Transferring events to the MATSim framework . . . 91
4.4.3 Implementing wrapper classes for the plans . . . 92
4.5 Implementing further functionality on the CUDA device . . . 96
4.5.1 Running event handling an simulation simultaneously 98

5 Visualization and Interaction 99
5.1 State-of-practice in traffic visualization . . . 100
5.1.1 TRANSIMS . . . 100
Discussion . . . 100
5.1.2 Visualizer of Christian Gloor . . . 101
Discussion . . . 102
5.1.3 MATSim's NetVis . . . 102
Discussion . . . 102
5.1.4 Other multi-agent simulations . . . 103
Discussion . . . 103
5.2 Goals and motivation . . . 103
5.2.1 An on-the-fly visualizer for the MATSim framework . 104
The MVC (Model-View-Controller) pattern . . . 106
Client-Server . . . 106
Java's RMI and Serializable Interface . . . 107

OpenGL . . . 108
5.3 Architecture of an interactive system . . . 108
5.3.1 Data formats and interfaces . . . 108
Writer and Reader . . . 109
Receiver . . . 110
Visualizer . . . 110
5.3.2 Quadtrees for spatial data storage . . . 110
5.3.3 Patterns used . . . 112
Proxy . . . 113
Factory method . . . 113
Visitor . . . 113
Bridge . . . 113
Decorator and Observer . . . 113
5.4 Implementation . . . 114
5.4.1 Client implementation . . . 114
OGLClient . . . 114
SwingClient . . . 116
Interaction . . . 117
5.4.2 Server implementation . . . 119
Server for OTFVis movie files . . . 120
Event file conversion . . . 122
Server for the TRANSIMS format . . . 122
On-The-Fly-Server for interactive simulation runs . . . 122
Querying the OnTheFlyServer . . . 124
5.4.3 Managing the data flow . . . 124
Building a visualization view . . . 127
The data flow from server to screen . . . 129
Java RMI . . . 130
5.4.4 Extensibility . . . 133
5.4.5 Version management . . . 134
Moving classes . . . 135
Adding or removing data . . . 136
Version management of the Buffer I/O . . . 138
5.5 Optimizations . . . 140
5.5.1 Data exchange via quadtrees . . . 140
5.5.2 Caching . . . 142
Refreshing the screen view . . . 142
SceneGraph . . . 143
SceneLayer . . . 143
The drawer implementation . . . 144
The chain of responsibility . . . 145
Working with the quadtree . . . 148
5.6 Conclusion . . . 148

6 OTFVis and CUDA **151**
6.1 Additions to the server side 151
6.2 Using device memory to exchange data 153
6.3 Calling the kernel to render the positions into a VBO 153
6.4 The handler for client-server communication 155
6.5 The drawing on the client side 155
6.6 Result from interactive runs 155

7 Conclusion and Outlook **159**
7.1 Running the server as an external module 159
7.1.1 Running several iterations 159
7.2 Adding reflection 161
7.3 Summary 162

Bibliography **165**

List of Figures

1.1 Schematic view of an interaction between processes 2
1.2 Evolution of the computing power and bandwidth of GPUs [49, 48] 8

2.1 The four step process of traffic assignment 15
2.2 The two layers of MATSim 17
2.3 Dynamic traffic assignment with agents 18
2.4 Parallel update by including an additional flow buffer. 29

3.1 CPU versus GPU layout. 32
3.2 Layout of the multiprocessors 35
3.3 Coalesced and uncoalesced memory access on the GPU. . . . 38
3.4 The grid of blocks of threads defined by $<<<(3,2),(2,4)>>>$ 42
3.5 The nvcc compilers compilation path (taken from [31]) 45
3.6 Translation of a XML Plan into the GPU structure. 55
3.7 Administration structure for vehicle data residing on a link or in a buffer as AoS (array of structs) 58
3.8 SoA (struct of arrays) layout for the administrative structure 59
3.9 Translation of the struct from AoS to SoA. 59
3.10 Administration structure implemented as a ring buffer 61
3.11 Speedup of the GeForce 8600 GT relative to the Java version 74
3.12 Speedup of the GeForce 8600 GT relative to the Java version (continued) 74
3.13 Speedup of the GTX280 relative to the Java version 76
3.14 Speedup of the GTX280 relative to the Java version (continued) 76
3.15 Real time ratio of the GTX280 (simulated secs / wall clock secs) 78
3.16 Speedup of the GTX280 relative to the Geforce 8600 GT (continued) 79

4.1 Parallel execution of CUDA simulation and CPU event handling 90
4.2 UML diagram of the relationship between the abstract and concrete implementations in Java and C++ 94

5.1 The MVC pattern for OTFVis 107
5.2 Sending dynamic data from server to client over RMI using byte buffers 109
5.3 Space decomposition and quadtree 111
5.4 Different layouts of the OTFVis. 115
5.5 The timeline bar for user interaction. 117
5.6 The query bar for user interaction. 117
5.7 UML diagram of the OTFServerRemote interface 119
5.8 UML diagram of the OTFLiveServerRemote interface 125
5.9 UML diagram of the OTFQuery interface 125
5.10 A visualization using the split view pane. 134
5.11 The update process per quadtree 141
5.12 Different unread regions resulting from user interaction. . . . 145
5.13 Requesting a new time step t+1. 147

7.1 An agent's plan visualized using PREFUSE 162

List of Tables

3.1 Benchmark results (MO = Mega Options) 48
3.2 Data format for plan chunks 52
3.3 Data structures and execution paths implemented for benchmarking. *(The last row is named "Event" as this configuration is used for implementing the event handling in Sec. 4.1)* 71
3.4 Technical data of CPUs and GPUs used 71
3.5 Real time ratios for the CPU-based Java simulation and sample sizes used in this thesis (all ratio values were taken at midnight, 24:00) 72

4.1 List of events used in the MATSim framework 82
4.2 Execution paths and maximum event count of these 84
4.3 Changes in real time ratio at midnight (24:00) with event writing (25% sample run on a GeForce 8600GT, EVENTNODES) 86
4.4 Running times for different implementations for link time aggregation 97
4.5 Running times for different implementations for link time aggregation with CPU and GPU working simultaneously 98

5.1 List of OpenGL related classes 116
5.2 List of queries implemented for the visualizer 124

6.1 Running times for stepping 24 hours forward with the OTFVis using CUDA and Java 157

Listings

2.1 A sample plans file in XML representation 19
2.2 The movement code by Gawron 25
2.3 Fair vehicle movement at the intersections as in Cetin [24]. . 27
2.4 Choosing eligible links 28
2.5 Vehicle movement at the intersections as presented by Cetin. (Note how the algorithm separates the flow capacity from intersection dynamics.) 29
3.1 Kernel implementation for the *triad* operation 47
3.2 Kernel implementation for the *Black-Scholes* formula for European options 49
3.3 Pseudo code for queue movement 51
3.4 Pseudo code for buffer movement 52
3.5 Kernel for link movement (AoS) 63
3.6 Kernel for buffer only movement (AoS) 65
3.7 Kernel for node movement (AoS) 66
3.8 Kernel for flow-capacity-dependent random pick of a starting link 68
3.9 Kernel for executing activities (AoS) 69
4.1 Struct for event data 83
4.2 The main CPU loop executing the kernels 88
4.3 The CPU loop's event handling section 89
4.4 The Java wrapper of the CUDA simulation 93
4.5 The abstract classes for plans exchange 95
5.1 Interface for a query options tab 118
5.2 Initializing and updating data in OTFVis 121
5.3 Class OnTheFlyQueueSimQuad holds an instance of OnTheFlyServer 123
5.4 A sample OTFConnectionManager instance 126
5.5 Creating a new View onto the simulation 127
5.6 A receiver interface for quad data 130
5.7 Interface for the remote server classes 132
5.8 New coloring scheme drawer for agents 133
5.9 Resolver for Java classes 135

5.10 Implementation of a simple LinkHandler class 141
5.11 The principal steps for updating a region in the client 146
6.1 OTFClient class for CUDA 152
6.3 Calling the kernel for the VBO update 153
6.2 Use of a VBO on the CUDA device 154
6.4 The drawer for drawing from the VBO 156
7.1 Running the server as an external module 160

Abstract

Using simulation systems for transport planning has become more popular in the past two decades. Most of these systems rely on the "four-step process". A more recent approach is to use transport-planning systems based on multi-agent simulations. MATSim is a framework for building multi-agent-based simulations for transport systems. Unlike the flow-based models such as the one used by the software VISUM, multi-agent systems are based on distinct daily plans for the whole simulated population. These plans are then executed in a simulated world for all agents in parallel.

One advantage of multi-agent simulations is that it is possible to build a complex system of traffic interaction by modeling the agent's behavior with simple rules. The complex behavior of the system is broken down to the behavior of simple agents.

By using the computationally inexpensive "Queue Model", the MATSim framework is capable of simulating even large-scale networks and population groups. The traffic demand of cities, counties or even entire countries can be simulated. For example, all of traffic in the Berlin-Brandenburg region in Germany with its seven million agents has been simulated using the MATSim framework.

Unfortunately, only more or less aggregated data can be used as an output of these simulation runs. Given the enormous amount of data generated during a simulation run, it is not feasible to examine the unaggregated data. However, since an aggregated view of the results of the simulation may obscure important details, a more finely granular view would be beneficial.

The aim of this thesis is to enable the researcher to have an unaggregated view of the simulation. In the first part of this thesis, ways of visualizing the unaggregated data will be examined. The unaggregated view of the actual traffic flow promises to be a useful tool when looking for causes of observed phenomena. The graphics capabilities of computers have continuously increased over the past years. Trying to visualize hundreds of thousands of agents simultaneously on screen no longer seems far-fetched. A software architecture will be developed that is capable of displaying these quantities of agents while being open and expendable enough to allow the researcher to add any form of visualization him or herself. This architecture will enable the researcher to investigate cause-and-effect chains in a multi-agent simulation by examining the unaggregated data. To achieve this, modern hardware acceleration of 3-D graphics is extended to accelerate the display of primarily 2-D traffic data.

The second part of this dissertation will address possible ways to increase the execution speed of traffic simulations by means of modern graphics hardware. Accelerating the execution speed of simulations has already

been tried a few times. This has normally involved incorporating large clusters of computers or using some other sort of expensive supercomputing hardware. However this is probably not a feasible way to reach a state in which ordinary engineering offices can use a multi-agent simulation to preview the results of a planned measure. Using modern graphics hardware, on the other hand, is simple and cheap. All that needs to be acquired is a €300 graphics card and a regular office PC. It will be shown that it is possible to achieve a speedup of up to 70 times more than the "usual" Java version of mobility simulations by using graphics hardware to calculate the simulation. This will make high-performance computing affordable even for small companies. Finally, fast simulation and visualization will be brought together to create a system capable of displaying a simulation of hundreds of thousands of agents in real time.

Zusammenfassung

Der Einsatz simulationsbasierter Systeme für die Verkehrsplanung hat sich in den letzten beiden Jahrzenten etabliert. Viele der Systeme basieren im Moment auf dem "4-Stufen Modell". Eine neuere Entwicklung ist es, die Simulation auf einer Multi-Agenten-Simulation aufzubauen. MATSim ist ein solches Framework für die Multi-Agenten-Simulation von Verkehr. Anders als in Fluss-basierten Modellen, wie z.B. von der Software VISUM genutzt, wird in der Multi-Agenten-Simulation eine Vielzahl von Agenten mit distinkten Tagesplänen erzeugt. Diese Agenten durchlaufen dann in einer physikalischen Simulation ihren Tagesablauf.

Multi-Agenten-Simulationen haben den intrinsischen Vorteil, dass das Zusammenspiel verschiedenster Verhaltensweisen und die daraus resultierenden komplexen Systeme mit einfachen Regeln gestaltet werden können. Das ganze System wird auf das Verhalten einzelner Individuen herunter gebrochen.

MATSim eignet sich durch das effizient berechenbare Queue-Modell sehr gut zur Simulation auch großer Städte oder Regionen. So wird zum Beispiel die Region Berlin-Brandenburg mit ihren ca. 7 Millionen Einwohnern simuliert. Sogar der Verkehr ganzer Länder ist in MATSim modellierbar. Die enorme Menge an Daten macht es mit herkömmlichen Mitteln allerdings nicht möglich, diese in einer interaktiven Weise zu untersuchen. Eine feinmaschige Visualisierung ist aber ein wichtiges Handwerkszeug zur Evaluierung von Agenten-basierten Simulationen.

Hier setzt die Arbeit dieser Dissertation in zweierlei Weise an. Im ersten Teil der Arbeit wird untersucht, inwieweit es mit modernen Computern möglich ist, diese großen Mengen an Agenten zu visualisieren. Diese Visualisierung der tatsächlichen Verkehrsflüsse ermöglicht eine bessere Untersuchung der Ursachen von beobachteten Phänomenen, wie zum Beispiel der Stau-Bildung. Für einen Versuch, Hunderttausende von Agenten gleichzeitig darzustellen, sind die verwendeten Algorithmen sorgfältig auszuwählen. Es wird eine Architektur entwickelt und vorgestellt, die in der Lage ist, die gewünschte Zahl von Agenten zu präsentieren. Sie wird dabei jedoch flexibel genug sein, jegliche Art von Information aus der Simulation mit kleinem Aufwand zu visualisieren. Die auf dieser Architektur beruhende Visualisierung wird es ermöglichen, verkehrswissenschaftlich interessante Szenarien in einer neuen Detailgenauigkeit zu untersuchen.

Der zweite Teil der Dissertation beschäftigt sich mit der Frage, inwieweit man die inzwischen verbreiteten 3D-Grafikkarten nicht nur für die Darstellung, sondern direkt für die Berechnung von Verkehrssimulationen einsetzen kann. Hier wird der Versuch unternommen, herkömmliche Grafikkarten

zu nutzen, um die Simulation auf interaktive Geschwindigkeit zu beschleunigen. Dabei zeigt sich, dass die Ausführung der Simulations-Schicht auf das Siebzigfache beschleunigt werden kann. Des Weiteren werden Wege aufzeigt, diese neue Simulation nahtlos in das bestehende MATSim Framework einzubetten. Letztendlich wird die so entstandene Simulation genutzt, um die interaktive Visualisierung von Vorgängen zu beschleunigen und zu unterstützen.

Der Vorteil dieser Herangehensweise liegt auf der Hand. Ein solches System ist von der reinen Simulationsgeschwindigkeit mit Clustern im Wert mehrerer hunderttausend Euro zu vergleichen. Es ist aber für vergleichsweise geringe Anschaffungskosten von ca. fünfhundert Euro und vernachlässigbaren Betriebskosten bereitzustellen. Jedes größere Ingenieurbüro sollte in der Lage sein, eine solche Anschaffung zu tätigen. Dies mag –natürlich im Zusammenspiel mit sinnvollen Resultaten der Simulation– der Akzeptanz von Multi-Agenten-Systemen zuträglich sein.

Chapter 1

Introduction

Playing the trumpet like Miles Davis did is a rare gift. It takes talent to reach such spheres of musicianship. Apart from talent, another key to success is practice. Practicing to play the trumpet is a difficult but certain way to attain at least an basic trumpet competence. Practicing an instrument means interacting with it. One could read a lot of books on trumpet playing and still not be able to coax more than obscene noises from the instrument. Practicing implies interaction with a chosen subject. Experiencing the principle of cause and effect helps one build a better understanding of or, more commonly placed, a better "intuition" for the area of interest.

Multi-agent transport simulation and trumpet playing differ in many ways, but what they do have in common is that practice and experience are vital tools for understanding and interpreting observed events. Following the Merriam-Webster online dictionary, "interaction" is defined as "mutual or reciprocal action or influence". Fig. 1.1 illustrates this rather elementary circumstance. If we exchange "System A" with "human user" and "System B" with "simulation" one sees that, from the user's perspective, this interaction can only take place when some means of inducing actions upon the simulation and some means of experiencing the results of this actions are supplied. Therefore, an interactive tool –which is the subject of this thesis– needs to offer ways of initiating actions and perceiving the resulting changes in the simulation. The two key issues for interaction are

- the processing user's input and
- the visualization of the simulation's output.

But "Real time interaction" offers one more challenge. The term "real time" does not necessarily mean that one second of the simulated world equals one second of wall-clock time. Rather, the actual interaction with the problem domain modeled in the simulation is expected to happen in "real time", meaning that the simulation may have to speed up or slow down the "simulated time" to present the modeled problem in a humanly

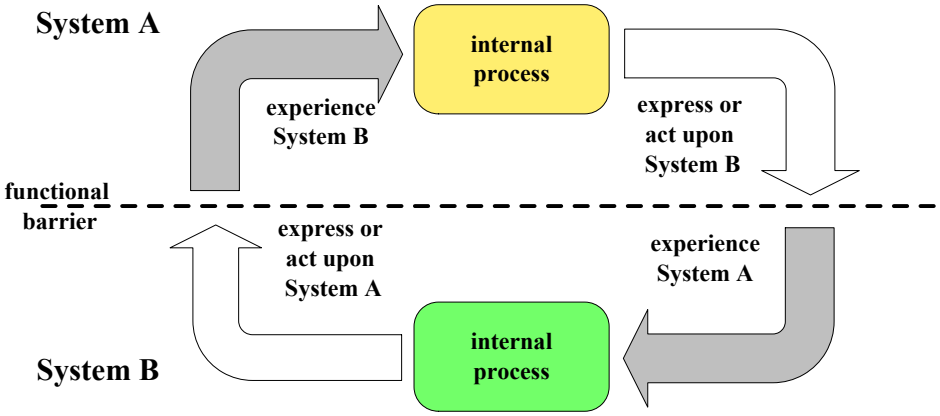


Figure 1.1: Schematic view of an interaction between processes

perceivable time frame. When simulating the impacts of a car accident, it might be necessary to slow down simulated time to be able to analyse the events taking place in seconds. When, on the other hand, the purpose of the simulation is to analyse the consequences of polar glacier melting, a decent speedup of simulated time versus wall-clock time would probably be necessary to enable human interaction and gain insight into the process.

The simulated time period for traffic simulation, which is examined in this thesis, is twenty-four hours. To interact with this period of time, the simulation should be able to run it in one or two minutes of real time. Thus, a ratio of simulated time against real time by an order of 10^3 should be minimally achieved to maintain expedient interaction.

Three key issues were detected for providing means of real time interaction with the multi-agent simulation of traffic or mobile intelligent particles:

- providing input mechanisms for the user to change (the view of) the simulation
- visualizing the state of the simulation in a meaningful way
- accelerating the execution of the simulation so that it is presentable in a humanly perceivable time frame.

The purpose of this thesis is to solve these key issues to provide real time interactive ways to experience traffic simulations and therefore gain insights into cause-and-effect chains of large scale traffic behavior by means of multi-agent simulations.

1.1 Roadmap of this dissertation

This thesis is structured as follows: Chapter 1 will introduce related work. Other traffic simulation systems and visualization tools will be discussed.

The reader will also be given an overview of how and in what domains GPU (graphics processor unit)-based calculations have been successfully used to decrease overall calculation time.

In Chapter 2 the generic model of the queue simulation will be discussed and algorithms which implement a queue-model-based simulation will be given. In addition, the benefits and drawbacks of using the queue model will be considered.

Chapter 3 will introduce the CUDA SDK, which is the SDK used in this thesis to harvest the computational power of the GPU. How to translate given generic algorithms to algorithms that make the best use of the infrastructure of the GPU will be explained. Furthermore, useful data structures for storing necessary data for the agents as well as the network will be constructed. These data structures are needed to feed information into the simulation and to update the state of the simulation. Another data structure for "observing" the characteristics of a simulation run will be introduced. The output of events at critical or interesting points of action will be implemented and discussed. Finally, the possibility of coupling the CUDA simulation to the existing Java code of MATSim will be investigated in Chapter 4.

Chapter 5 will treat the topics of visualization and interaction with MATSim. Therefore some information on the architectural structure of the MATSim framework will be provided. This will be followed by a discussion of client-server architectures with respect to interactive visualization. Then the implementation of the OTFVis (on the fly visualization) tool for MATSim simulation runs will be discussed. Finally, the combination of OTFVis and GPU simulation will be presented in Chapter 6.

Chapter 7 concludes this thesis and will contain a summary of the results achieved.

1.2 Traffic simulation

Using simulation techniques in transport planning has become increasingly popular. Many transport simulation systems have already been implemented. The more traditional approach of the *four-step process* as described in [92, 78] has certain drawbacks, but has been popular for many years, as it is rather easy to apply and yields mathematically provable results. Although its strong mathematical foundation is appealing, its shortcomings are not. As Michael Balmer [13] accurately stated, "Traffic is not caused by vehicles, but by travelers." The MATSim approach is therefore to model every single individual, who causes traffic. By stating fairly simple rules for an individual's behavior, an complex, emergent behavior of the overall system can be observed. With this approach, it is possible to describe the characteristics of individuals or groups of individuals in a freely chosen granularity,

therefore partially refining the system's behavior partially. Conversely, it is also possible to identify the winners and losers of a change to the system and analyse them with respect to different groups within the population. Thus, multi-agent-based simulation tools offer more precise insight into the actual causes of an emergent system's behavior, albeit without the strong mathematical foundation of other approaches. Multi-agent simulations of traffic systems are being developed in many places all over the world, examples can be found in [99, 71, 33]. In this thesis, the multi-agent simulation framework MATSim [66] will be used as a basis for the implementation of a traffic-assignment layer executed on graphics hardware.

1.3 Visualization

As the legendary American baseball player and trainer Yogi Berra put it, "You can observe a lot by watching" [16]. Or as a perhaps more trustworthy source of inspiration, René Descartes, stated in 1637 [58]:

> "Imagination or visualization, and in particular the use of diagrams, has a crucial part to play in scientific investigation."

The term (scientific) visualization is often attributed to McCormick, DeFanti and Brown in their article "Visualization in Scientific Computing" (1987) [68]. According to the rather basic definition of this young discipline given in the article,

> "The goal of Visualization in [scientific] computing is to gain insight by using our visual machinery."

There is quite a lot of definitions for the word "visualization" in terms of (scientific) computation around. Richard Hamming maintained that,

> "The purpose of [scientific] computing is insight, not numbers."

This leads to the question of how visualization can help one to attain insight. The word "sight" in "insight" might point into the right direction. Presenting something as a visual stimulus helps engage a large area of the cortex, as 50% of our neurons are associated with vision. As R.M. Friedhoff and T. Kiley stated,

> "The standard argument to promote scientific visualization is that today's researchers must consume ever higher volumes of numbers that gush, as if from a fire hose, out of supercomputer simulations or high-powered scientific instruments. If researchers try to read the data, usually presented as vast numeric matrices, they will take in the information at snail's pace. If the information is rendered graphically, however, they can assimilate it at a much faster rate".

Finally, as J. Foley and B. Ribarsky propounded,

> "A useful definition of visualization might be the binding (or mapping) of data to representations that can be perceived. The types of bindings could be visual, auditory, tactile, etc., or a combination of these."

The last two citations were taken from [29].

One of the the driving forces behind the development of visualization techniques over the past decade is the ever increasing computational power of workstations. The power to create detailed visual stimuli has increased tremendously leading to new dimensions in visualization. Another trend that increased the interest in visualization is again based on the massive computing power at hand:

Supercomputers calculate and output gigabytes of data nowadays, producing a huge and ever increasing flow of data to be analysed. These numbers cannot be looked at without aggregating or representing them in a multi-dimensional way that engages the visual perception of the human brain, thus leading to new insights.

The demand for visualizing ever larger data sets has kept pace with the capability of doing so.

This is true for traffic simulations as well. The MATSim simulation is used for huge networks comprising millions of agents going about their daily business. Therefore, one simulation run generates gigabytes of data to be analysed. One way to deal with the tremendous volume of data is certainly to aggregate the data and evaluate meaningful operating figures. Aggregation, however, is a useful but dangerous tool; the agents might well behave perfectly according to the given aggregated operation figures, but the simulation might still be completely wrong, but even though it adds up to seemingly correct figures. Therefore, an inspection of the unaggregated data is a useful debugging tool for the researcher.

Offering aid with the inspection of data is one of the main purposes of visualization. DeFanti, Brown and McCormick [34] identified two main goals for visualization: To them visualization is

> "- **A tool for discovery and understanding**. The deluge of data generated by supercomputers and other high-volume data sources (such as medical imaging systems and satellites) makes it impossible for users to quantitatively examine more than a tiny fraction of a given solution. That is, it is impossible to investigate the qualitative global nature of numerical solutions.
>
> (...)
>
> - **A tool for communication and teaching**. Much of the modern sciences can no longer be communicated in print. DNA sequences,, molecular models, medical imaging scans, ...and so on, all need to be expressed and taught visually over time."

This holds true for traffic simulation as well. The first aspect was already discussed above. An additional benefit of visualization is that it can be used to generate images and impressions better suited to inform or instruct the general public about certain aspects of political decisions, e.g., with regards to land-use or traffic-system planning. This is where visualization and presentation meet.

1.4 The evolution of (graphics) hardware

The next two sections will give an overview of developments in the field of graphics hardware as well as certain aspects of CPU development. The first section will describe the way that consumer graphics devices have evolved from simple VGA adapters to today's multi-core architectures. The second part will discuss CPU's development and Moore's Law.

1.4.1 A short history of consumer graphics cards

In the 1980s the first graphics devices were developed by IBM and were named VGAs (Video Graphics Adapters). Their sole purpose was to contain some RAM for storing the letters to be displayed and some simple transistors for converting the letters into scanlines that could be displayed on an external screen. Although in scientific research better solutions were already being used at that time, the consumer market remained largely untouched by these highly expensive developments until the 1990s. Based on the huge success of "2.5"-dimensional games like "DOOM" that were rendered on plain 2D hardware at the time, in 1995 the company 3dfx released the "Voodoo" series of 3D graphics adapters. These were used solely for rendering 3D images; they did not have regular graphics hardware and were therefore installed as an add-on card to supplement a regular graphics device. Therefore, it was even possible –in 1998– to install two of these cards together, which formed something like an early multi-core architecture. In the early 2000s first 3D-capable graphics adapters entered the mass market. Triggered by the invention of games played in a three-dimensional virtual environment, the demand for rendering these virtual worlds in ever-increasing resolutions overburdened the CPUs available at that time.

These 3D graphics devices were restricted to a fixed-function pipeline which could be run in hardware to outplay the CPU on a very confined range of duty. Still, improvements to this devices over the past decade have been outstanding, not only in terms of computing power, but also in terms of the expressiveness of the hardware. The first generations of 3D accelerator cards had a fixed-function pipeline to do the "usual" mathematical transformations in hardware, such as calculating a perspective view or simulating light sources. But with the increasing popularity of 3D games, the need of designers and game developers to distinguish their work from the

work of others became more pronounced. Therefore, hardware manufacturers had to break up the fixed-function pipeline to introduce ways of adding custom-generated code to it. This demand for new features was thwarted by the demand to unify the interfaces needed to implement features on the different graphics devices. This process was accompanied by the development of two significant graphics APIs: DirectX and OpenGL. Both offered a more or less unified view of the different hardware platforms. With DirectX 9.0, the notion of "shaders" was introduced, giving programmers the possibility to transform vertices and shade pixels directly on the graphics device and freely describe the process in some sort of assembler language. This was at first cumbersome and evolved into today's shading languages, which are more reminiscent of some higher languages like C or Java. This development finally led to the devices we find today, which are multi-core computing devices and are no longer limited to the sole purpose of rendering graphics anymore.

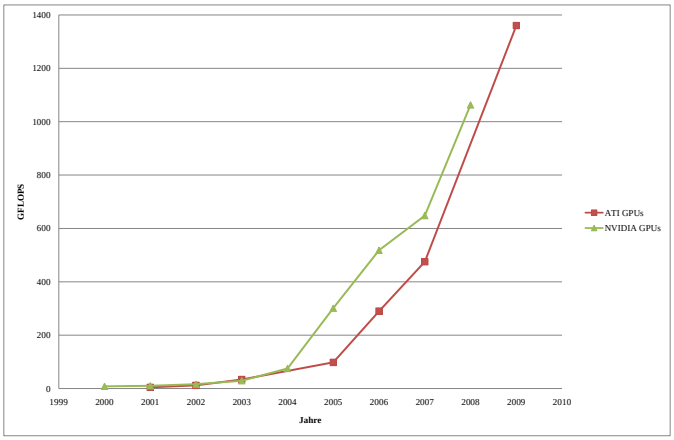
The development of computing power and memory bandwidth over the past decade is illustrated in Figure 1.2 for devices by NVIDIA and ATI. As the figures show, modern graphics devices can execute up to 1 TFLOPS of operations and can transfer more than 100 GB per second. By comparison, the most recent Intel Dual core hardware (Intel Pentium Dual Core Presler 965) issues about 20 GFLOPS [47] and has a memory bandwidth of 5-10 GB per second.

1.4.2 The free lunch is over

In 1965 Gordon Moore made a prediction in his article "Cramming More Components onto Integrated Circuits" in the "Electronics Magazine" which came to be known as *Moore's Law*:

> The complexity for minimum component costs has increased at a rate of roughly a factor of two per year (...) Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.

Although the original formulation – which Moore later changed to a doubling every two years– referred only to transistor densities, it was adopted to many different areas of computing. The overall CPU performance, storage capacity and processing speed have all adhered to Moore's Law. Up until 2002, the processor speed doubled every two years (or even every 18 months,



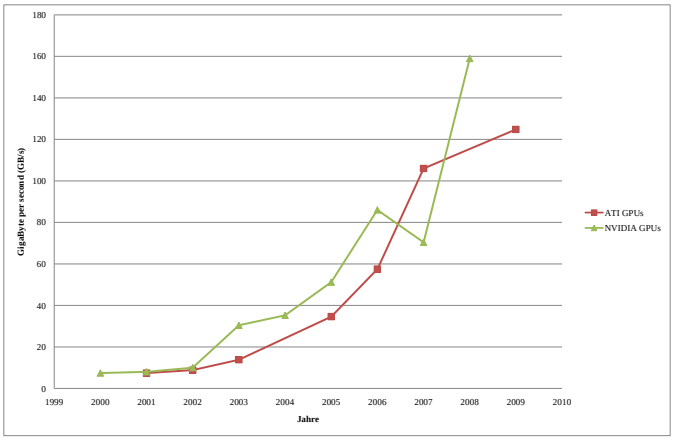


Figure 1.2: Evolution of the computing power and bandwidth of GPUs [49, 48]

as another Intel executive predicted). But this came to a rather sudden halt in the years 2002 to 2004. Intel introduced the first processor running at 3 GHz in November 2002. The peak processor speed was reached in January 2006 with 3.73 GHz. The additional 0.73 Ghz improvement apparently did not conform to Moore's law. Nowadays Intel concentrates on multi-core processors running at lower speeds of around 2.3 GHz [56]. The race for higher processor speeds seems to have come to an end.

Herb Sutter, C++ evangelist and long-time secretary of the C++ standards committee, wrote an article entitled "The Free Lunch is Over: A Fundamental Turn toward Concurrency in Software" [96] in the computer science magazine "Dr. Dobb's Journal," stating the obvious: Moore's Law did not hold any longer in terms of processing speed. The CPU speed had stopped progressing at around 3GHz, not showing any relevant improvement over the next few years. As Moore's Law propagates exponential growth, it was bound to hit the wall some time, because "light is not going to get any faster," as Mr. Sutter put it.

It had been a convenient truth for several years that programmers could rely on the speed improvements of the next generation's CPUs to speedup their application. When an application had a performance lag in some parts of its calculations, a viable way of dealing with it was to simply wait for the next processor generation.

Sutter predicted that although improvements in processor speed have come to a rather abrupt stop, improvements in performance will not. To maintain increasing performance, CPU manufactures will have to resort to concurrency, employing a multitude of cores. In 2005, when Sutter wrote his article, neither Intel nor AMD had multi-core processors on the market, but server- and workframe-oriented companies like Sun Microsystems or IBM were already offering them. Nowadays, Intel and AMD offer four- or even eight-core processors. Additionally Intel plans to release an new kind of processor named *Larrabee*, which is expected to comprise up to 64 cores, sometime in 2010. These multi-core processors promise to be able to follow Moore's Law.

Unfortunately for the lazy programmers, these multi-core processors do not offer their potential for free. Sutter suggested that a new paradigm in software engineering be adopted with this new generation of processors: Concurrent programming. Though this concurrent programming has been around for a while, Sutter maintained that it has not been and will not be adopted by the mainstream until proper toolsets are in place for a reliable development framework. Sutter compared this to the "revolution" of object-oriented programming, which had existed "since at least the days of Simula in the late 1960s," but did not make it into the mainstream until the 1990s due to the lack of reliable tools and the absence of the necessity for such a framework. Not until the 1990s did software projects became so large scale that "OOP's strengths in abstraction and dependency management made it

a necessity for archiving large-scale software development that is economical, reliable and repeatable".

A new programming paradigm is being introduced and no software will benefit from the new multi-cores without adapting to the new programming style. There may have been a noticeable speedup when computers were upgraded from one- to two-core processors, as one of the cores took over all the little tasks such as the virus protection, DRM, drivers, etc. which constantly run on every computer system, thus freeing up the second CPU core to perform the main application better. But this construct will not live on to four or more cores.

Therefore, parallel, concurrent algorithms are a must nowadays for all software which requires powerful performance, as Herb Sutter proclaimed in 2005.

In more recent talks, such as one he held in 2007 at a Northwest C++ User Group (NWCPP) meeting [97], Sutter has also placed great emphasis on the latency of memory accesses as an important factor for performance gains or, more often, performance losses. He claims that latency will cause bandwidth lag when a certain balance is lost.

First, he showed that the latency of CPU systems has not improved as much as bandwidth has in the past twenty years. He stated that over this period, latency had increased by factor of four, whereas bandwidth had increased by a factor of 500, measured in actual wall-clock times. When latency is measured in clock cycles – which apparently is an even more relevant parameter– it decreased by a factor of 150. This led Sutter to draw an analogy to queuing theory:

Little's law states that, given a long term arrival rate λ of customers and a long-term average time a customer spends in a system W, the long-term average count of customers L is:

$$L = \lambda W$$

Sutter adapted Little's law to computing by replacing λ with the bandwidth of a memory system, replacing the average time spent in the system with the latency of the memory system and finally using L as the average count of items executed concurrently:

$$Concurrency = Bandwidth \times Latency$$

This tells us that with staggering latency, an ever-increasing bandwidth can only be utilized by offering higher concurrency. This is interesting with regard to the actual consideration behind the NVIDIA CUDA framework, as will be shown in Chapter 3.1. Only a higher degree of concurrency can save an application's performance from the risk of great latency in memory accesses.

With the advent of multi-core processors and the need to fill the gap between memory latency and bandwidth, parallel models must be found and used as the new programming paradigm for performance hungry applications such as the large-scale traffic simulation. How to achieve this with the aid of graphics devices will be the topic of the next section.

Chapter 2

The MATSim framework for traffic generation

As discussed in the previous chapter, the MATSim framework will be used as a basis for this thesis. The MATSim framework is a traffic simulation framework capable of computing large-scale scenarios. Up to millions of agents with multiple trips per agent can be handled within the MATSim framework.

The agents use genetic algorithms to improve their plans in an iterative process. This chapter will introduce the MATSim framework and the options it offers for *dynamic traffic assignment* (DTA) and activity-based plan generation as well as the feedback learning based on genetic algorithms.

The following section will describe how the framework is constructed and how it provides agents with the ability to learn. The section after that will examine the differences between the MATSim framework and the more traditional "four-step process". Finally, the implementation of the "strategic" and "physical" layers responsible for the actual generation and execution of the agents' plans will be discussed.

2.1 The four-step process and beyond

The "four-step process" described in [92, 78] is a state-of-the-practice simulation technique for traffic-system planning. This process divides a given network into a set of disjunct zones. By describing trips, or more precisely counts of trips, taken by travelers between different zones, it can compute an equilibrium solution of the actual flow of vehicles in the network. This is called the *four-step-process*, as it involves four separate steps of execution. These steps are:

- *Trip generation*: for each zone the number of outgoing trips is evaluated. Likewise, each zone is assigned a number of incoming trips.

- *Trip distribution*: For every zone that has outgoing trips, the distribution of trips to every other zone is determined. That is, the fraction of outgoing trips entering another zone as incoming trips is established. The resulting matrix of trip fractions is called the origin-destination (OD) matrix.
- *Mode choice*: The choice of the mode of transportation is determined here for each trip, i.e., whether the trip was taken by public transport, on foot, by bicycle or by car.
- *Route assignment*: For each trip taken by car, a route is assigned. This process typically searches for a user equilibrium, which means that all paths used for a certain *OD pair* have the same travel time, and no unused path is faster than this travel time [105].

The four step process is illustrated in Figure 2.1

The route assignment process can lead to unique mathematically proven solutions, which is a great advantage of the *four-step-process*. Nevertheless, this process has two major drawbacks. It cannot model time-dependent aspects of the network's load. That is, it cannot take time-dependent factors like peak traffic spreading or congestion spillback into account, making it impossible to spot time-dependent bottlenecks [25].

Another disadvantage of the four-step process is that all data are aggregated at the zone level, or rather, at a stream level describing the traffic between two zones. Individual behavior cannot be distinguished. Therefore, a fine-grained modeling of travelers' behavior based on demographics is hard to introduce into the process.

2.1.1 Agent-based dynamic traffic assignment

The latter drawback can be overcome using *activity-based demand generation (ABDG)*. This models each traveler's plan based on given activities and has different requirements compared to the four-step process. First of all, the zone-wise aggregated information on how many travelers leave or enter zones apparently does not sufficiently describe the demand. Other means of determining the demand for a traffic network have to be been found using activity-based demand generation, as described in [52, 11].

This is a process of choosing daily plans for the agents. Starting from whatever data has been collected from a microcensus or other sources of information, a synthetic population is generated. In a second step, a complete chain of activities is computed for each member of the synthetic population. That is, a chain of typical tasks these members of the population have to deal with over the course of the day is determined. The activities include both a description of the activity type, i.e., "shopping", "working" and "attending school", and the location where the activity will take place. (See also [100]).

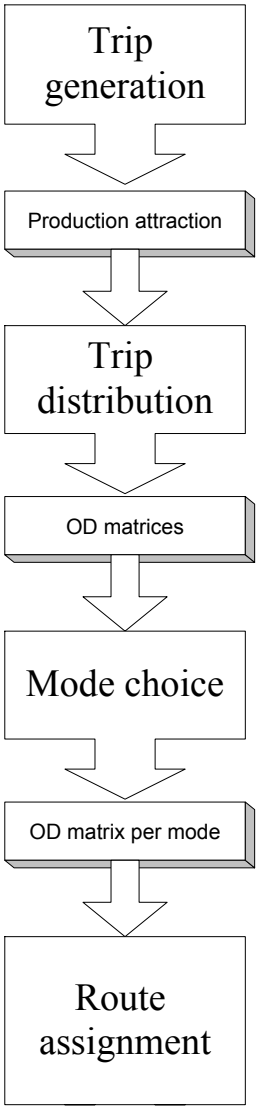


Figure 2.1: The four step process of traffic assignment

Given these sequences of activities at different places, the need to travel arises for the agents. Which mode of transportation they choose can depend on the attributes of the individual traveler, in contrast to the mode-choice step of the four-step process. This mode-choice step can optionally be done simultaneously with the activity choice [62] or after the location choice [64].

2.1.2 Dynamic traffic assignment

Having the description of a whole day's worth of activities and places where the activities will take place, it is now feasible to make the traffic assignment dynamic in contrast to the static assignment of the four-step-process. As the time-based demand is already known from of the activity-based demand generation, it would be a pity to aggregate everything, and throw away the already gained timing information, just to be able to generate OD matrices that fit into the static assignment process.

Dynamic traffic assignment (DTA) has been used for quite a while [59, 8, 19, 37]. Apart from the time-based demand mentioned above, DTA requires a traffic dynamics model that described how vehicles are moved in the network. The major drawback of DTA is that it is not as fully embeddable in a stringent and well-understood mathematical background as the four-step process. Although some theoretical background for DTA is known [19, 23], DTA lacks a provable unique solution. Daganzo, [32] for example, showed that DTA can end up in more than one Nash equilibrium. This makes it more complicated to compare different versions of the DTA process.

To avoid the intricate analytical approach of DTA, it is feasible to use a simulation technique. Assuming that the activity-based traffic assignment is already providing us with complete daily plans, it would be logical to use an agent simulation approach to compute the traffic assignment. The multi-agent simulation approach [38] perfectly complements the activity-based demand generation. Multi-agent simulation exactly needs what demand generation has to offer: *individual plans* for individual agents, describing their unique way through the day.

2.1.3 Iterative demand optimization

Apart from the (initial) demand generation, the simulation-based assignment is often coupled with systematic relaxation, [59, 72, 19] introducing a learning process based on feedback learning. This can be outlined as follows:

1. The assignment starts with some initial routes for the agents' plans. These routes will often be the shortest paths from origin to destination links.

2. Run the simulation, executing all agents' plans simultaneously.

3. Use the results from step 2 to calculate new routes for some or all agents.

4. Continue with step 2

Steps two and three of this iteration can also be viewed as two different layers of the agents' reality. The layer represented in step 2, the execution of the agents' plans, is called the *physical layer*. From an agent's point of view, this is the "reality" he or she is experiencing. Therefore, it is often called "synthetic reality". The other layer, in which the agents re-plan their routes, is based on the actual events of the last simulation run and is often referred to as the "strategic layer". It can also be seen as the agent's "mental map" of the "synthetic reality". These two-layered information exchange is illustrated in Fig. 2.2. The strategic layer provides the simulation with the agents' plans in order to execute them in the network, whilst the simulation provides the strategic layer with a description of observed events so that this information can be used to improve plans.

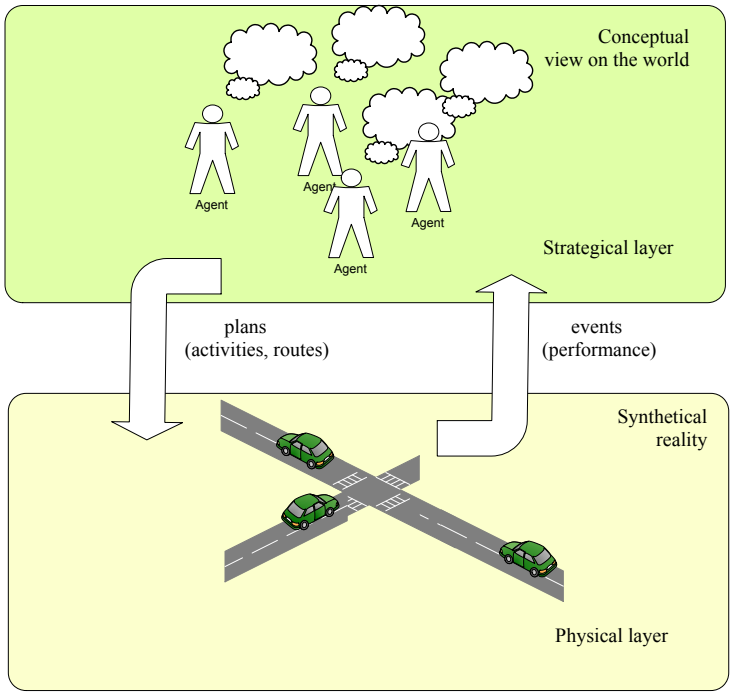


Figure 2.2: The two layers of MATSim

There are several ways to actually implement such DTA, e.g. [103, 20, 17, 80, 81, 7]; most of them deliver time-based OD matrices. For smaller scenarios or telematics applications, there are fully agent-based implementations

[91, 10, 14, 12]. Large-scale scenarios can be computed with TRANSIMS [99]. Other applications for agent-based modeling include land-use models like URBANSIM [104] or ILUTE [87].

The agent-based traffic simulation framework MATSim [15] will be used in this thesis. The traffic assignment in MATSim-T is illustrated in Figure 2.3.

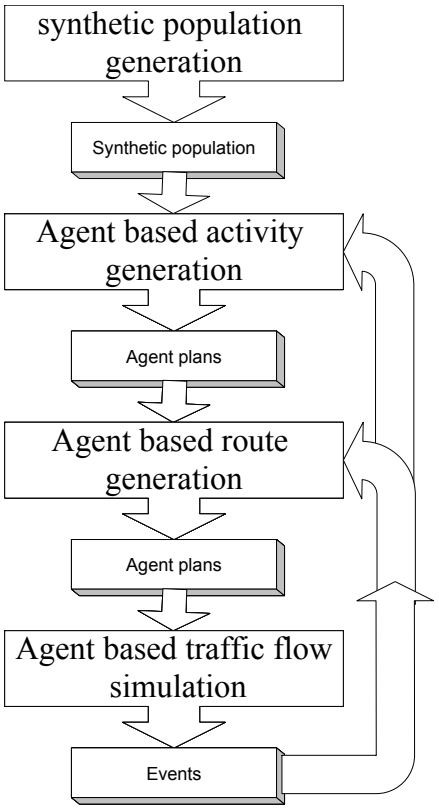


Figure 2.3: Dynamic traffic assignment with agents

MATSim and its physical layer based on the queue simulation [43] will be discussed in the next section.

2.2 The strategic layer

A *Nash equilibrium* [73] is reached, when no agent is capable of improving his or her own plan unilaterally. The agents' plans converge against this Nash equilibrium in an iterative process. This process was already mentioned in Section 2.1.3 and will be discussed further in Section 2.2.2. The strategic layer consists of the agent's mental model of reality and his or her beliefs about the "world". This strategic layer is modeled in MATSim-T as an agent

Listing 2.1: A sample plans file in XML representation

```
<plans name="example␣plans␣file" xml:lang="de-CH"> ...
  <person id="393241" age="37" income="50000">
  <plan>
    <act type="home" link="58" start_time="00:00" dur="07:00"
        end_time="07:00" />
    <leg mode="car" dept_time="07:00" trav_time="00:25"
        arr_time="07:25">
      <route>1932 1933 1934 1947</route>
    </leg>
    <act type="work" link="844" start_time="07:25" dur="09:00"
         end_time="16:25" />
    <leg mode="car" dept_time="16:25" trav_time="00:14"
        arr_time="16:39">
      <route>1934 1933</route>
    </leg>
    <act type="home" link="58" start_time="16:39" dur="07:21"
         end_time="24:00" />
  </plan>
</person>
...
</plans>
```

database plus strategic modules serving the database. That will be the topic of the next section. The database can support the other layer, the "synthetic reality", with a set of plans. Each agent chooses one plan for execution prior to a simulation run. The physical layer then executes the agents' plans and thereby generates *events*. Events are understood as "observations" of what is the actually happening inside the physical world. The events should not interpret the ongoing simulation, but rather merely chronicle it. The events thus reported permeate the strategic layer and serve the strategy manager as decision support. Based on these events decisions can be made to make up new plans or build new mental maps of the witnessed "reality". This process is illustrated in Figure 2.2.

2.2.1 Agent database

The term *microscopic simulation* has been interpreted quite diversely in transport planning. Often, the term "microscopic" only refers to the actual model used to simulate the flow of mobile particles along the network [106] implemented in some simulation package like VISSIM [102].

In the MATSim framework, the individual level of microscopic simulation is preserved in both layers. Every single agent is modeled individually. Each agent's activity chain is modeled for a whole day. This daily plan is defined by an XML representation. Listing 2.1 taken from Balmer, [13]

shows the representation of an agent (person) and one of his or her plans. This also means that plans have to be assigned dynamically over the whole day, including "executing" the activities. However, these activities are not the focus of our traffic simulation and are often implemented as waiting times, after which the agents get inserted into the simulation again. All of the agents needed to execute a meaningful simulation run are stored inside a large agent database. This e.g. read from an XML file containing several plans per agent, including much supplementary information about the agents such as to which household they belong, their gender, income, etc. A selection of plans from the agent database is chosen for execution at every simulation run. This process will be described in the following section.

2.2.2 Co-evolutionary algorithm

As already stated in Section 2.1.3, the MATSim framework uses an iterative process to optimize certain aspects of the agents' plans. Not all information about an agent's daily activities can be extracted from the census data or questionnaires in a meaningful way. The actual route taken from one activity to another is a good example of information that is usually not part of the information provided. Therefore, assumptions have to be made in order to find a feasible way to gather missing information. Given the assumption that every agent will try to find the "best" possible route for his own daily plan, and –as we are modeling a typical weekday– is also capable of knowing the best way to get around, a state of equilibrium must be found in which all agents have the best routed they can find for themselves. The *Wardrop equilibrium* [105] describes such a state for route choice.

In MATSim-T, agents may determine not only their routes, but also their departure times and choice, order and location of their activities. As these factors and the actual degree of success for the agents' daily plans depend on further constraints like street capacities, business hours, etc., the success of an agent's plan can only be diagnosed by executing the plans of all agents simultaneously. The MATSim-T framework uses co-evolutionary algorithms [53, 54] to optimize the daily plans of all agents simultaneously.

A *evolutionary* algorithm consists of these basic steps:

1. Initialize a set of agent's plans, $P(t = 0)$, the generation at time zero
2. Calculate a score or fitness function for P
3. Do a selection of P' from members of $P(t)$: "survival of the fittest"
4. Re-combine (cross over) and mutate P'
5. $P(t + 1) = P'(t)$; increase time step $t = t + 1$
6. Go to step 2 and repeat until some indicator is fulfilled.

Transferred to the realm of MATSim-T, the algorithm reads as:

1. Load or generate an initial set of daily plans for all agents
2. Run the simulation to evaluate the plan's fitness and calculate a score
3. Remove plans with low scores
4. Create variations of the remaining plans for a set of agents
5. Go to step 2.

In MATSim-T there are modules for the different stages of execution of the above algorithm. Namely, there are modules for

- generation of the initial demand
- execution of the physical simulation
- calculation of the score of every executed plan and
- replanning the daily plans of some agents.

The next sections will cover each of these modules except for the initial demand generation module, which was discussed in Section 2.1.1.

2.2.3 Execution of the physical layer

There are many ways to model traffic flow. Some are detailed models like the ones presented by Wiedemann [107] or the model implemented in VISSIM [79]. These detailed models take a rather long time to compute. Given that traffic-system planning is not concerned with detailed driving characteristics, but with congestion formation and spill-back queues and their resolution, a simpler and computationally less expensive model could be employed to serve as the physical layer of our simulation, such as the *queue model* designed by Gawron [43].

Two different implementations of the queue model are available in MATSim-T. One implementation is a re-implementation of the single-core variant of the mobility simulation presented by Cetin [24]. It is written in Java, the host language of the whole MATSim-T project. Therefore, it can be seamlessly integrated into the MATSim-T execution, sparing users the effort of converting data formats or writing and reading back from files. An additional benefit is that it naturally makes use of Java's platform independence, running out-of-the-box under Unix, Windows and Mac OS X.

The simulation is discrete, and each step simulates one second of time. As the main issue is to update two queues per link, the total runtime T_{mobsim} depends on the number of links. It can be estimated with this formula:

$$T_{mobsim} = t_{sim}/\delta t * 2 * s,$$

where t_{sim} is the simulated period of time (e.g., 86.400 seconds for a whole day), δt is the resolution of the simulation (1 second in our runs) and s is the number of links. The execution time is therefore more dependent on the number of links than on the number of agents simulated.

This mobility simulation is the basis for the later implementation of the CUDA variant discussed in Section 3.2. It is therefore described in more detail in section 2.3.

Another implementation of the queue simulation used in MATSim-T is the *DEQSim (deterministic event-based queue simulation)* described by Charypar et al. [26]. It implements a more sophisticated variant of the queue simulation that handles not only the actual FIFO (first-in, first-out) characteristics, but also the backward movement of free link-spaces more realistically. Furthermore, its execution is based solely on the actual events an agent experiences, or, simply put, nothing is computed when nothing is happening in the simulation. Therefore, the runtime of this simulation is not only linked to the number of links in the network but is proportional to the number of agents in the system. The DEQSim is implemented in C++. Therefore, it is not so easily ported between operating systems and an additional overhead must be taken into account for exchanging data between the simulation and the Java-based framework. A native Java version of the DEQSim-algorithm has recently been developed, with impressive runtime characteristics. Unfortunately, as of writing this thesis, there is no reference given for new this implementation.

2.2.4 Evaluation of the fitness of a plan with a scoring function

After executing all plans in the mobility simulation of choice, a score needs to be determined for each plan executed to estimate the actual benefit of the plan for the agent. An individual fitness function describing the actual goal of the agent and eventually possibly its his or her behavior forms the basis of this calculation. One fitness function exemplarily used in conjunction with MATSim-T was described in by Nagel and Charypar [28]. It is based on Vickrey's departure-time choice model [101], but enhanced from dealing only with single trips to handling whole-day plans [82]. The total utility of a plan is described as:

$$U_{total} = \sum_{i=1}^{n} U_{perf,i} + \sum_{i=1}^{n} U_{late,i} + \sum_{i=1}^{n} U_{travel,i} \tag{2.1}$$

where U_{total} being the summed-up gross utility of the plan with n activities, $U_{perf,i}$ is the (positive) total of the utility of the performed activities, $U_{late,i}$ is the (negative) utility of being late and $U_{travel,i}$ is the (negative) utility of travel on trip i.

The positive utility of activities $U_{perf,i}$ is assumed to be of logarithmic shape:

$$U_{perf,i}(t_{perf,i}) = \beta_{perf} \cdot t_{*,i} \cdot \ln\left(\frac{t_{perf,i}}{t_{0,i}}\right) \tag{2.2}$$

Here $t_{perf,i}$ is the actual length of the activity i, $t_{*,i}$ is the wanted duration of activity i. β_{perf} defines the marginal utility of the activity at t_* and $t_{0,i}$ is an additional factor, influencing the importance and the minimum duration of an activity. If the agent is not allowed to drop activities from his or her plan, then the utility for all activities is the same.

The utility for being late is defined as:

$$U_{late,i} = \beta_{late} \cdot t_{late,i} \ , \tag{2.3}$$

with β_{late} as the marginal utility for being late and $t_{late,i}$ as the late arrival for activity i.

Finally, the utility of traveling is defined as:

$$U_{travel,i} = \beta_{travel} \cdot t_{travel,i} \ , \tag{2.4}$$

Again, β_{travel} is the marginal utility of traveling and $t_{travel,i}$ is the time spent traveling during trip i. The marginal utilities are usually described as *Euros/hour* and accordingly, the travel times and delays are described in *hours*.

The utility function describes the behavior and goals of an agent. The agent will search for the best solution in the solution space defined by the utility function. No optimizations outside of the solution domain of the utility function 2.1 can be incorporated by the agent. This has direct impact on the replanning choices that are possible with a given utility function, as will be discussed briefly in the next section.

2.2.5 Replanning modules

This strategic layer is itself split into a set of possible modules for replanning certain aspects of an agent's original plan. This part of the process corresponds to the fitness function described in the previous section. If a new module is added to the replanning capabilities of an agent, the fitness function must be adapted so that it includes the newly generated solution space. On the other hand, it is permissible to use a function that evaluates more than the solution space actually engaged in the current simulation. Therefore, it is permissible to use the function with the provision for departure times given in formula 2.1 of the last section, even though only the routes are varied.

The modules can be roughly divided into modules using random mutation and those trying to utilize the experiences gained in the last iteration of the physical simulation. The latter ones are called "best response" modules, as they attempt to deliver the best response to the information from

the previous run. The random-based modules are often faster in terms of execution speed, but are more likely to need a higher number of iterations before converging.

The most important modules for replanning in MATSim-T are as follows:

- *Time allocation mutator*: A module that randomly mutates the departure times and durations of activities. Because it is an extremely fast module, its actual computing time is negligible.

- *Router modules*: There are three different flavors of router modules implemented in the MATSim-T framework, all based on time-dependent Dijkstra algorithms [36]. The newest is based on a *Landmarks-A** algorithm [63] and offers best performance.

- *Planomat*: Another *best choice* module is the *planomat* module described by Meister [69, 70]. This module is not restricted to varying only certain aspects of the day's plan, but affects all parts simultaneously, including the coordination of the plans of all household members and their joint activities. In addition, the *planomat* module serves as a time-allocation mutator and can substitute above random module.

The open framework of MATSim-T makes it possible to easily add new replanning modules to already existing ones. A wide variety choice of modules has been added to MATSim-T by the community, with features ranging from secondary location choice to the inclusion of social networks [46]. Even schedules of local public transport systems have been incorporated into a module [98].

2.3 The physical layer

In this section, the model behind the Java implementation of the mobility simulation used in the MATSim-T framework will be discussed. The queue model designed by Gawron will be explained as will the modifications implemented for a better reproduction of traffic behavior. As discussed above, the traffic simulation should fulfill some constraints. Namely, it should

- use individual particles or vehicles that are being moved through a network. These should match the agents of our framework

- be a rather simple model so it can be compared to the methods used for static assignment

- be computationally inexpensive in order to limit the computation time needed for the iterative relaxation process

- result in traffic assignments comparable to real traffic counts.

An approach based on queuing theory [94, 93] was chosen. The queuing model was introduced by Gawron and it will be discussed in Section 2.3.1. The following will deal with some drawbacks of the Gawron model and will discuss the addition of some desired features missing in the original model.

A drawback of queue models is their inability to correctly model the speed of the backwards traveling kinematic wave. A singular free link space should travel backwards at about 15km/h [51]. In a queue model a vehicle leaving a link opens a free space immediately. This is visualized rather descriptively in the dissolving of congestion, which in reality would happen from the front of the queue. In a queue model, congestion will dissolve starting from the *end* of the queue. This problem is solved by the DEQSim implementation, which was discussed briefly in Section 2.2.3.

2.3.1 Gawron's queue model

The queue model introduced by Gawron [43] defines three key components for describing a link's attributes:

- Free flow travel time T_0
- Storage capacity $C_{storage}$
- Flow capacity C_{flow}

Each link is given a regular speed (per XML input network file) for the uncongested traveling v_0, a length of the link segment is L, and it is given flow capacity C_{flow} and a number of lanes n_{lanes}.

The above attributes match these readings as follows:

- The free-flow travel time is defined as $T_0 = L/v_0$
- the storage capacity is defined as $C_{storage} = L \times n_{lanes}/l$, where l is a pre-defined space consumption of a single vehicle and
- the flow capacity is given directly as input.

Listing 2.2: The movement code by Gawron

```
for all links do
  while vehicle has arrived at end of link
    AND vehicle can be moved according to flow capacity
    AND there is space on destination link do
      move vehicle to next link
  end while
end for
```

The intersection logic designed by Gawron is given in pseudo-code in listing 2.2. All links are processed in an arbitrary sequence of execution. For each link a vehicle can move to its respective destination link if

1. it is the first vehicle on the source link
2. the link's flow capacity is not exhausted and
3. there is enough space left on the destination link.

The first condition implies that every vehicle that enters the link at time step t will have to stay on the link at least until $t + T_0$. Establishing the second condition is a little bit tricky. The outgoing flow capacity of a link per second will rarely be an integer number. Therefore, the link logic has to deal with the occurring fractions, as the vehicle is seen as an atomic unit and can not be partially sent over the link. There are basically two ways to deal with this situation, and they need only to be employed if there is an actual fractional residual r_{frac}. Either a statistical approach is chosen by selecting a random number $0 < rnd < 1$ and comparing this against the residual, moving a car whenever $rnd < r_{frac}$. The other approach is to add up the residual fraction in an extra capacity counter. If the non-fraction part of the capacity is exhausted, one additional vehicle is allowed to leave the link whenever the extra counter is larger than one. The counter will then be reset. The third condition is met when the number of vehicles on the destination link is less than the space capacity of the link.

2.3.2 Improvements to Gawron's algorithm

Although the Gawron algorithm presented in listing 2.2 is a beautifully simple algorithm, it is not optimally suited to traffic simulation. The fixed sequence of execution running over all links and exchanging vehicles contains two catches. Both involve shortcomings caused by the sequential execution of the movement code for all links. The problems and possible solutions will be discussed in the next two paragraphs.

Fair intersection

One problem intrinsic to the simplistic queue model is that the links are always selected in the same sequence. Hence, certain links are more equal than others, i.e., they will get an earlier chance to get rid of their vehicles before other links can. This leads to unrealistic behavior in congested states, when a small byroad can block out a main street if the byroad happens to be before the main road in the sequence of links after the space of the incoming link has opened up. To resolve this behavior, the vehicles' movement algorithm is changed from Gawron's link-centered approach to a node-based method [24]. By doing this, the links are served by stepping through all intersections of the network and then executing all incoming links for the nodes. The pseudo-code implementation of this can be found in listing 2.3.

Listing 2.3: Fair vehicle movement at the intersections as in Cetin [24].

```
// Move vehicles across intersections:
 for all nodes
   while there are still eligible links
      Select an eligible link randomly proportional to capacity
      Mark link as non-eligible
      while vehicle has arrived at end of link
        AND vehicle can be moved according to capacity
        AND there is space on destination link
          move vehicle from source link to destination link
      end while
   end while  // eligible links
 end for  // all nodes
```

The incoming links of one node (which are the links competing with each other for space on the outgoing links) can then be ordered in a newly defined sequence. To implement a fair intersection that prefers the main roads but still gives smaller roads a chance to issue vehicles from time to time, the Java mobility simulation of the MATSim-T framework uses the following approach: By choosing the sequence of links to be executed with a random draw into these links based on their capacity, it guarantees that links with larger capacity get chosen more often, but smaller links get a chance to update first from time to time. To implement this, the sum of all capacities of the incoming links of a node n which have vehicles waiting are calculated as $inSumMult_n$.

$$inSumMult_n = \sum_{incominglinks_n} capacity_{flow}$$

Then a random number between zero and this sum $inSumMult_n$ is drawn. A loop adds up the capacities of the links and an index pointer i until the sum of the capacities is larger or equal to the random number. The link with the index i is chosen to be executed next. The flow capacity of this link is then subtracted from the $inSumMult_n$, and the link is removed from the list of eligible links. This algorithm is repeated until all links are served. Apparently, in each step the probability for each link to be chosen as the first link is

$$p1_l = flowcap_l / \sum_{links_{remaining}} flowcap_{incoming}$$

This gives larger links a greater probability of being chosen over smaller links, while still maintaining a fair chance for all links.

Listing 2.4 shows a pseudo code version of the algorithm. This version implements the behavior described by the line *Select an eligible link randomly proportional to capacity* in listing 2.3 and it needs to be inserted as a replacement for that line into the code of listing 2.3.

Listing 2.4: Choosing eligible links

```
// this replaces ``Select an eligible link randomly
    proportional to capacity''

// build sum of all link capacities
for all eligible links
 sum_cap += link flow capacity
end for

// draw one random link based on capacities
rndNum = draw random number between 0 and sum_cap
for all links left
  act_cap += actual links flow capacity
  if act_cap >= randNum
    choose link for MOVEMENT
    sum_cap -= actual links flow capacity
    break this loop
  end if
end for
```

Parallel update

The second drawback to Gawron's original algorithm is that it is dependent on the link sequence in terms of the uniqueness of the simulation's run. With Gawron's algorithm the position of a vehicle in the network can change to different new positions in one time step, depending on the sequence of execution of the links. If a link has a free travel time smaller than one time step, then a vehicle that is ready to leave an incoming link to that link can enter the link and leave it in the same time step or has to stay on the link, depending on whether the incoming link is in front of the link in the execution sequence or not. This is clearly an undesirable behavior. The exhibited traffic characteristics should not depend on the purely implementational constraint of the sequence of links. Figure 2.4 illustrates this.

To make the state of the simulation independent of the order of execution, another buffer is introduced. A link buffer capable of fitting the size of the outgoing capacity, i.e., the next integral value larger or equal to the capacity, is constructed. The movement of vehicles is then split into two parts. First, a vehicle moves from the link's main queue (the queue describing the spatial constraints of a link) into the newly constructed buffer, based on the outgoing flow capacity of the link. Vehicles are allowed to move only if the following conditions are met:

- It is the first vehicle in the source link's spatial queue
- The outflow capacity is not exhausted and

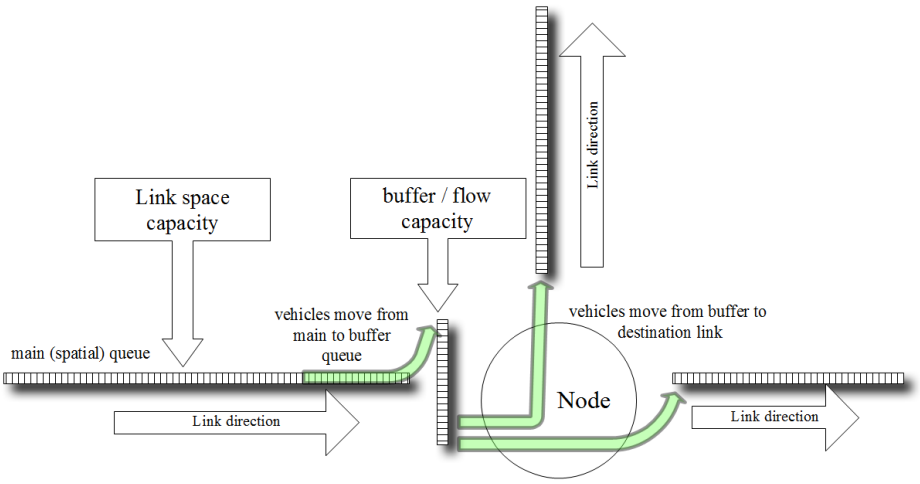


Figure 2.4: Parallel update by including an additional flow buffer.

- There is enough space left on the new buffer, which might still hold vehicles from the previous time step.

The second step is to move vehicles from the new buffer construct onto their respective destination links. This step is only constrained by the amount of free space left on the destination link. The algorithm taken from Cetin [24] is written in listing 2.5.

Listing 2.5: Vehicle movement at the intersections as presented by Cetin. (Note how the algorithm separates the flow capacity from intersection dynamics.)

```
// Propagate vehicles along links:

for all links do
  while vehicle has arrived at end of link
    AND vehicle can be moved according to capacity
    AND there is space \emph{in the buffer} (see Fig~\ref{fig:
        buffer})
      move vehicle from link \emph{to buffer}
  end while
end for

// Move vehicles across intersections:
 for all \emph{nodes}
   while {there are still eligible links}
      {Select an eligible link randomly proportional to
          capacity}
      {Mark link as non-eligible}
```

```
      while {there are vehicles in the buffer of that link}
        {Check the first vehicle in the buffer of the link}
        if {the destination link has space}
           {Move vehicle from buffer to destination link}
           {Proceed to the next vehicle in the buffer}
        else
           {Break the inner while loop and proceed to the next
               eligible link}
        end if
      end while // still vehicles
   end while  // eligible links
 end for  // all nodes
```

It is interesting to note that this addition to the original algorithm makes each link's update independent of all other updates, relying solely on the link and the link's buffer. Likewise, a node's update can be done independently of all other nodes' updates. This will be of interest when the concurrent execution is discussed in Section 3.2.

Chapter 3

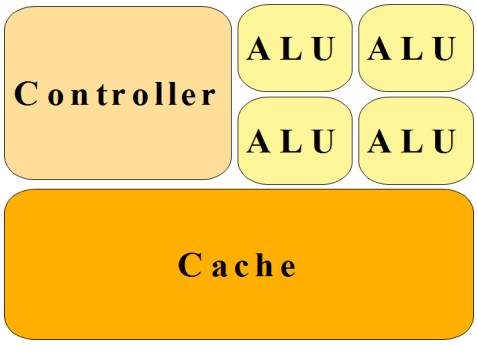
Implementing the queue model on graphics devices

Simulating large-scale scenarios with the queue model described in the last chapter has a high computing demand. Therefore, executing a simulation run with a frame rate high enough for real time interaction is not a simple task. In the work presented in this thesis, this will be achieved by the use of common graphics hardware. This chapter will introduce the hard- and software utilized to accomplish high frame rates. Furthermore, it will compare different implementations and data structures to reveal their usability for our task. The last part of this chapter will concentrate on the necessary tasks for running the actual GPU algorithms on a host computer and for combining the GPU-based simulation with the Java-based MATSim framework.

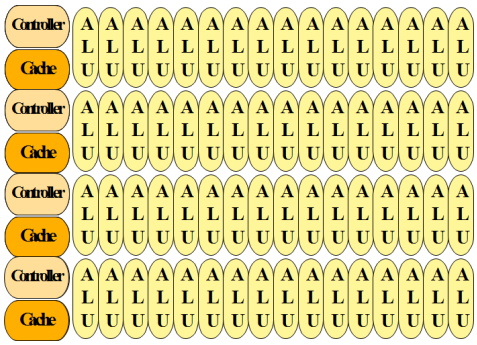
As has been said in the introduction, the goal of this thesis is to accomplish interactive frame rates by using a single affordable desktop computer system instead of high end cluster hardware. There were two possible choices of hardware capable of general purpose calculations, namely graphic cards from either ATI/AMD or NVIDIA.

3.1 CUDA SDK

In this thesis NVIDIA hardware was chosen to implement the queue simulation on. Although both relevant hardware vendors deliver capable hardware of about the same performance, the CUDA SDK was more commonly used at the time of writing the thesis. In this chapter the hard and software architecture of the latest NVIDIA GPU devices will be explained.



(a) CPU scheme



(b) GPU scheme

Figure 3.1: CPU versus GPU layout.

3.1.1 The NVIDIA hardware

The NVIDIA hardware has its roots in the consumer 3D graphics market. Therefore it is not surprising that this hardware has specialized in maximizing the floating point capabilities measured in FLOPS (floating point operation per second). Consumer 3D-graphics has a very high demand for these operations. All 3D to 2D operations in this market revolve around floating point operations, whereas double precision floating point operations are more a domain of scientific research. Therefore, it has been ignored by NVIDIA for a long time. Figure 1.2 gives an impression of the computing power of the latest GPU generations. 3D-graphics rasterization, which is the main duty of these cards, is easily expressed in a parallel manner, resulting in the GPUs to have evolved into multi-core parallel devices. Fig. 3.1 illustrates the different forms of specialization of CPU and GPU.

The CPU must be flexible in terms of flow control and achieves fast memory access mainly by having a sophisticated caching strategy combined with a rather big memory cache. Therefore it physically does not have enough room for many ALUs (Arithmetic Logical Unit), the actual computing units, and has only one central control unit.

The GPU on the other hand does not have a big on-chip-cache to speed up memory access. It uses mainly uncached access to the DRAM and provides several autonomous control units, each of which controls a rather large number of ALUs.

The GPU is especially well suited for problems that can be expressed in data parallel algorithms. This means that there is one algorithm executed in parallel on many different data elements. Again, this is the actual demand of the rasterization process in 3D-graphics. After some preprocessing, large sets of pixel or vertex data can be processed in parallel with the same algorithm in place. But, as described in the last chapter, this type of computational model also fits other domains outside the actual 3D rendering.

Streaming multiprocessors

The NVIDIA hardware groups sixteen ALUs and one control unit to an array of multi-threaded streaming multiprocessors (SMs). Beside these sixteen ALUS, also called scalar processors (SP) or *core*, a SM consists of a multi-threaded instruction (or control) unit, two special function units for transcendental functions and a small block of on-chip shared memory. The multiprocessor can create, manage and execute concurrent threads with zero scheduling overhead. In addition to that, it implements a *barrier-synchronization* intrinsic with a single instruction. These qualities enable lightweight thread creation and managing and efficiently support very fine grained parallelism. With the NVIDIA architecture it is possible to decompose a problem into thousands of threads efficiently.

To manage these many threads executing different instructions or at least different branches of a common instruction list, the SM employs a new architecture called SIMT (single instruction, multiple threads). This expression is chosen to distinguish the new approach from the more usual SIMD (single instruction, multiple data) solutions.

In SIMD architectures multiple ALUs execute the same instruction over a set of input data in parallel. With SIMT, each thread has an individual instruction counter and register set. The multiprocessor maps a thread to each of the eight SPs. This thread is then executed on the SP independently of the other threads, using its own instruction address and register state.

Having the ability to run different code (-branches) in each thread of execution without infringing upon computational correctness of the executed program is a big benefit. Nevertheless, for gaining substantial speed-ups in program execution the programmer will have to use data parallel techniques. The entry level to successfully program the GPU is very low with CUDA, giving the programmer the chance to implement correct code first, optimizing it later. SIMD architectures on the other hand oblige the programmer to follow the data parallel paradigm from the start on.

Warps and half-warps

The execution of threads in the CUDA architecture follows these rules: Each SM unit creates, manages, schedules and executes threads in groups of 32 parallel threads. This group is called a *warp*. All threads of a warp start at the same program address. After that, they are completely free to branch and execute individually. At each clock cycle the SM chooses a thread warp that is ready to execute (i.e. not stalled by waiting for a global memory access) and then executes the next common instruction to all threads that this instruction applies to.

If threads of a warp choose different execution paths by data-dependent branches, then these branches are serialized and executed on the respective threads attending the particular branch. Full efficiency is achieved only when all threads of a warp agree on their execution path. Threads that do not share the same warp run independently regardless of their execution path.

Multiprocessor memory layout

Fig 3.2 shows the overall layout of a GPU device. Apart from the DRAM of the graphics device the multiprocessor has four different memory types on-chip:

- One set of local 32-bit registers per processing unit SP
- A parallel data cache or shared memory block that is shared by all SPs of a multiprocessor. Caching has to be implemented by the threads.

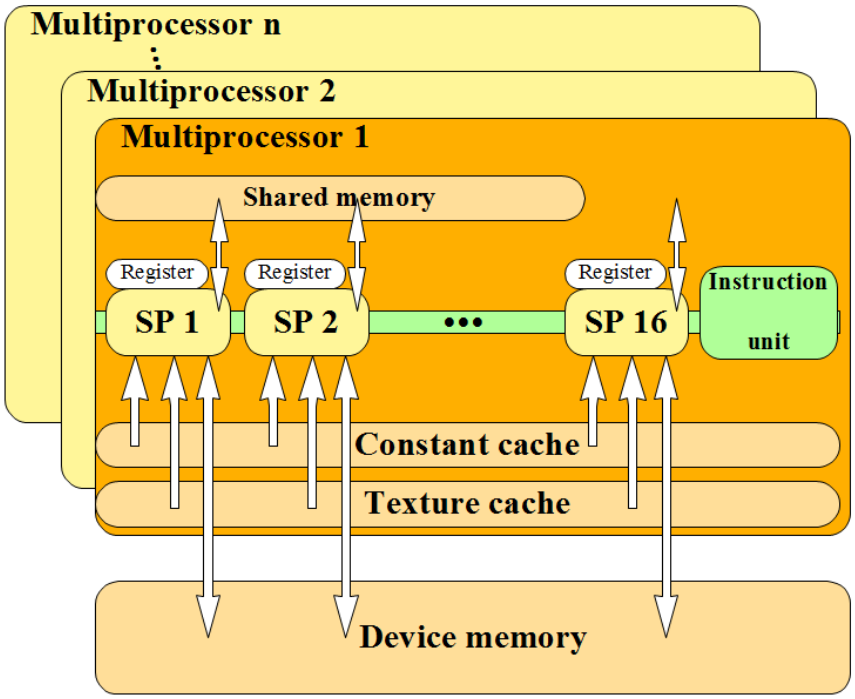


Figure 3.2: Layout of the multiprocessors

- A read-only constant cache that is shared by all SPs and speeds up reading into special constant memory areas of the global memory.
- A read-only texture cache. This is most useful for accessing data in an interpolated and filtered way often needed with textures.

Accessing memory

The general cost for accessing memory differs very much depending on which memory is accessed. Accessing a register will not generally cost extra cycles at all, but delays might occur because of read-after-write dependencies or internal memory bank conflicts. The read-after-write dependency results from a new assignment to a register taking about 24 cycles to be available for reading. This latency can be overcome when a thread count of at least 192 threads per SM is maintained. As 4 threads of the 32 threads of a warp are being executed in each of the 8 SPs, it needs 6 warps ($6 * 4 = 24$) to mask the 24 cycles. That makes $6 * 8 = 192$ active threads per SM.

Accessing the on-chip shared memory is as fast as reading a register, but again with constraints in respect to possible memory conflicts. The shared memory is divided into memory banks. If two threads try to access the same memory bank, these reads have to be serialized. For texture and constant memory the access costs a full global memory access only if the memory access has a cache miss. Otherwise the cost is like one register read for every thread that accesses different constant memory addresses.

For the texture cache the above held true as well, although the cache is optimized for spatial locality. For general purpose programming of the GPU, the texture cache memory is often not easy to adopt in a meaningful way.

The global memory space available for reading and writing is not cached. The only way for the SM to deal with this waiting time is to compute other warps. Therefore it is important to engage a large number of threads, so that there are enough warps of threads waiting to be executed at any time to cover the waiting times caused by global memory access. It takes four clock cycles to issue a memory request. If the memory request reads/writes from or to global memory there will be an additional penalty of 400 to 600 cycles for this operation. This stresses the importance of having enough threads at hand to cover these clock cycles.

In general, memory access is capable of reading 32-bit, 64-bit and 128-bit words in a single instruction. For having an assignment like this

```
__device__ type device[32];

type data = device[index];
```

compile to a single read instruction the `type` has to comply with these rules:

- The `sizeof(type)` must be 4,8 or 16 bytes
- Variables of type `type` must be aligned to `sizeof(type)`

For CUDAs predefined data types the latter condition automatically holds true, for user defined types the could be enforced by using alignment specifiers, namely `__align(8)__` or `__align(16)__`.

For structures larger than 16 bytes the compiler has to generate multiple load instructions. For ensuring to generate the lowest number of load instructions, it is advisable to define those structures with the `__align(16)__` specifier. A variable of the type

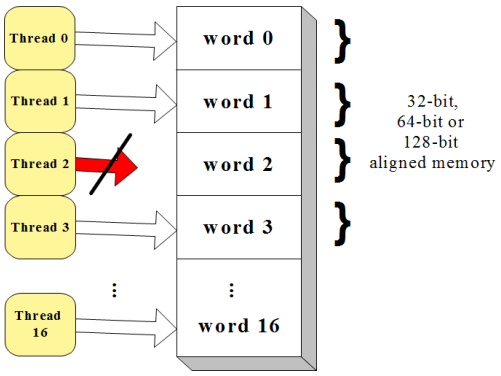
```
struct __align(16)__ \{
long a, b, c, d;
\} long4;
```

will be read using two 128-bit instructions. With the `__align__` the efficiency of single read instruction can be optimized. But there are more opportunities to optimize global memory access when inter-thread relationship is taken into account, as we will see in the next section.

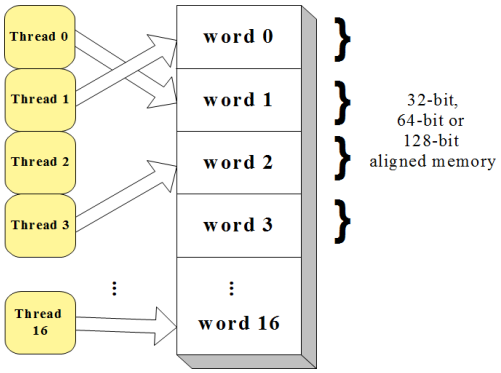
Coalesced memory access versus uncoalesced access

As a memory access to global memory is expensive, optimizing the execution of the code in terms of minimizing the stalled time from waiting for global memory is mandatory. One way to achieve better execution times with respect to memory access is by maintaining coalesced memory accesses. Memory accesses take place in sizes of 32, 64 or 128 bytes. If all threads of a half warp (either the first 16 threads or the second 16 threads of a warp form a half-warp) access data in conformity to the following rules, memory access is performed in a coalesced way.

- All threads access words of equal size. This size might be either a 32-bit word, a 64-bit word or a 128-bit word.
- The read instruction's size depends on the word size and is one 64-byte, one 128-byte transaction and two 128-byte transactions for the aboves sizes.
- All 16 words must reside in the same memory segment (aligned to the size of the respective memory transaction)
- Threads must access the words in sequence, i.e. thread i must access the i-th word.
- Threads might skip reading memory, as long as all reading threads meet the above conditions



(a) Coalesced access



(b) Uncoalesced access

Figure 3.3: Coalesced and uncoalesced memory access on the GPU.

If coalesced memory access is possible one (or two in the 128-bit word case) memory transactions are issued to read the whole 16 words. If these instructions are not being met, up to 16 memory access instructions are necessary. This depends on the hardware revision of the CUDA device in use. This is discussed in more depth in the CUDA Programming manual [31] .

The memory bandwidth of coalesced memory access is two times higher in case of 128-bit accesses (the uncoalesced case will need 16 32-byte accesses, summing up to 512 bytes to be transported, whilst the two 128-byte accesses only send 256 bytes), but about four times higher in case of 64-bit accesses and about an order of magnitude faster in case of 32-bit accesses.

Fig. 3.3 illustrates one example of coalesced and one example of uncoalesced memory access. In the next section actual ways to achieve coalesced memory access in terms of traffic simulation will be discussed in depth. It will be shown that changing a simple data structure can result in noticeable speedup by increasing the amount of coalesced memory accesses.

Technical capabilities of the Geforce G80 series

The most advanced hardware that will be covered by this thesis is the GeForce G80 series by NVIDIA. This chip has 240 processing units grouped in 15 multi-processing units. The chip is credited with having a peak performance of about 1.0 TFLOPS (one trillion single-precision floating point instructions per second) and a memory bandwidth of about 100 GB per second. In comparison to that, the latest Intel CPU *Hapertown* is capable of executing about 80 GFLOPS and has a memory bandwidth of 17 GB/s.

Summary

The latest NVIDIA hardware is based on multiprocessors consisting of a control unit, several different types of on-chip memory and 16 processing units. Collaboration in a multiprocessor is based on a SIMT architecture. This architecture is an extension to more generally used SIMD architectures. It allows each thread running on a processing unit to hold its own instruction counter and register set, enabling each thread to act completely independent from the other threads.

The programmer is not bound to any SIMD-wise constraints if he is not interested in gaining maximum performance, but is only interested in writing correct code. Nevertheless it is more likely that one wants to write code that is optimized in terms of execution speed. In this case global memory access is the main bottleneck. This can be overcome by obeying certain rules like alignment of data structures and maintaining coalesced access. The sheer computing power of the GPU is impressive, but to what extend this power is utilizable for the queue simulation is the question still to be answered.

3.1.2 The CUDA SDK software API

Given the above hardware, NVIDIA needed to come up with an API (application programmer interface) to make the capabilities of the hardware accessible to programmers. OpenGL and DirectX provide this accessibility for graphical operations, but are not designed for general purpose computation. Only lately, both big graphic device vendors came up with SDKs intended to support the programming of more general problems on GPUs.

ATI/AMD introduced an extended version of a programming language developed with massive multi-threading in mind, namely the BROOK language [21], named BROOK+ [9].

NVIDIA voted against a new language and extended the C-language by some new keywords and syntax constructs instead. By resorting to a well known language like C, NVIDIA tried to keep the learning curve as low as possible. This design decision makes it very easy for programmers already capable of programming C or C++ to enter the realm of GPU programming. Syntactically, there are just a few additional keywords to be learned and restrictions to be considered. This programming model will be described in more detail in the next section.

CUDA Computing model

There is basically only one new paradigm that the C-programmer has to internalize to start programming the GPU: *The kernel*. A *kernel* is a specially marked C-function. This function, when called, will be executed on a large number of parallel threads, simultaneously, but on different data. To define a kernel in CUDA, the declaration specifier `__global__` is used. Additionally, the programmer has to declare how many threads to start in parallel with this kernel. This is specified by a new syntax using triple brackets `<<<...>>>`.

A simple example in which N float numbers from a float array A are added to numbers stored in an array B and the result is stored in array C like this:

```
// kernel definition
__global__ void vecAdd( float* a, float* b, float* c)
 {
  int i = threadIdx.x;
  c[i] = a[i] + b[i];
}
...
int main()
{
  // kernel invocation
  vecAdd<<<1,N>>>(A,B,C);
}
```

Each of the N threads that execute the `vecAdd` kernel is given a unique *threadID* that can be accessed inside the kernel by the use of the internal variable `threadIdx`. In this example each kernel is using this `threadIdx` to access one specified element in the array given and adds the array elements pairwise.

Apart from kernel declared with the keyword `__global__`, the API distinguishes two more function types. Functions marked with `__device__` are functions that solely run on the GPU. These functions can only be called from kernels. Functions marked as `__host__` are run on the host machine. If a declaration specifier is omitted, `__host__` is implicitly assumed. The keywords `__device__` and `__host__` can also be combined, in which case the compiler generates two versions of this function, one to be executed on a CPU and one for GPU execution.

There are some more restrictions for functions declared either as `__device__` or `__global__`:

- They do not support recursion.
- They cannot declare static variables inside the function body.
- They cannot use variable argument numbers.
- Function pointers to `__device__` functions are forbidden, to `__global__` allowed.
- The maximum size of the argument list of a `__global__` function is 256 bytes.

A grid of thread blocks

The internal variable `threadIdx` is designed as a vector (x, y, z). This makes accessing data organized in either two or three dimensional array more convenient. Threads within this index description form an one-, two- or three-dimensional *block of threads*. This provides an easy and natural way to invoke multi-threaded computation of elements structured in domains of vectors, fields or matrices. A simple example would be the above addition extended to a two-dimensional array.

```
// kernel definition
__global__ void arrayAdd( float a[N][N], float b[N][N], float c
    [N][N])
 {
  int i = threadIdx.x;
  int j = threadIdx.y;

  c[i][j] = a[i][j] + b[i][j];
}
```

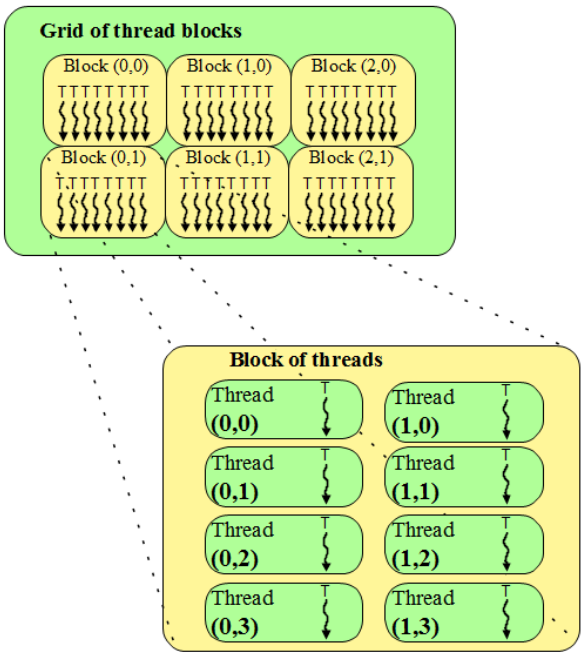


Figure 3.4: The grid of blocks of threads defined by $<<<(3,2),(2,4)>>>$

```
...
int main()
{
  // kernel invocation
  dim3 dimBlock(N,N):
  arrayAdd<<<1, dimBlock >>>(A,B,C);
}
```

The actual index of a thread is the x-component in the one-dimensional case. In the other cases it is straightforward. For a two-dimensional block, the thread ID of index (x, y) is $x + y * D_x$ and in three dimensions it is likewise $x + y * D_x + z * D_x * D_y$. The dimension (D_x, D_y) can be accessed through the built-in variable `blockDim`.

Threads within a block can cooperate. They can exchange data by using the multi-processor's shared memory and synchronizing their execution. This is achieved by setting explicit synchronization points in the kernel by using the special function `__syncthreads()`. This method acts as a barrier that all threads of a block of threads have to reach before execution is resumed behind this point.

This synchronization mechanism and shared memory access is only possible for threads within one block of threads. The number of threads within one block of threads is restricted. In the NVIDIA G80 series the maximum number of threads in a block of threads is 512.

However, a kernel call can be executed by multiple equally shaped blocks of threads. The total number of executed threads is then the number of threads in a block of threads times the number of blocks the kernel is run on. These blocks are organized as a *grid of thread blocks*. This grid can be one- or two-dimensional. The dimension of the grid is specified by the first parameter of the `<<<...>>>` syntax. A grid of thread blocks is illustrated by Fig. 3.4.

Each block index within the grid will be accessible from within the kernel call by the intrinsic variable `blockIdx`. The dimension (D_x, D_y) of the grid of blocks can be accessed through the build-in variable `gridDim` and the *threadID* within the grid of block is constructed following the same pattern as above. Therefore, though only up to 512 threads can make up one block of threads, the grid of blocks can run thousands or even trillions (the actual upper limit of individually indexable threads being $65536 \times 65536 \times 512$) of threads in parallel.

Thread blocks are required to execute independently. Depending on the number of cores, they might be executed in parallel or in series. Communication between two threads in different thread blocks is only possible through global memory. There is no synchronization mechanism between blocks respectively, or between threads in different blocks. If two instructions need to be synchronized between all threads, these must be put into subsequent kernels.

Choosing the values for grid and block sizes depends on the problem domain. It might be necessary to synchronize certain sets of threads. These need to remain within one block of threads.

But, as all threads of a block have only one small set of shared memory at their disposal, it might be necessary to cut the number of threads in a block down, if the kernel needs larger amounts of shared memory. The function arguments for the kernel call also remain in shared memory, reducing the available shared memory even more.

Memory types of the API

Reflecting on the numerous memory types of the hardware, there are several ways to define memory types in the CUDA API.

- `__device__`: declares a variable residing on the GPU device.
  - Resides in global memory space.
  - Has the lifetime of the application.
  - Is accessible from all threads; is accessible from host by API calls.
- `__constant__`: declares a variable that:
  - Resides in constant memory space.
  - Has the lifetime of the application.
  - Is accessible from all threads; is accessible from host by API calls.
- `__shared__` declares a variable that:
  - Resides in shared memory space of a thread block.
  - Has the lifetime of the block.
  - Is accessible from all threads within the same block.
  - Writes are guaranteed to be finished only after execution of `__syncthreads()`.

Compiler for CUDA

Extending the C-language by additional symbols eventually brings the need for an appropriate compiler. NVIDIA's CUDA compiler is called *"nvcc"*. The *"nvcc"* compiler's role is twofold. First of all, it acts like a preprocessor for the CUDA-related source code files. It splits the code that is to be run on the GPU from the code run on the host. The GPU related code is then translated, and new files in regular C are created. In these C-files the CUDA-device code (called PTX code) is embedded as binary data. Additional code for loading and starting the CUDA binary object is also included. In the latest release of the CUDA SDK, PTX code compilation can be triggered on-the-fly through the driver API.

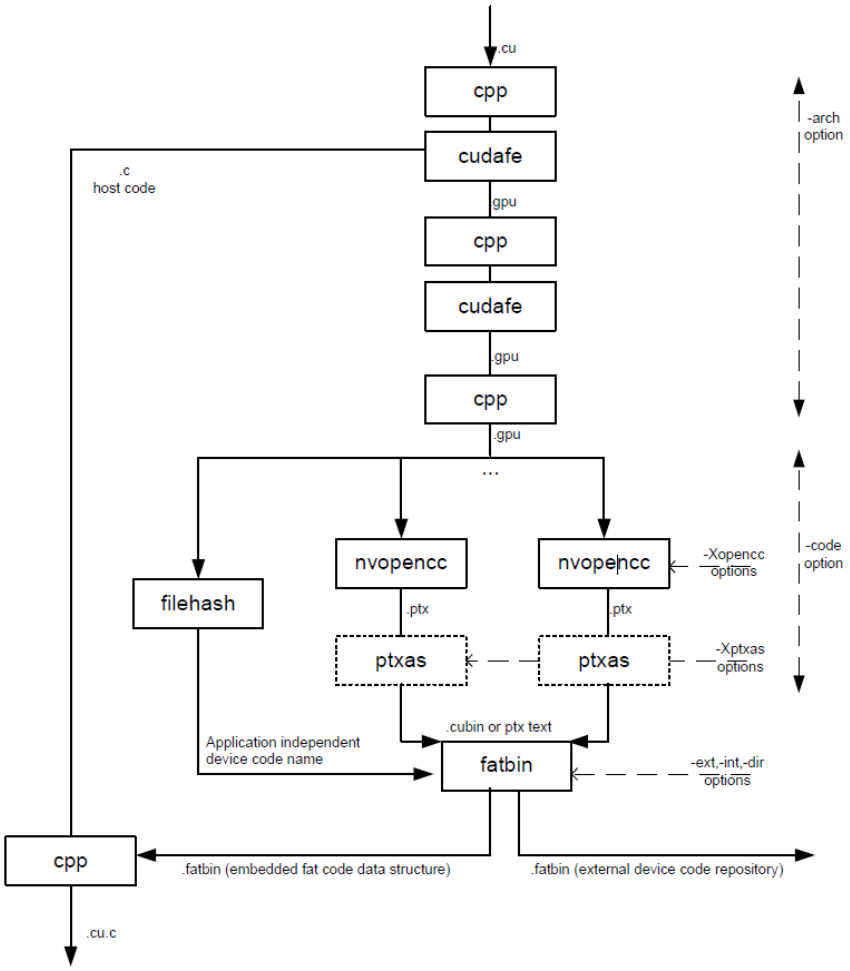


Figure 3.5: The nvcc compilers compilation path (taken from [31])

For preprocessing and compiling the C/C++ code NVIDIA uses a modified version of the open-source C++ compiler *Open64* mainly developed with Intel funding, but currently hosted by Hewlett-Packard and the University of Delaware. The translation process is illustrated by Fig. 3.5.

Atomic operations

As explained earlier, threads which do not run in the same block of threads have no way to synchronize. Mutual access of the same global data is not synchronized. If two threads write to the same address in global memory, the sequence of writing is undefined. If one thread needs to read data and write it back in a synchronized way it needs to use *atomic operations*. NVIDIA's SDK offers a set of atomic operations which guarantee interference-free read-

modify-write cycles. These atomic operations are rather expensive in terms of execution time, but nevertheless useful in some situations that would otherwise result in undefined race conditions. Atomic operations operate on integer values only. Atomic operations include addition, increment, exchange and a *compare and swap* operation reminiscent of C's `? :` operator. Also, the bitwise operations AND, OR and XOR are implemented in an atomic version.

OpenGL and DirectX interoperability

Given that CUDA code executes on data that resides in the graphic devices RAM anyway, it suggests itself to offer access to this data through the drawing interfaces. *OpenGL*, supervised by the Kronos Group, and *DirectX*, developed at Microsoft, represent the relevant 3D Graphics APIs. Both can interact with CUDA memory blocks. CUDA provides necessary API functionality to read and write from and to DirectX and OpenGL resources, making it easy to modify graphics data with CUDA routines or to visualize CUDA results by exchanging data directly on the graphics device. This functionality will be used in implementing the visualizer for the CUDA traffic simulation.

Asynchronous execution

A kernel execution started with the `<<<...>>>` syntax will resume CPU code execution immediately. This means that CPU and GPU code execute in parallel. The kernel calls will be queued for execution and will not run in parallel. To synchronize CPU and GPU execution the API provides the function `cudaThreadSynchronize()` to make sure all kernel calls have finished their execution. Memory transfer functions between CPU and GPU also synchronize both devices. `cudaThreadSynchronize()` should be used conservatively as it might hold the execution layer from choosing the best execution strategy. But for benchmarking subsequent kernels it might be a good choice.

3.1.3 Theoretical peak of calculation speed and memory bandwidth

In this section, some simple benchmarks will be presented to get an impression of the capabilities of the GPU devices in terms of memory bandwidth as well as computational power. The next section will present three different approaches for benchmarking the performance of either CPUs or GPUs. The next section will present results from benchmark runs on different architectures.

Listing 3.1: Kernel implementation for the *triad* operation

```
// kernel definition
__global__ void arrayAdd( float* a, float *b, float *c, float
    scalar, int N)
 {
    int index = __mul24(blockIdx.x,blockDim.x) + threadIdx.x;

    if(index >= N) return;
    c[index] = a[index] + scalar * b[index];
}
```

Some standard problems

The three different benchmarks used in this section will try to get a wide overview of different figures addressing different areas of performance. The first benchmark used will be a simple copy instruction on the GPU device. This instruction will therefore present the sheer bandwidth of the GPU device when moving parts of the memory. This will give us an impression of the peak performance possible with the different devices, but is not necessarily a figure that is of any use in real life. Two more benchmark applications were chosen with having the computational power of the GPU in mind as well as a mix of computation and memory access. The latter one is based on the STREAM benchmark [67] published by John D. McCalpin of the computer science department of the University of Charlottesville, Virginia. This STREAM benchmark has been run on a wide range of computers and supercomputers. It includes four different tests which are mainly targeted at memory throughput. The four operations implemented for the benchmark are *copy*, scalar-*multiply*, *add*, *triad* (add and multiply). Small changes had to be made to adopt the original STREAM code to the CUDA SDK, namely kernels had to be defined. One such kernel can be found as an example in listing 3.1.

The other benchmark chosen is an implementation of the Black-Scholes-Model introduced in [18] in 1973. It describes the market for an equity with the equity's price being modeled as a stochastic process. From that model follows a partial differential equation (PDE), which again can be solved for e.g. a European call option. The resulting formula is called the Black-Scholes formula. Apart from being a Nobel Prize worthy formula, its benefit in terms of benchmarking lies in its good measure of memory access and mathematical calculations. This last factor distinguished it from the other chosen benchmark, the STREAM code. The steam benchmark does not really test the mathematical functions of the device, except for addition and multiplication, whereas the Black-Scholes solution relies on exponential and logarithmic functions, making good use of the transcendental functions

units integrated in each MP, as described in Section 3.1.1. Therefore it probably is a benchmark where a smart CUDA application can shine, as mathematical functions are one of its advantages. Implementations of the Black-Scholes model can be found in a huge variety of computer languages at [108]. We will not discuss the actual algorithm in this thesis, as we only use it as a benchmark, but the implementation is given in listing 3.2 taken from the NVIDIA CUDA SDK samples [31].

Benchmarking of standard problems and discussion

	GeForce 8600GT	GTX 280	CPU Pentium
Bandwidth	7638 MB/s	120 GB/s	1068 MB/s
STREAM *Copy* (double)	1955 MB/s	118 GB/s	2422 MB/s
STREAM *Scale* (double)	1948 MB/s	118 GB/s	2123 MB/s
STREAM *Add* (double)	1974 MB/s	121 GB/s	2285 MB/s
STREAM *Triad* (double)	1965 MB/s	120 GB/s	2292 MB/s
STREAM *Copy* (float)	7797 MB/s	127 GB/s	5089 MB/s
STREAM *Scale* (float)	7699 MB/s	126 GB/s	4595 MB/s
STREAM *Add* (float)	7484 MB/s	127 GB/s	3845 MB/s
STREAM *Triad* (float)	7413 MB/s	128 GB/s	3621 MB/s
Black-Scholes	763.85 MO/s	8165.94 MO/s	6.62 MO/s

Table 3.1: Benchmark results (MO = Mega Options)

The results of the bandwidth test in Table 3.1show how badly the passively cooled GeForce 8600GTX performs in terms of memory bandwidth. The sheer numbers have to be divided by the number of SPs (or cores) to compare both CUDA devices with each other, as every core has its own memory link. But still the memory of the older card performs only half of the GTX280 with 238 MB/s per core against 500 MB/s per core. Apart from that both devices clearly outperform the CPU in terms of memory bandwidth. The STREAM suite of tests basically underlines the performance of the raw memory bandwidth tests. It is also rather obvious that NVIDIA overhauled the double precision rendering for the GTX280. Performance of the STREAM test with *double* values can compete with the single precision results on the new architecture, but is basically half as fast on the older revision of the graphics device. The Black-Scholes algorithm, which combines the memory throughput with some elaborate calculations, finally makes the GPU device really shine. The Black-Scholes implementation runs about 115 times faster on the GeForce 8600GT than on the CPU. The CPU implementation does only engage one of the two cores. The GTX280 runs the algorithm about 1200 times faster than the CPU. This implies that even one single core of the GPU manages to run the calculation five times faster

Listing 3.2: Kernel implementation for the *Black-Scholes* formula for European options

```
// Copyright 1993-2007 NVIDIA Corporation. All rights reserved.
// Polynomial approximation of cumulative normal distribution
    function
__device__ inline float cndGPU(float d){
    const float       A1 = 0.31938153f;
    const float       A2 = -0.356563782f;
    const float       A3 = 1.781477937f;
    const float       A4 = -1.821255978f;
    const float       A5 = 1.330274429f;
    const float RSQRT2PI = 0.39894228040143267793994605993438f;
    float  K = 1.0f / (1.0f + 0.2316419f * fabsf(d));
    float cnd = RSQRT2PI * __expf(- 0.5f*d*d) *
        (K *(A1 + K*(A2 + K*(A3 + K*(A4 + K * A5)))));

    if(d > 0) cnd = 1.0f - cnd;
    return cnd;
}
// Black-Scholes formula for both call and put
__device__ inline void BlackScholesBodyGPU(
    float& CallResult,
    float& PutResult,
    float S, //Stock price
    float X, //Option strike
    float T, //Option years
    float R, //Riskless rate
    float V  //Volatility rate
){
    float sqrtT, expRT, d1, d2, CNDD1, CNDD2;

    sqrtT = sqrtf(T);
    d1 = (__logf(S/X) + (R + 0.5f*V*V)*T) / (V*sqrtT);
    d2 = d1 - V * sqrtT;

    CNDD1 = cndGPU(d1);
    CNDD2 = cndGPU(d2);

    //Calculate Call and Put simultaneously
    expRT = __expf(- R * T);
    CallResult = S * CNDD1 - X * expRT * CNDD2;
    PutResult  = X *expRT *(1.0f - CNDD2) - S *(1.0f - CNDD1);
}
```

than the CPU. This has to be attributed to on the one hand the good memory bandwidth but also the optimized implementation of the mathematical functions used here. The CPU version was not optimized for any special instruction set, other than the with the use of the regular *-O3* flag which does some optimization.

To sum things up, the above benchmarks do show impressively what the GPU devices are capable of, given the right problems. Unfortunately, these high figures cannot be expected when dealing with the implementation of the queue simulation. As we will see in the coming section, random access to memory comes with a rather severe performance penalty, but is nevertheless at the core of the queue algorithm. So it remains an interesting question if the GPU device is nevertheless capable to speed things up in this domain. If a way is found to exploit the power of the GPU in terms of traffic simulation, further generations of graphics devices will –again– yield a free performance dinner, so exploring this new road might be fruitful even for generations yet to come.

3.2 The queue model transferred to CUDA

The algorithms needed to implement the queue model for transport simulation were discussed in Section 2.3.1. In this section, a parallel version of these algorithms will be implemented to run on a GPU device. Different variations of implementation will be derived and tested for their performance.

3.2.1 Parallelize the queue model

To parallelize the queue model's algorithm found in listing 2.5, it needs to be dissected into distinct parts which can safely be run in parallel. As there are two main loops that run over all links and all nodes respectively, these are possible candidates for parallelization. Figure 2.4 illustrated the link update process. As one can easily see, there is only the link's storage and the buffer of the same link involved. No other memory area needs to be accessed. Therefore, it is safe to do this update for all links in parallel. For the `moveNode()` method, we can see that all links contribute solely to one particular node as incoming link and to one particular node as outgoing link. Only this particular node touches buffers of its incoming links, therefore no conflicting, parallel access to these buffers can take place when running all nodes concurrently. Also, all links occur as outgoing link of at most one link, so no link's storage is accessed by more than one node.

As seen, it is safe to concurrently run all link updates and all node updates and still maintain a deterministic execution of the simulation. The execution of these two updates should be sequential, though. If we run node and link updates concurrently, it is indeterminate whether the node or the link updates a certain buffer first, resulting in different outcomes.

When transferring the algorithm onto the GPU, two kernel calls are necessary to achieve the above deterministic parallel execution of the code. One kernel will update the buffer of all links in parallel and after that another kernel is responsible for executing the node's movement code for all nodes concurrently. The pseudo-code for the general movement loop will look like listing 3.3.

Listing 3.3: Pseudo code for queue movement

```
void simstep() {
  for all links do {
    moveLinks(time);
  }

  for all node do {
    moveNodes(time);
  }

  time++;
}
```

Simulation code without node logic

This section will start with an even simpler approach of executing the simulation loop than the one implemented by the queue model itself.

The basic flow of vehicles in the queue model is from links to buffers and from buffers back to links, as seen in Section 2.3.1. The flow from the buffer to another link is managed by the code that implements the node logic. In a first and very simple implementation of vehicular flow, this code might be replaced by moving the vehicles directly from a buffer onto the next link. This means completely omitting the notion of a node, solely relying on the links and the link's buffer. The next link is given by the vehicle/agents plan. In this first approach to the queue model the `moveNode()` method of the algorithm listing 3.3 is replaced by a simpler version acting directly upon all buffers. This method is called `moveBuffer()` and a pseudo-code listing is found in listing 3.4.

Following the discussion in the last section, it is obvious that this implementation of `moveBuffer()` which runs over all buffers in parallel bears one major drawback: As all incoming links of a node compete for space on the outgoing links in parallel, a deterministic execution is no longer given. Additionally it is necessary to install mechanisms to avoid that one buffer overrides data which another buffer has already written, as they access the outgoing link's storage concurrently. This can be achieved by using CUDA's atomic operations, as will be shown in more detail in the discussion of the data structures used.

Listing 3.4: Pseudo code for buffer movement

```
void moveBuffer() {
  while (buffer is not empty)
  {
    dest = destination link of first veh
    if( dest.hasSpace())
    {
      move veh on top to destination link
    } else {
        // if first veh cannot leave, none can
        break and return;
    }
  }
}
```

f1 | f2	d1	d2	d3	d4	d5	d6
ACT	f	f	f	f	f	f
Flag	←	24-bit data				→

Table 3.2: Data format for plan chunks

3.2.2 Activity chains

The Population sample used for a simulation run consists of a list of Objects of class `Person`. This `Person` object holds information about this particular person, e.g. age or sex, as well as the plans of this Person.

A `Person` can have more than one plan, but only one plan of these is marked *selected*. Only the selected plan will be executed in the physical mobility simulation.

Therefore only this plan needs to be reproduced in the GPUs main memory. As the GPU memory is a tight resource, a representation with a preferably small memory footprint must be found.

The daily plan of an agent consists of a sequence of activities the agent plans to attend. These activities are intermitted by the need to travel from the location of one activity to the location of the following activity.

Each piece of information of a person's chosen plan is stored in a single 32 bit integer value. The upper eight bits of information are being used as a flag to indicate the actual meaning of the information bit. The remaining 24 bits of the integer value are left for additional data. Table 3.2 shows this.

There are three different types of data defined: ACT, LEG and ULEG. The following bit combinations were chosen for the three flags: ACT(0x01), LEG(0x02) and ULEG(0x03). The following sections will discuss the meaning of each flag. At the end of this section it will be discussed how one can build the representation of a whole day plan from these primitives.

Data structures for activities

Per definition, each daily plan starts and ends with an activity taking place at the same location, most often this location is "home". Each activity has an activity type. Examples are "home", "work", "leisure" or "school". Each activity has a defined end time, when the agent will finish that particular activity. Two activities might take place at the same location; in that case, no travel is necessary between these two activities. Integer values preceded by the ACT flag represent a certain activity taking place. Activities have two data items relevant for the execution of the simulation. The end time of an activity is the one information needed and the *LinkId* of the link on which the activity takes place is the second mandatory information. The activity's location can be found in the first LEG or ULEG entry following an activity. One integer value is used for storing the activity's end time. The activity's type is not of interest for the mobility simulation; therefore this data is omitted in the reduced representation of the plan.

The data bits of an ACT item indicate the end time of this activity in seconds. As there are 24 bits left, an activity can last for a maximum of nearly 16 million seconds (the data token 0xffffff being reserved for a special activity) or roughly 194 days. The final home activity has the value `ACTEND == 0x01ffffff`, being the ACT Flag combined with the special value 0xffffff as data.

Data structures for Routes

The class describing a travel between activities is called a leg. This leg class contains amongst other information the mode of transportation (e.g. car, pt (public transport), bike) as well as the actual route taken. In this simulation, we will only execute legs with mode choice "car". Any other mode will be handled as if the agent has been teleported from one activity to its succeeding activity.

When a travel time is given for a non-"car" mode, this travel time is being waited for before another activity is started.

The data bytes of a LEG indicate which *linkID* to travel next on this leg's route. As the data region of our integer value is only 24 bits wide, our network is constrained to a maximum of 16 million links. Each LEG route is complete in the sense that it contains all links necessary for the car to travel along including the start and end link.

A LEG is only used for "car" mode travel. With any other mode, the ULEG data flag is used. ULEG routes consist of the start links as the first entry of the route, followed by a special entry which has a time difference as data. It is the time difference given in the travel time duration member of this particular leg. Finally the destination link's ID is given as the third entry of the ULEG route. Although the ULEG "teleportation" would not

need the start or end link IDs, as it never enters the link's buffer or queue but just gets from one activity to the next, we supply this information to be able to send more meaningful information to the framework as will be discussed in Section 4.4.2.

A complete plan

The activity's location is stored in the first LEG or ULEG information of the route following this activity. If two activities are on the same link, these two activities follow in direct sequence, without an extra LEG between them. A "car"-mode route consists of a sequence of LEG data.

A complete plan consists of at least two activities. It might contain more iterations of an activity followed by either three ULEG entries or a sequence of LEG data. If there is a LEG or ULEG data immediately behind an ACT data it will be the link this activity took place on.

Caused by this information reduction, the simulation is not capable of determining the actual location of any daily plan that wholly takes place on one link. In this case the Java framework has to supply the additional information from the agent's plan, as will be discussed in the Section 4.1 about the event generation.

See Fig. 3.6 for an example plan translation from its XML representation to the data chunks used on the GPU.

3.2.3 Network data

Apart from the plan's data, describing the daily activity chain of every agent in the simulation, the network upon which the activities and routes are executed must be read. A (street) network in MATSim consists of links and nodes. The links represent the streets of our network, whereas the nodes represent the street crossings. A link represents one side of the street including all lanes. The opposite direction is modeled by another link.

In addition to the links and nodes, a third data structure is needed to implement the queue model: each link has an additional buffer to store vehicles which are ready to go to the next link.

Links

As seen in Section 2.3.1, in order to a link in the queue model, we need certain information. The `freeLinkTravelTime` is the time a vehicle needs to travel a certain link in the uncongested case. The `spaceCapacity` of a link holds the upper limit of cars being on a link at the same time. It is calculated as

$$space_{link} = length * lanes/carsize$$

```
<plan>
  <act type="h" x="-25000" y="0" link="1"
      end_time="06:00" />
  <leg mode="car">
    <route>2 7 12</route>
  </leg>
  <act type="w" x="10000" y="0" link="20"
      end_time="17:00" />
  <leg mode="car">
    <route>13 14 15</route>
  </leg>
  <act type="h" x="-25000" y="0" link="1" />
</plan>
```

(a) A sample plan

```
ACT 0x005460
LEG 0x000001
LEG 0x000002
LEG 0x000007
LEG 0x000012
LEG 0x000020
ACT 0x00EF10
LEG 0x000020
LEG 0x000013
LEG 0x000014
LEG 0x000015
LEG 0x000001
ACT 0xffffff
```

(b) A data representation

Figure 3.6: Translation of a XML Plan into the GPU structure.

`Car size` is set to $7.5m$ in all examples executed within this thesis. The final attribute missing is the `flowCapacity`, which is the maximum number of vehicles that can leave a link in a given time step. It is used to determine the size of the buffer needed for every link and is calculated as follows:

$$size_{buffer} = capacity_{flow} * \Delta t_{timestep} / \Delta t_{flowperiod}$$

The $capacity_{flow}$ and $\Delta t_{flowperiod}$ is given by the network XML file, $\Delta t_{timestep}$ is 1 second in all runs performed here. Another information needed for describing a link is which two nodes this link connects, namely the $fromNode$ and the $toNode$. A link does not hold any geographical information about its placement or orientation. This information has to be derived from the appropriate node's data.

Nodes

The nodes hold information about their actual coordinates. This information is only used for visualization, not to extract link length from, as the actual length of a link might differ from the Euclidean length between the two nodes. The node's coordinate system might not allow length calculation, or the actual street represented by the link might be winding. Albeit this information is not given explicitly by the network's XML file, all incoming links of a node will be calculated and stored with every node. As this information will be needed at every time step, it is pre-calculated so as to not having to search for it while the simulation is running.

3.2.4 Administrative structures

All links and nodes are given a new internal ID after being read from the XML file. The original ID is stored in a map along with the new ID to identify these links and nodes for event writing later on. The new ID is an integer number starting from zero to $max_{link} - 1$ or $max_{node} - 1$ respectively. This guarantees that the internal IDs are contiguous and confined.

As seen above, the maximum sizes for all buffers and link spaces necessary to store the vehicles en route is known. To avoid memory segmentation, all vehicle data cells for the link space and all vehicle data cells for the buffers will be allocated in one big chunk of memory. Additionally, we define administrative data structures to hold the bounds of the actual link space or buffer queue. These administrative structures will give us a start and an endpoint for every link or buffer and the actual assignment.

As no link can be an incoming link of more than one node, the link data can be sorted by their respectively $toNode$. Then another additional data structure for the nodes is used that points to the first and the last incoming link of a given node. An order might get applied to the links within one node. At first, the links will be sorted by their relative flow capacity. In

order to check all incoming link buffers for vehicles ready to leave the link, each node can step from its first link to its last link in sequence serving them in order of their capacity or shuffle these indexes to serve the links in changed order.

Choosing the implementation of these administrative structures will greatly impact the actual performance of a simulation run. Different implementations of data structures will be described in the following sections. Finally the performance measured with the different implementations will be discussed.

3.2.5 Array of structs

Coming from a CPU based environment, where dynamic data structures like the STL-library or Java's collections are used to store our data, the obvious choice is to use an array of a data structure (AOS = array of struct) combining the necessary administrative information. Such a *struct* might be defined as follows:

```
QueueAdmin{
  int pos;
  int start;
  int end;
}
```

This structure is illustrated in Fig. 3.7. For every link it contains information about the starting position of this link's data in the large vehicle array as well as the end position and the insertion point for the next vehicle. The position member tells us how many of the queue's slots are actually filled with cars right now. If *pos* == *start*, as it is at the beginning of the simulation, there is no car in the queue; otherwise, all cells from *start* to *pos* are filled with car data. The integer values *start*, *pos* and *end* refer to indexes inside of the large car data array.

If a vehicle gets added to the link or buffer, it is inserted at the position pointed to by the *pos* member of the struct, and this member is increased by one to point to the next free space. If, on the other hand, a car has to be removed from the start of the queue, all remaining cars have to be moved one position nearer to the start and the *pos* pointer must be decremented to point to the now empty slot.

Apart from the benefit of an easy and straightforward implementation, this struct bears two main disadvantages. First, moving all vehicles in the queue every time the first vehicle gets removed is costly in terms of performance. Second, the data structures are not aligned in such a way that coalesced access to this data is possible. Each thread executing the link movement code will at least have to access its *pos* and *end* member to check whether any cars are in the buffer. For a given index i and a size of 4 bytes for an integer variable, this will access byte positions 0 and 8 in the above

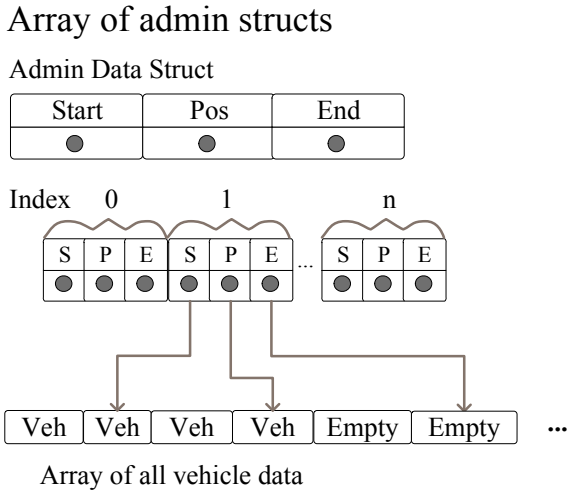


Figure 3.7: Administration structure for vehicle data residing on a link or in a buffer as AoS (array of structs)

`QueueAdmin` data structure. The struct itself will be situated from byte position $3 \times i$. So byte indexes $3 \times i$ and $3 \times i + 8$ will be accessed. This is not feasible in a coalesced way as described in Section 3.1.1. As the penalty for uncoalesced memory access is rather high, other data structures should be used.

3.2.6 Struct of arrays

One inexpensive way to achieve coalesced access to memory is to rearrange the data into a multitude of different streams of data that can be accessed by the same index. Instead of using an array filled with structs of the above layout, this struct is changed to hold pointers to arrays of a simple data type, i.e. an integer value in this case. This will –in terms of memory access– give rise to a memory layout that is better aligned with thread indices. Two adjacent kernels will have adjacent memory accesses, separated by 4-byte-wide integers, which enables coalesced memory accesses, the fastest way to access the global memory from a thread block. In terms of the implementation it is also a simple optimization step, as one only "shifts" the index to the right.

This has been done to all administrative memory layouts, and the allocation code has to be slightly adapted. The modified layout of the data structure is illustrated by Fig. 3.8. This Struct of Arrays (SOA) layout has also been suggested in a CUDA workshop [50].

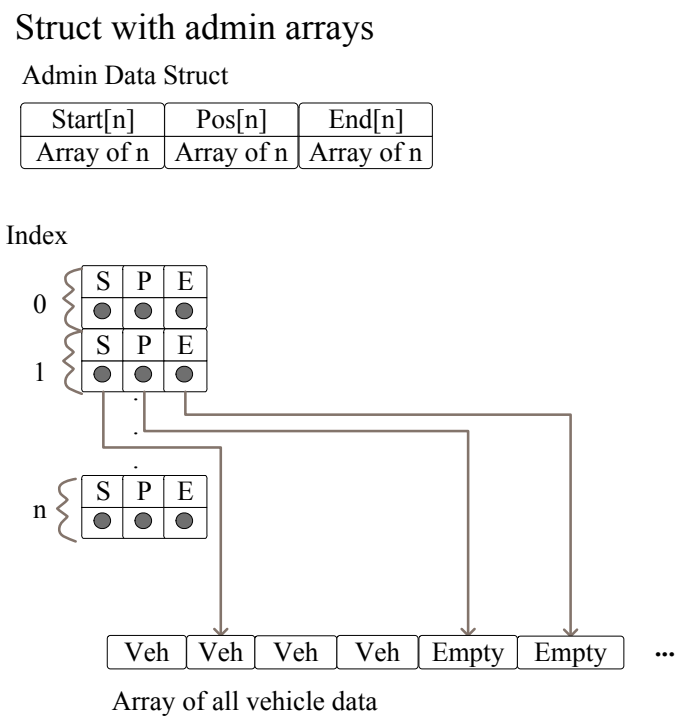


Figure 3.8: SoA (struct of arrays) layout for the administrative structure

```
QueueAdmin{
  int pos;
  int start;
  int end;
}
```

(a) AoS before

```
QueueAdminSoA{
  int pos[];
  int start[];
  int end[];
}
```

(b) SoA after

Figure 3.9: Translation of the struct from AoS to SoA.

The actual struct is to change from the form in Fig. 3.9(a) to that in Fig. 3.9(b). Likewise the implementation needs to be changed. A former expression

```
int size = array[index].pos - array[index].start;
```

will change to

```
int size = array.pos[index] - array.start[index];
```

This transition could be done in a nearly mechanical way. It was applied to the buffer and the link's and agent's administrative structs. The performance gains through this optimization will be discussed in Section 3.4.

3.2.7 Ring buffer

The second drawback of our initial implementation from Section 3.2.5 is the need to move all remaining vehicles, whenever vehicles leave the queue. This leads to performance penalties in a congested situation, when only a small number of vehicles is allowed to leave a link and all remaining vehicles must be moved forward each time step. It also causes additional uncoalesced memory accesses, which should be avoided.

Fortunately, there is a well known cure to this problem, namely the use of a *ring buffer* as known from file I/O implementations [45]. The *ring buffer* basically adds another pointer to the original struct, pointing to the start of the queue. So, whenever a vehicle is removed, the start pointer is decremented, and when a vehicle is inserted at the end of the queue, the end pointer is incremented. A possible ring buffer data structure is given by the following struct, as illustrated by Fig. 3.10.

```
QueueAdminRing{
  int start[];
  int len[];
  int ep[];
  int count[];
}
```

The ring buffer comes with some extra overhead in administrative data and effort. The ring buffer implementation above has one *start* pointer which is pointing into the vehicles block to indicate this link's first memory position as before. The other members of the struct are relative to the start position. The *len* member gives us the maximum size of this buffer. So *start* + *len* points behind the last element of this buffer. The *ep* (extraction point) member indicates where the first element of the actual queue resides. The *count* member is also relative to the extraction point member and indicates how many units there are in the queue at a given time. Vehicles are removed from a memory position calculated by

$$pos_{top} = start + ep$$

.

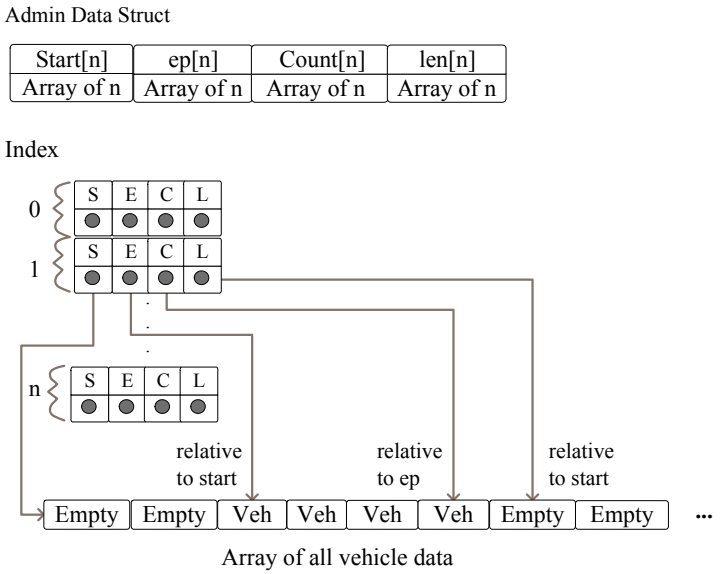


Figure 3.10: Administration structure implemented as a ring buffer

Up to *count* vehicles can be removed from there, calculating the next position as

$$next_{top} = start + (ep + i)\%len$$

where i runs from *0..count -1*. We can insert up to *len - count* vehicles at the insertion position calculated as

$$pos_{insert} = start + (ep + count)\%len$$

Insertion and removal of vehicles does not need to move any existing vehicles anymore, enhancing performance in situations where the AOS data structure was not performing well. The size of our administrative data structure is increased by one which could result in performance losses in the uncongested time steps of the simulation, as more reads are needed to evaluate the ring buffers state.

3.2.8 Vehicle data

Two versions of data handling for the vehicle data were implemented. For the AOS and SOA implementation the vehicle data is defined as

```
typedef struct vehdata {
  int nextdata;
  int id;
  int deptime;
} VehData;
```

whereas the more sophisticated implementation with the *ring buffer* structure uses a plain

```
typedef unsigned int VehIdType;
```

for storing the vehicle's ID only. The first data structure `VehData` caches additional data, namely the next departure time as well as the next plan's entry, for easier access in the kernels. Accessing these data locations would cost two dereferencing actions otherwise. Although this seems a good idea, the later kernel for the *ring buffer* based algorithm uses only the vehicle ID as data for the buffer and link-space and caches the departure time in the plan's administrative structures and just executes the double referencing for the plan's data. This makes vehicle movement less costly, as only one integer value has to be moved. As the benchmarks will show, this will not necessarily improve the performance of a run. The additional references to get the uncached data from the plans file can, in some situations, take more time than is saved by not having to move the bigger data structure. But the buffer and link space sizes shrink to a third of their original size, thereby saving memory and making it possible to run bigger scenarios. This was deemed more important than the performance loss. Exact figures will be shown in the benchmarks in Section 3.4.

3.3 The simulation loop

After having implemented the various data structures, in this section, the actual code to implement the queue model on the CUDA hardware will be discussed. Only one implementation of the code for the internal movement on links is presented as the `moveLink()` code did not present many options for diversification. Three different implementations for the movement of the vehicles across the nodes were implemented. All of these implementations will be discussed in the following sections.

The starting point of this thesis was not so much to present the best parallel code for implementing a transport simulation on the GPU, but to see if it is feasible to achieve relevant speedups with using existing –CPU based– paradigms and algorithms and transport them to GPU without much change. Therefore the most straightforward implementation was chosen, whenever possible.

3.3.1 The moveLink() kernel

The pseudo code for the movement within the links can be found in Section 2.3.1. To transfer this to the GPU, mainly the access to the data structures, described in the sections above, needs to be implemented. An implementation of the `moveLinks()` kernel for the AOS (array of structs) data structure can be found in listing 3.5.

In line 6 is an "early out" option which is chosen whenever no vehicle is on the link at that time. If there are vehicles on the link, the maximum possible number of vehicles on the link is calculated in line 8. The `dataEnd` member might be larger than the actual `linkEnd` would allow, as the buffer's `linkpos` member is incremented in `moveNodes()` regardless of the actual insertion of a vehicle into the link. This will be discussed in depth in Section 3.3.3. In lines 10 to 12 the allowed flow of vehicles for this time step is calculated. Then all vehicles will be iterated, until either a vehicle has a departure time later than the time of this step, or the buffer's flow capacity is exhausted or all vehicles were iterated. If one vehicle has an applicable departure time, the kernel will check whether the next planned item is to enter another link or to perform an activity. In the latter case, the timer for this particular activity will be set, and the vehicle is removed from the link's queue. The handling of vehicles in activities is obliged to the `insertWaiting()` kernel discussed later. If a vehicle is ready to enter another link and enough buffer space is left, lines 27ff. will move that vehicle into the link's buffer and decrease the `flow` related variables accordingly.

If either the first vehicle does not fit into the buffer, or the first vehicle's departure time is later than this time step, the iteration of the queued vehicles can be stopped. In lines 42 to 48, the remaining vehicles are moved forward, if necessary. Line 50 finally stores the updated `flow_accu` variable for use in the next time step.

Listing 3.5: Kernel for link movement (AoS)

```
1  __global__ void moveLink_k(int time, ...) {
2      int linkIndex = __mul24(blockIdx.x,blockDim.x) + threadIdx.
           x, p = 0, move = 0;;
3
4      int dataIndex = linkAdmin[linkIndex].start;
5      int dataEnd = linkAdmin[linkIndex].pos;
6      if (dataIndex == dataEnd) return;
7
8      dataEnd = min(linkAdmin[linkIndex].end, dataEnd);
9
10     float cap = flow_cap[linkIndex];
11     float accu = flow_accu[linkIndex];
12     if (accu < 1.0) accu += cap-(int)cap;
13
14     for (p = dataIndex; p < dataEnd; p++){
15     //get vehicle data and check departuretime
16     VehData* veh = &linkData[p];
17
18     if (veh->deptime <= time){
19       // this vehicle is ready to go to a buffer if there is
             space
20       // first check if next data is ACTIVITY, then we do not
             need buffer
```

```
21        if ((veh->nextdata & ACTDATA) == ACTDATA) {
22          int actend = veh->nextdata & 0xffffff;
23          //set act endtime in plans, just remove veh from link
                aka inc move;
24          plans[veh->id].nextactend = actend;
25          move++;
26        } else {
27          int bufferpos = bufferAdmin[linkIndex].pos;
28          if( (bufferpos < bufferAdmin[linkIndex].end)&& ((cap >=
                1.0) || (accu >= 1.0))) {
29            //yes, there is place in the buffer, move vehicle
                  there
30            moveVehicle(&bufferData[bufferpos], veh);
31            move++;
32            bufferpos++;
33            bufferAdmin[linkIndex].pos = bufferpos;
34            if (cap >= 1.0) cap -= 1.0;
35            else accu -= 1.0;
36          } else break;
37        }
38      } else break; // if nothing has been removed from link, or
          no place in buffer stop here
39    }
40
41    // from now on just step through remaining veh to copy them
        to front of link
42    if (move != 0) {
43      for(;p < dataEnd;p++) {
44        VehData* veh = &linkData[p];
45        moveVehicle(&linkData[p - move], veh);
46      }
47      linkAdmin[linkIndex].pos = dataEnd - move;
48    }
49
50    flow_accu[linkIndex] = accu;
51   }
```

3.3.2 The moveBuffer() Kernel

In a first implementation of moving particles on the GPU, the actual *node logic* was completely omitted. Looking for the simplest possible solution for moving particles around the network, these particles were transported from the buffer directly back into the link space of the appropriate link that is the next link on the route of the particle. Although this does not necessarily deliver a realistic traffic pattern, it is a very simple test to see what speedup can be achieved by the use of CUDA devices. The `moveNodes()` code of the queue model is replaced with a `moveBuffer()` method, which simply takes

all vehicles from a link's buffer and tries to put them into the next link's space. This functionality is shown in listing 3.6.

Listing 3.6: Kernel for buffer only movement (AoS)

```
__global__ void moveBuffer_k(int time, ...) {
  int linkIndex = __mul24(blockIdx.x,blockDim.x) + threadIdx.x,
      p, move = 0;

  int dataIndex = bufferAdmin[linkIndex].start;
  int dataEnd = bufferAdmin[linkIndex].pos;

  //clip too large pos pointers from moveLink() code
  dataEnd = min(bufferAdmin[linkIndex].end, dataEnd);

  for (p = dataIndex; p < dataEnd; p++){
    //get vehicle data and check departuretime
    VehData* veh = &bufferData[p];
    // go to a buffer if there is space...so, get next link's
       buffer
    int toLinkIndex = veh->nextdata & 0xffffff;
    int linkpos = atomicAdd(&linkAdmin[toLinkIndex].pos,1);

    if( linkpos < linkAdmin[toLinkIndex].end) {
      // calc nextdata and deptime here
      int nextDataIndex = plans[veh->id].idx;
      PersonData nextData = plansData[nextDataIndex];
      veh->nextdata = nextData;
      plans[veh->id].idx = nextDataIndex + 1;
      veh->deptime = time + linkAdmin[toLinkIndex].traveltime;
      moveVehicle(&linkData[linkpos], veh);
      move++;
    } else break; // if the "first" vehicle can not move stop
       the whole process
  }
  // from now on just step through remaining veh to copy them
     to front of link
  if (move != 0)
    for(;p < dataEnd;p++) {
      VehData* veh = &bufferData[p];
      moveVehicle(&bufferData[p - move], veh);
  }

  bufferAdmin[linkIndex].pos = dataEnd - move;
}
```

The algorithm iterates over the vehicles from the buffer's start position to the current insertion point. The destination link of the topmost vehicle is being looked up, and an atomic addition is executed on the destination link's position pointer. The contents of the pointer before the addition is returned

and stored in `linkpos`. As this operation is atomic, it is guaranteed that no other concurrently running thread can get the same link position counter. Still it has to be ensured that the gathered link position counter is a valid one. It must be smaller than the destination link's `endPosition`.

If this condition is met, it is safe to store the vehicle at the top of our buffer into this link's space. After that the iteration continues. If the topmost vehicle cannot be moved, the whole iteration stops. Again, at the end of our kernel remaining vehicles have to be moved to the front, and the latest end position of the buffer has to be saved for later use.

3.3.3 The moveNode() Kernel

A more realistic version of the movement at an intersection is given by the implementation of `moveNodes()`. This kernel can rely on knowledge about what incoming links an intersection has to service. For this implementation the incoming links are being served in a predetermined order. The incoming links are sorted from larger flow capacities to smaller ones. The algorithm iterates over all incoming link's buffers, satisfying the buffer's demand as far as possible and then switching to the next link, if this link's buffer is empty or the first vehicle cannot leave the link. Then the next link has a chance to distribute its vehicles. The code for that is quite simple. It is based on `moveBuffer_na()` method from the last section. The only difference is that this `moveNode_na()` method does not need to use an *atomic operation* for the destination link's position increment. The only links that can possibly access the destination link's space are now being serialized inside the `moveNodes()` method, as only the incoming links of this particular node are candidates for inserting vehicles into the destination link. The method is listed in listing 3.7.

Listing 3.7: Kernel for node movement (AoS)

```
1  __global__ void moveNodes_k(int time, ...) {
2   int nodeIndex = __mul24(blockIdx.x,blockDim.x)+threadIdx.x;
3
4   int dataStart = nodesData[nodeIndex].start;
5   int dataEnd = nodesData[nodeIndex].end;
6
7   for (int p = dataStart; p < dataEnd; p++) {
8     // serialized calc of movement for link's start to end
9     moveBuffer_na(p, time, ...);
10  }
11 }
```

3.3.4 The moveNode() Kernel with random selection

The above code privileges links with higher flow capacities. This can lead to overly decreasing traffic on the smaller links and is not a good implemen-

tation of traffic behavior. Therefore it is necessary to give the other links a chance to continue with their traffic event when the main road is constantly congested.

A common way, also implemented in the Java version of the queue simulation, is to choose the sequence of precession for the links in a random fashion. To maintain a higher probability for the "main streets" to be chosen by the random draw, the domain of the draw is divided into different sized parts, depending on the flow capacity of the respective link. This has been described in Chapter 2.3.2. Multiple random numbers have to be drawn for this approach. For the CUDA simulation a different approach was chosen. Again, the domain of the random draw was divided into sections of length proportional to the relative flow capacities of the incoming links of a node. Then only one random number is drawn and a link is chosen as in the Java implementation. Beginning from this link, all incoming links are then served in a ring-buffer like fashion. Therefore it is still maintained that the link with the highest flow capacity has the highest probability to be drawn, but other links will get a chance from time to time, depending on their flow capacity. To reduce the necessary computation and memory accesses in the kernel a precomputed value is stored with each node as follows: As the random number generator presented by [55] and used in this implementation generates positive 32-bit integer values, the integer number $inSumMult$ necessary to multiply the actual sum of incoming flows for one node with to yield the maximal positive 32-bit integer `(MAX_INT == 0x07fffffff)` is stored with every link. So for every Node n

$$inSumMult_n = \sum_{incominglinks_n} capacity_{flow}/0x7fffffff$$

is stored with the node.

In the kernel, one random number is used. The sum of the incoming flows multiplied with the node's $inSumMult_n$ is computed. This is guaranteed to sum up to `MAX_INT`. As soon as the sum is greater than or equal to the random number, the loop is stopped and the according link is chosen as the start link. All other links are served in a ring-buffer fashion starting from this link and ending with the link before this start link. So for a link $l1$ the probability to be chosen as the first link is exactly as it was in the Java implementation:

$$p1_{l1} = flowcap_{l1}/\sum_{links_{incoming}} flowcap_{incoming}$$

But the probability of a link $l2$ to be picked as a second link will differ from the java version's

$$p2_{l2} = (1 - p1_{l2}) \times flowcap_{l2}/((\sum_{links_{incoming}} flowcap_{incoming}) - flowcap_{l1})$$

and will be equal to the previous link's probability to be the first link. Nevertheless this behavior will assign every node a certain chance to deliver their vehicles first.

Both the Java version and the version presented here try modeling (unknown) logic of the street crossings. Neither of the implementations have a straight-forward real-life interpretation. Dividing the time steps into time windows according to the actual capacities of the links would have been a feasible approach to model the intersection logic without resorting to random numbers. The lowest common denominator for all link capacities must be found and all link intersections must get time slices according to their capacity times this number. This approach would model some sort of traffic signal installed at the respective node or street crossing. Replacing them with a more precise model dealing with the traffic signals installed would be advantageous. This has not been tried in this thesis, because the complexity of such an implementation is high and additional sources of data would have to be acquired for modeling the signals correctly.

Listing 3.8: Kernel for flow-capacity-dependent random pick of a starting link

```
1  __device__ int
2  chooseRandomStart(int dataStart, int dataEnd, int inSumMult,
       int rnd) {
3    unsigned int sum = 0;
4    int i;
5    for (i = dataStart; i < dataEnd; i++) {
6        sum += (int)(inSumMult * dsimdata.flow_cap[i]);
7      if (rnd <= sum) break;
8    }
9    return i - dataStart;
10 }
```

3.3.5 The alternative moveNode() and moveBuffer() kernels

As avoiding uncoalesced memory accesses proofed to be a good strategy for achieving speedup, another kernel was implemented trying to exploit this even further. The implementation called RING2 in the results of Sec. 3.4 differs from the RING implementation in that the `moveBuffer()` resp. `moveNode()` kernels only mark vehicles ready to move over to the link space. This is done by writing the vehicle's destination index into an additional field `vehDest`. A second kernel runs over the complete block of buffered vehicles moving only the vehicles with a destination set.

This additional kernel can access the buffer data in a coalesced way. This leads to performance gains in the RING2 setting with `moveBuffer()` but does not perform well in the case with `moveNode()`. While with `moveNode()` there is a speedup possible with doing only coalesced memory accesses, the

`moveNode()` code still executes the read access to the buffer in an uncoalesced manner, as it is aligned to the node data. The alternative implementations only make sense when one has to move the larger `VehData` struct. In case of the `VehIdType` used in the EVENT variants, the writing of one integer pointer per vehicle is as expensive as directly writing the vehicle's `Id` into the appropriate link space. Therefore the RING2 variant was discontinued in favor of the, also less space consuming, EVENT variant with the `VehIdType` data type.

3.3.6 The insertWaiting() kernel

One last kernel is needed. As seen in Section 3.3.1, whenever a vehicle has reached the destination link to perform an activity on, it is simply removed from the link's queue and a timer is set to the time the activity will stop.

The `insertWaiting()` kernel is responsible for checking whether a vehicle wants to leave an activity. It checks the timer for every plan on every tick. If the timer is set to -1 it ignores the plan, as this indicates that the vehicle is driving along a route. Otherwise it tests whether the activity's end time is reached. When that is given, the vehicle is put back into the link buffer of the activity' link, when there is space left in the buffer. Otherwise the kernel will repeat trying to do this in the next time step, until it succeeds. It is also possible that two activities take place in succession at the same link. In this case the timer is reset to the next activity's end time.

Again the kernel is shown in listing 3.9 for the AOS data structure.

Listing 3.9: Kernel for executing activities (AoS)

```
1  __global__ void insertWaiting_k(int time, ...) {
2   int planIndex = __mul24(blockIdx.x,blockDim.x) + threadIdx.x;
3
4   int deptime = plans[planIndex].nextactend;
5   if (deptime > time) return;
6
7   int nextDataIndex = plans[planIndex].idx;
8   PersonData nextData = plansData[nextDataIndex];
9
10  if ((nextData & LINKDATA) == LINKDATA) {
11    int linkIndex = nextData & 0xffffff; // link of activity;
          insert in buffer
12    int bufferpos = atomicAdd(&bufferAdmin[linkIndex].linkpos
          ,1);
13
14    if( bufferpos < bufferAdmin[linkIndex].linkend) {
15      //this will now drive away, put it in buffer, if there is
            room
16      plans[planIndex].nextactend = 0xffffff; //no activity
            being performed right no
17
```

```
18        //yes, ther is place in the buffer, fill vehicle there
19        VehData* veh = &bufferData[bufferpos];
20        veh->id = planIndex;
21        nextData = plansData[nextDataIndex+1]; //this is the NEXT
              link
22        veh->nextdata = nextData;
23        plans[planIndex].idx = nextDataIndex + 1;
24      }
25    }else { //two ACTIVITIES on the SAME link
26        // increase counter to next activity
27        plans[planIndex].idx = nextDataIndex + 1;
28        // set next departuretime
29        plans[planIndex].nextactend = plansData[nextDataIndex+1]
              & 0xffffff;
30    }
31  }
```

3.4 Benchmark and performance results

Not all combinations of the above algorithms and data structures have been tested. The AOS and SOA data structures have been implemented with the `VehData` structure only, the plain *ring buffer* implementation uses the `VehData` struct as well, but the versions used for writing events in the next chapter, which are also based on *ring buffers*, use the simpler `VehIdType` as defined in Section 3.2.8. The simpler data type comes with the advantage of not having to move three integer values every time a vehicle is moved across a link or buffer. But on the other hand this version has to get the next data chunk needed to determine the next link by using two indirections, whereas the older version only needs one. As the benchmark results will show, these differences about balance each other. Still the use of `VehIdType` cuts the memory consumption of link space and buffer to a third, so this version is preferred in later implementations. There are four different implementations of data structures used named SOA (struct of arrays), AOS (array of structs), RING (ring buffer) and one called EVENT (ring buffer, with `VehIdType`). Furthermore three different ways to move vehicles from the buffer to the next link's space were implemented.

The plain `moveBuffer()` method does not implement any regulation for the competition of the incoming links for space on the outgoing links. It is the simplest implementation. The `moveNode()` implementation does serve the incoming links in the order of their respective flow capacities, which is unfair to the smaller links. And a third implementation `moveNodeRnd()` chooses a random incoming link as a starting link at every time step instead of using the predetermined order. Not all methods have been implemented for all data structures. Table 3.3 gives an overview. The combinations

	data struct	`moveBuffer()`	`moveNode()`	`moveNodeRnd()`
AoS, sec.3.2.5	VehData	AOS	AOSNODES	
SoA, sec.3.2.6	VehData	SOA	SOANODES	
Ring, sec.3.2.7	VehData	RING	RINGNODES	RINGRND
Ring2, sec.3.3.5	VehData	RING2	RINGNODES2	
Event, sec.3.2.8	VehIdType	EVENT	EVENTNODES	EVENTRND

Table 3.3: Data structures and execution paths implemented for benchmarking. *(The last row is named "Event" as this configuration is used for implementing the event handling in Sec. 4.1)*

with blank fields have not been implemented. The names in the other cells correspond to the names used in the benchmark tables.

3.4.1 Hardware

Two different hardware systems have been used for evaluation of the benchmarks. The technical data of the two systems can be found in Table 3.4. The LOW system uses an inexpensive, passively cooled version of the GeForce 8600 series. With its severely reduced memory bandwidth and small number of cores, it marks the lower end of possible GPU acceleration. The HIGH system uses the latest installment of NVIDIA GPUs. This GPU has 240 cores and, albeit being at nearly the same clock rate, a much higher bandwidth. This is due to the fact that each core has its own link to the main memory. The bandwidth per link is with 238 MB/s (LOW) and 500 MB/s on the HIGH system more reasonable to compare. Nevertheless the bandwidth of the lower end system is inferior to that of the high end system, which should result in a speedup decrease that is higher than the actual ratio of cores.

System		GPU	CPU
LOW	Name	GeForce 8600 GT	Intel Pentium
	Num. cores	32	2
	Clock rate	1.18 GHz	2.0 GHz
	Memory size	1 GB	3GB
	Mem. bandwidth	7638 MB/s	
HIGH	Name	GeForce GTX280	Intel Pentium
	Num. cores	240	2
	Clock rate	1.3 GHz	2.2 GHz
	Memory size	1 GB	3 GB
	Mem bandwidth	120 GB/s	

Table 3.4: Technical data of CPUs and GPUs used

3.4.2 Data samples and network

Each of the configurations in Table 3.3 has been run on both systems with a variety of data sets. To construct feasible results a data set from a previous work of Chen et al. [30] has been taken. It represents the overall work traffic in the Zurich area on a normal business day. This sample has about 1.8 million agents. To construct a base set for the runs in this thesis, 1 million agents have been taken from the Zurich area sample. Starting from this base set of 1 millon agents further samples have been taken. Each sample of 100%, 50%, 25%, 10% and 1% was run on the Zurich area network consisting of approximately 37.000 links and 25.000 nodes. The actual Java

Code		LOW Java base Rev. 3562	LOW Rev. 7241	LOW Rev. 7241	Opteron Rev. 7241
event handling		w/o	w/o	with	with
Sample	# Agents				
100%	1 mio.	108	–	–	96
50%	500.000	181	142	–	170
25%	250.000	308	323	254	267
10%	100.000	462	533	463	471
1%	10.000	1123	1004	938	1169

Table 3.5: Real time ratios for the CPU-based Java simulation and sample sizes used in this thesis (all ratio values were taken at midnight, 24:00)

version of the MATSim queue simulation was run on the same set of data to get an idea of the speedup achievable by using the GPU instead of a highly optimized implementation on the CPU. These runs were executed on the system labeled "LOW" in Table 3.4 with an earlier version of the Java simulation (Revision 3562) and all event handling disabled. The running times of these runs can be found in the column *Java base*. All speedups mentioned in the rest of this chapter are relative to these running times.

To get an impression of the computing power of this *home PC* system, additional runs were executed on an *Opteron-workstation*. These runs were executed with a more recent version of the Java simulation (Revision 7241) with improved speed. The running times can be found in Table 3.5 and include the event generation and handling. To compare these values to the values used for speedup calculation additional runs were made. The *LOW* system was run on certain samples with event handling installed, thus with exactly the same configuration as the *Opteron* runs. Then a further run was started, this time without event generation and handling, as the original *Java base* configuration. The real time ratios achieved for the different configurations together with the actual agent count of the samples produced

can be found in Table 3.5. The running times of both CPU-based systems are almost equal for the configuration with event handling. The additional runs without event handling show an improvement of about 15%. One can assume that the *Opteron* runs would also improve to that magnitude with event handling disabled. The running times of the original runs with the *Java base* are thus expected to have about 5% to 15% less performance.

Finally, the samples were also run with a CPU based version of our ring buffer algorithm, for getting an impression of how the algorithm performs on a CPU. This will help to estimate whether a possible performance gain is to be attributed to the multi-core hardware used, or solely to the necessary algorithmic changes made to implement the queue model on CUDA.

All real time ratio and speedup values were calculated from the simulation time necessary to run the simulation up to midnight (24:00). After midnight only a very small group of agents was still active. But these few agents drove for another 10 hours. So the sample until midnight deemed a better performance indicator than the overall runtime of the simulation including the long periods with nearly no traffic.

3.4.3 Results for the GeForce 8600 GT

The GeForce 8600 GT used in running the benchmarks is a passively cooled version of the GeForce 8600 series of GPUs. This version is at the lower end of the GPUs capabilities. It has 32 cores running at 1.18 Ghz. The memory bandwidth is also at the lower end. Still it has 1GB of RAM so even the bigger samples up to the 100% sample can be run. The benchmark results for this card will be discussed in the following sections.

Results for AOS and SOA structs

The results for the simpler implementations shown in Figure 3.11 indicate that the AOS and AOSNODES implementation do perform a bit better than the actual Java version which we are using in the MATSim framework. The results for the SOA and SOANODES data structures look more promising. This has to be attributed to the data access pattern, which is not coalesced in the AOS case. Albeit the speedup is still rather low, it performs better or equal to the Java implementation. Proportionally seen it is apparent that the GPU implementation of the queue model scales in nearly the same way as the Java version, so that the ratio stays about the same for all sample sizes. Only the SOA offers remarkably more performance on the 10% sample than the Java version.

Results for the RING implementations

When taking into account the ring implementations RING, RINGNODES, RING2 and RINGNODES2, another relevant speedup occurs as shown in

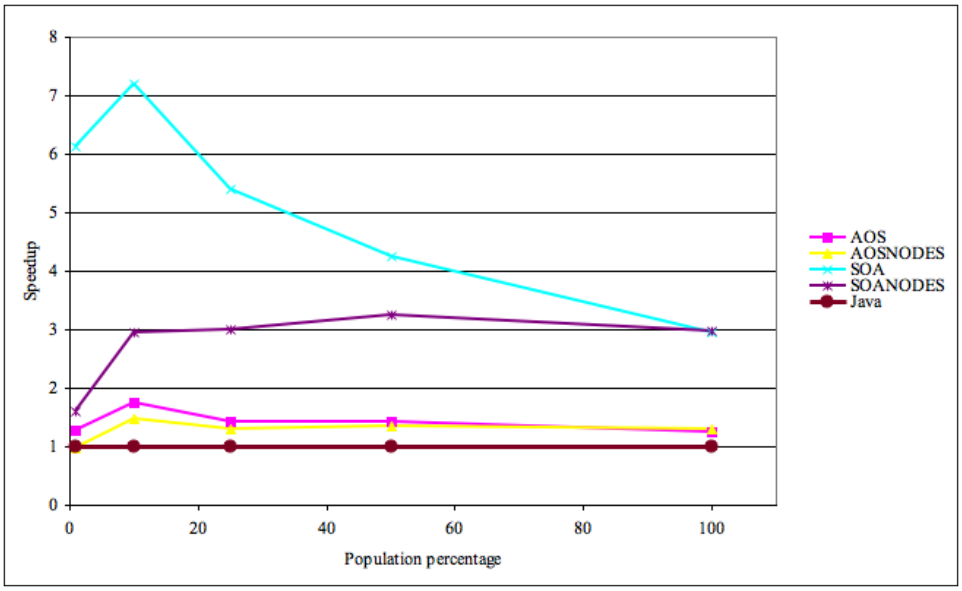


Figure 3.11: Speedup of the GeForce 8600 GT relative to the Java version

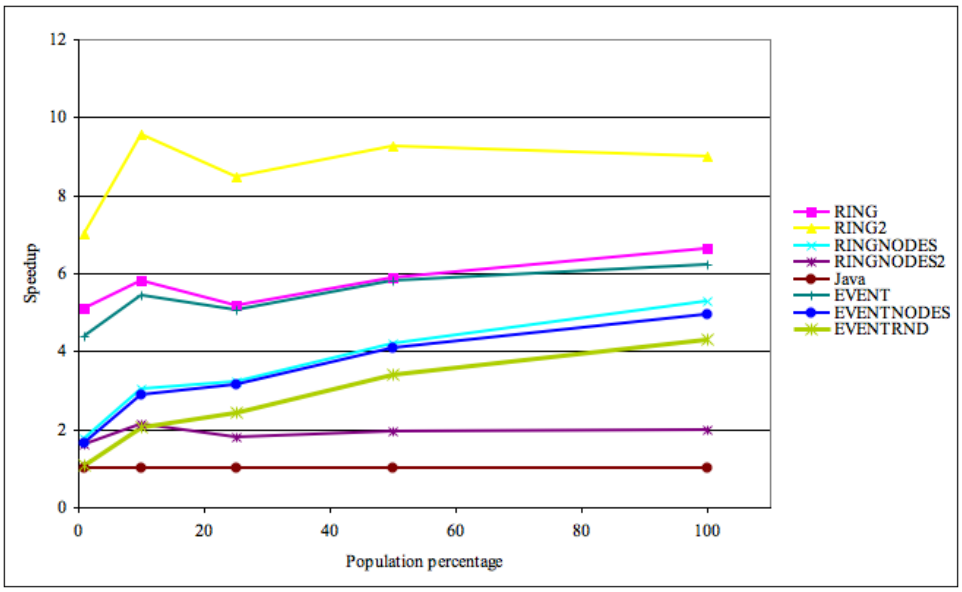


Figure 3.12: Speedup of the GeForce 8600 GT relative to the Java version (continued)

Fig. 3.12. The RING versions outperform the simpler implementations. With the bigger samples, this difference gets even more obvious. This is due to the higher number of vehicles moving through the net at one time step. This results in more performance being spent on moving the actual vehicles in the simpler implementation, which is not necessary for the ring buffer based implementations. The speedup achieved with the ring buffer based implementations goes up to nine times faster than the Java implementation. The version of the code which additionally generates events is slower than the version without. Still this performance loss is rather small. Even the generation of random numbers does not impact the performance prohibitively. As EVENTRND is the implementation closest to the Java version, the speedup of this version might be called the most relevant speedup. A speedup of four against the Java version is possible with the EVENTRND version.

Although the speedups achieved with our GPU implementation are not very big, still every implementation is faster than the Java version. In the next section we will run the code on the high end system and will compare the speedups to the ones here.

3.4.4 Results for the GeForce GTX280

The GTX280 is the latest installment of the G80 series of NVIDIAs GPUs. It runs at 1.3 Ghz, and has a remarkable memory bandwidth of 120GB per second and 240 cores. This is 7.5 times the core count of the lower end GPU. When our implementation scales well with the number of cores we should be able to achieve a speedup of 7 against the other runs. All test were run on this card with exactly the same code that was run on the low end card. Not even a re-compile was necessary to start the application.

Results for AOS and SOA structs

As we can see from the diagram 3.13, the speedup goes up to 37 for the SOA variant. Apparently the speedup with respect to the sample size is not linearly to the Java version anymore. The implementation is more sensitive to the sample size on the GTX280. This tendency is also visible in the direct speedup comparison of GeForce 8600 GT and GTX280 in diagram 3.16. Again one can see that the speedup decreases with the increase of the sample size. Nevertheless this speedup is remarkable. The GTX280 performs up to 16 times faster than the older card. As the number of cores only suggests a speedup of about seven. This must be attributed to the bandwidth of the memory access which is doubled in the newer card.

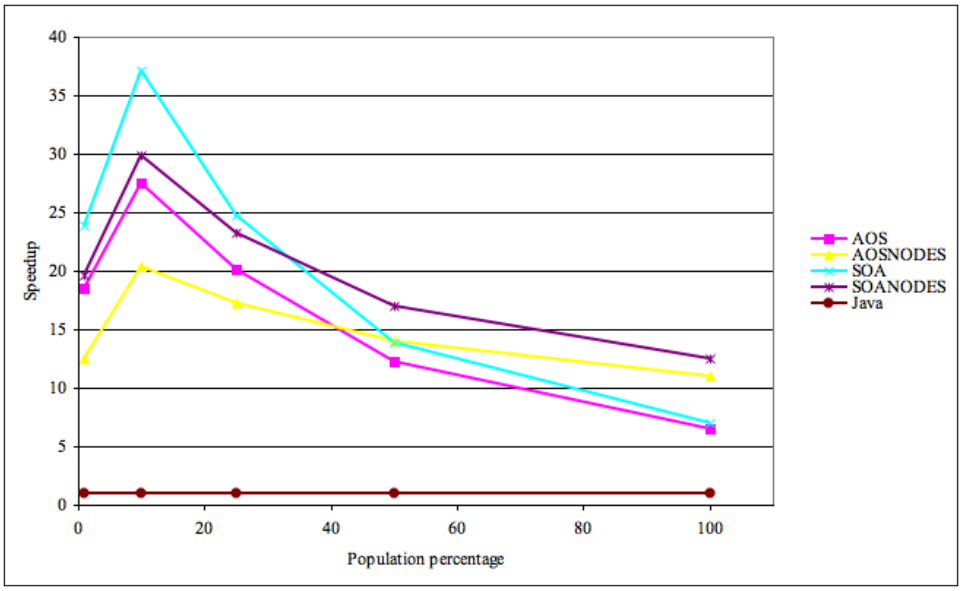


Figure 3.13: Speedup of the GTX280 relative to the Java version

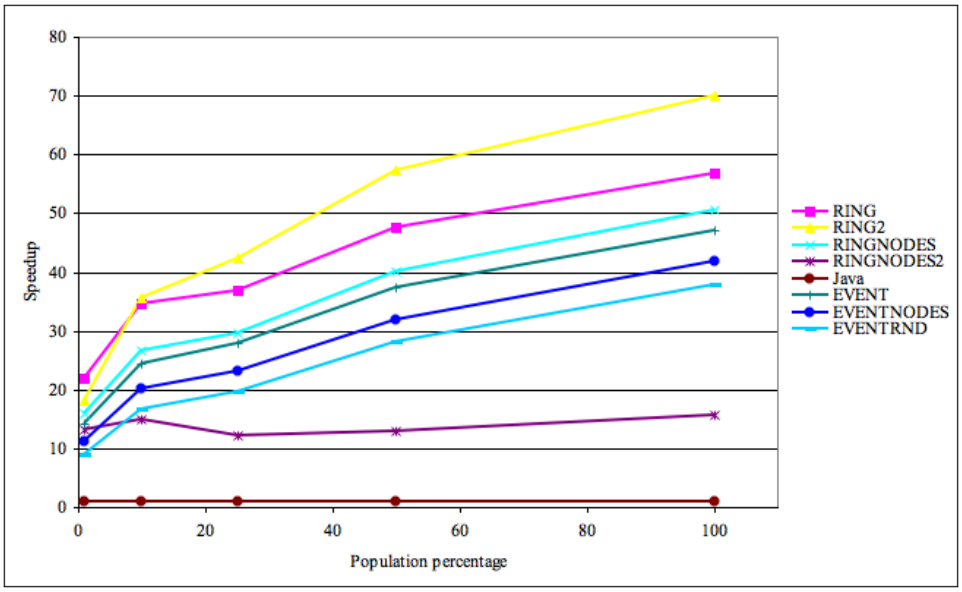


Figure 3.14: Speedup of the GTX280 relative to the Java version (continued)

Results for the RING implementations

For the more sophisticated implementations, which do not depend on memory access so much with the bigger samples, the picture of the last section changes again. As we can see from diagram 3.14 the speedup increases with the bigger samples. RING2 reaches a peak performance of nearly 70 times faster than the Java implementation. Compared to the GeForce 8600GT the speedup is somewhere between 7 and 8, which gives us a linear speedup with respect to the core count. This is a very good result. It seems that the RING implementations do scale very well with the count of cores available, whereas the slower implementations are rather memory bound, therefore improving in speedup against the GeForce 8600GT because of the higher memory bandwidth. The very low speedups with the 1% sample indicate that the GTX was not capable to cover the stalls from memory access with the small data sets for the high number of cores.

3.4.5 Conclusion

As the results show rather prominently, the GPU is capable to outperform any of the given CPU based versions of the physical layer's simulation. Even the most elaborate implementation EVENTRND would perform with a speedup of about 38 compared to the Java version. Diagram 3.15 shows the actual real time ratio that was achieved with the GPU version. A 50% sample is therefore executed about 13.000 times in one second of realtime, simulating 3.5 hours of these 500.000 agents in one second. Another example is that one can run a simulation with 1 million agents of the Zurich network from 6:00 in the morning to midnight with the RNDEVENT implementation in about 15 seconds.

Also, the speedup of the GTX 280 against the GeForce 8600 GT is interesting. Many times the speedup is higher than the actual core ratio would allow. This has to be attributed to the faster memory bandwidth and a newer implementation of the core logic. Results for the relative speedups can be found in diagram 3.16.

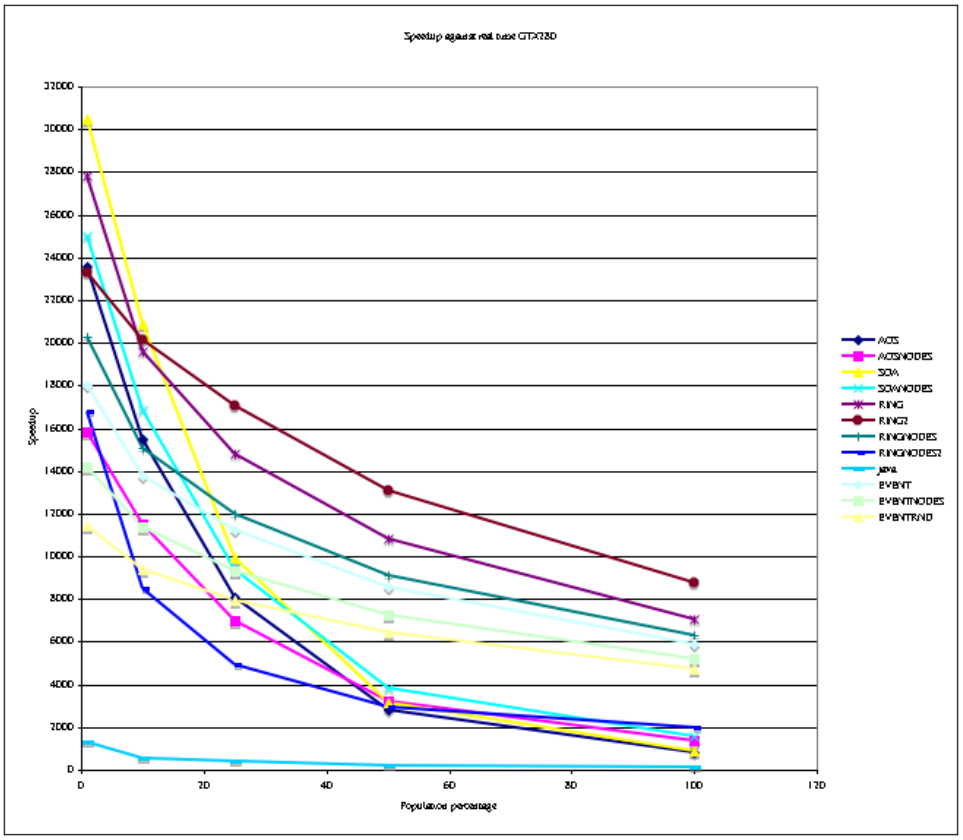


Figure 3.15: Real time ratio of the GTX280 (simulated secs / wall clock secs)

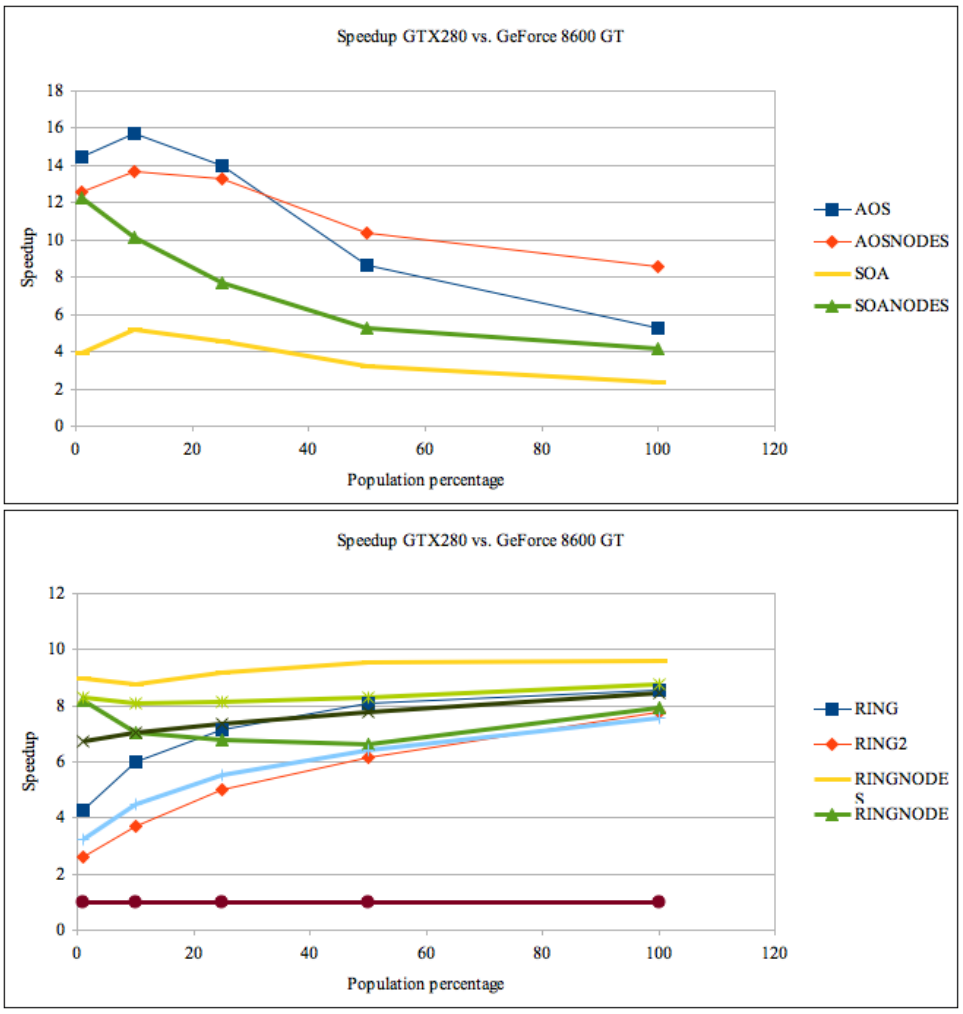


Figure 3.16: Speedup of the GTX280 relative to the Geforce 8600 GT (continued)

Chapter 4

Integrating the CUDA Simulation into the MATSim framework

Apart from being a useful tool for benchmarking the computational potential of CUDA devices, the actual benefit from our new implementation of a mobility simulation is rather limited. To make use of it, it is necessary to integrate the physical simulation into the MATSim framework. As seen in Figure 2.2 the connection of the simulation to the strategy layer of MATSim is twofold. On the one hand the simulation needs to be capable of reading MATSim plan files as an input format. This was already achieved with the implementation described in Chapter 3. On the other hand, the simulation needs to output descriptions for all relevant events inside the "virtual reality" simulated. These observations are the basis for all reasoning inside the strategical layer of MATSim in almost the same manner as the given plans are the basis for all activity that takes place inside the simulation. In the C++ version of the simulation from Cetin [24] and the *DEQSim* from Charypar [27] the plans as well as the events are written to a file by the generating layer and then read by the recipient layer.

In this chapter the event representation in MATSim will be described as well as the measures taken to implement the generation of events in the CUDA simulation. Furthermore the JNI (java native interface) will be described. It will be used to omit the need to write plans and events to a file as an intermediate step. With JNI all necessary data can be exchanged directly.

4.1 Event generation and transportation

The information flow from the physical layer to the strategy layer in the MATSim framework is implemented as a messaging mechanism. The physical layer can create messages for all relevant events that occur in executing the physical layer. These notifications are called *Events* in the MATSim

ActEnd	**ActStart**
AgentMoney	AgentReplan
LinkEnter	LinkLeave
AgentArrival	**AgentDeparture**
AgentStuck	AgentWait2Link
PersonEntersVehicle	PersonLeavesVehicle
ArrivalAtFacility	DepartureAtFacility

Table 4.1: List of events used in the MATSim framework

framework. There is a set of possible events that the MATSim framework knows, but this is not a fixed set of possible events. Users might extend the events for their own need. Not all event types available in MATSim are needed for implementing the scoring of the executed plans. The CUDA simulation will only implement and generate a subset of the possible MATSim events. This subset includes all events necessary for the scoring function. Table 4.1 lists all types of events that MATSim has implemented at this time. In Table 4.1 the events relevant for the scoring function are displayed in bold type.

Data structure for events

The scoring function depends on five different events to determine the actual score of a plan.

- *ActStart event*: The *ActStart* event is issued whenever an agent is ready to start with a certain activity.
- *ActEnd* event: At the end time of an activity the *ActEnd* event is created.
- *AgentDeparture* event: This event indicates that an agent has started traveling from one activity to the next activity.
- *AgentArrival* event: Whenever an agent stops traveling and is ready to execute the next activity, this event is generated.
- *AgentStuck* event: The agent has not been able to move on for a certain defined period of time.

ActStart and *ActEnd* events will only appear in pairs in a successfully executed simulation run. Excepted from this rule are the first and the last activity, which must be of type "home". These activities will only issue *one* event. The first activity only issues an *ActEnd* event, whereas the last activity only generates an *ActStart* event.

Listing 4.1: Struct for event data

```
typedef struct eventstruct {
    int* eventid;
    int* agentid;
    int* linkid;
} Event;
```

The *AgentDeparture* event will either have a matching *AgentArrival* event, or –in case of an unforeseen termination of the plan's execution– an *AgentStuck* event.

Under certain conditions, the agent has to be removed from the simulation. In this case no *AgentArrival* event is issued and the agent is removed from the simulation. The scoring function will get informed of this event, as it has to get an opportunity to penalize this agent's plan. If the agent is removed from the simulation and his plan execution is canceled the *AgentStuck* Event is sent.

When an event is generated, it is accompanied by additional info about the event. The following data is mandatory for all implemented event types:

- event type: The type of the event to be reported
- time: The actual simulation world's time, when the event occurred.
- agentID: The agent which experiences the action described in the event type
- linkID: The id of the link this event occurred on

Additionally some events need further information:

- Agent Departure/Arrival Events: the index of the executed leg for this travel.
- Agent Start/End Activity: the index of the executed activity

In listing 4.1 the data structure for storing an event in the CUDA simulation is illustrated. Again, the data structure for storing the events is build in a SoA manner, to enable coalesced access. For each relevant date we reserve an full 32 bit integer. This is apparently oversized, as in 3.2.2 we already learned that *linkIDs* are constrained to 24-bit size, therefore it should be easy to put `eventtype` and `linkId` into one full 32-bit integer. This optimization has not been implemented.

Missing from this struct is the time information as well as the mutual indices into the activities and legs, respectively. The events are collected and transported to the MATSim framework at every time step of the simulation. Therefore the time of an event is implicitly given by the simulation

AgentStuck Event	1 Event	in `insert_waiting_rev`
Agent Arrival Event	1 Event	in `insert_waiting_rev`
AgentDeparture/WaitforLink Event	2 Events	in `insert_waiting_rev`
LeaveLink/EnterLink Event	2 Events	in `moveNodes_rev`

Table 4.2: Execution paths and maximum event count of these

time when the event is generated. The index of either activity or leg is reconstructed in the actual Java wrapper of the CUDA simulation by keeping track of the count of Agent Departure Events for each agent. This is discussed in more detail in Chapter 4.4.

Agents with all activities on one link

As has been said in the last chapter, when an agent spends the whole day at home, the physical simulation will not have a *linkID* to send for the Activity Start/End Events. We will send a zero to the Java framework instead of an agent's *linkID* for indicating this situation. The Java framework will have to find the actual location from the original plan of the agent. For the other "non-car"-mode transportation we are capable of creating meaningful events as the necessary data is provided by the –otherwise unnecessary– ULEG link ID data, as discussed in Section 3.2.2.

Memory demand for Events

As said above, the events of the CUDA simulation get collected at every time step. The events are stored in an array. An additional integer counter keeps track of how many events occurred in a time step. As dynamic memory allocation is not an option with CUDA, an upper limit of the count of events possible in one time step has to be found.

This upper limit is given by the following consideration: We cannot know how many agents will issue an event at any given time step. So we have to assume that all agents want to issue events. How many events one agent will be capable of generating in one time step is the next question to be answered, as the product of both will give an upper limit for event space constraints. To estimate this upper limit we take a closer look at the actual possible event schedules. Given the execution as described in 3.3 all events are generated in the `moveNodes_rev` and the `insert_waiting_rev` kernel. Therefore we need to inspect the execution paths of these kernels to identify possible sequences of events. The sequences of events are summed up in Table 4.2

This table shows us that there is a maximum of two events issued per kernel by one agent in each time step. We will have to allocate four times the number of agents of event structures to hold the maximum possible number

of events.

4.1.1 Mapping of internal link's and agent's representation

As has been described before, the representation of an agent's ID and a link's ID has been mapped to integer values for all uses in the CUDA simulation. The agent's ID is equivalent to its storage location in the agent administration data structure. The same holds for the *linkID*. To issue meaningful events to the Java MATSim framework this internal representation has to be resolved back to the original ID before generating an event. When reading the plans and network data from the XML-file, a map for either ID's has been constructed for backward mapping these IDs. This map will be used to reconstruct the original representation. The mapping is implemented as a simple array of integer values. The index into this array is given by the internal *linkID* or *agentID*. This is consistent, as the internal representation of all links/agents starts at zero and is sequential up to the actual number of links/agents.

It is hereby assumed that the external ID can be represented as an integer value as well. This was valid for the plans/networks used for evaluation of the implementation, so there was no need for a more general implementation.

Nevertheless this array could easily be extended to use strings instead of integers to hold the external representations. So, before the external method for issuing events is called, the IDs get converted back to their original value, making the internal representation fully transparent to the external application.

4.1.2 Event generation

Only the `moveNodes_rev` kernel, running over all nodes, and the activity related `insertWaiting_rev` kernel, running over all plans, issue events. The `moveLinks_rev` kernel will set a flag in case an event needs to be sent; the actual sending of the event will then take place in the `insertWaiting_rev` kernel. The `moveNodes_rev` kernel issues a maximum of two events per vehicle and time step, when a vehicle moves from one link's buffer into another link's space. The other kernel has no loops; therefore it is a simple task to extract the maximum number of possible events per kernel call as we did in the previous section. It also issues a maximum of two events per vehicle. Both kernels might issue their maximum number of events for one vehicle at the same time step, so the overall maximum count of events per vehicle and per time step is four.

The events are written to the events buffer using an atomic operation for synchronizing access to the buffer-counter. Another way to implement the events would have been to use fixed addresses for the events to be written, based on the *threadId*, storing the events in `event[threadID]` and

Implementation	real time ratio	Percentage of base case
No events (base case)	1517	100
Empty call	1517	100
ThreadSynchronize()	1484	98
Generating events	1286	85
Generating events & ThreadSynchronize()	1279	84
Saving to HOST & ThreadSynchronize()	1226	81
Saving to DISK & ThreadSynchronize()	1213	80
called from Java & transfer to Java	1028	67

Table 4.3: Changes in real time ratio at midnight (24:00) with event writing (25% sample run on a GeForce 8600GT, EVENTNODES)

`event[threadId*i]` for example. This has not been implemented though.

Three ways to output the events were implemented. The created events could either be written to disk or sent to the Java application or just be thrown away. The last implementation was for benchmarking only.

4.2 Performance results with Events

Another important question to answer is to what extent the creation and computation of the events will decrease the performance of the CUDA simulation. In Table 4.3 the different stages of event generation are benchmarked for one set of sample data (ivtch data, 25% sample) to give an impression of the penalty for the different output paths and for generating events in the CUDA kernels alone. In the table the actual real time ratio is listed along with a percentage relative to the version without event writing.

The "empty call" means that the `"tossEvent()"` method was called in the kernel, but had all lines with memory interaction commented out. This will give an idea of how expensive a sub-routine call in a CUDA-kernel is. Apparently the empty subroutine was removed by the optimizing compiler; no performance degradation is measurable. The `ThreadSynchronize()` call is more costly. This will be discussed in more depth in the Section 4.3 about the CPU loop. The actual generation of events degrades the performance of the simulation by about 15%. Writing the events to disk amounts to an overall decrease of 20%. There is only a very small 1% gap between the time it takes to actually get the events from the device memory to host memory and the additional writing of the data to disk. This is surprising at first glance, but it is probably due to the sophisticated scheduling described in Section 4.3. In summary it can be said that the generation of events does not come for free, but nevertheless with a reasonable degree of performance degradation. In any case 80% of the overall performance can be maintained

when writing the events to disk.

4.3 The CPU loop

The CPU loop is responsible for calling the kernel methods. As described before, more than one kernel was utilized to implement the simulation loop. Furthermore, events are created by the kernel calls. These events need to be transported either to a file or to the Java framework. This is the responsibility of the CPU loop. The CPU loop can go over several time steps before returning control to the framework. This is for convenience but also to reduce the calling overhead caused by switching from Java to C. A pseudo-code version of the event loop is found in listing 4.2.

As a first step the actual breakdown on grid size and block size for the kernel calls is calculated. `THREADMIN` is a constant describing how many threads we want to run in a block of threads. If the actual number of threads needed is higher – which should be the case in every relevant simulation run – then `THREADMIN` is taken as the number of threads per block and the number of blocks is calculated as the upper integer limit of the expression `ThreadCount / THREADMIN`.

All data that is used in the kernel is aligned to `THREADMIN`. That is, if `Plans_Count % THREADMIN != 0`, the plans will be filled with dummy plans consisting only of an `ACTEND` activity. Likewise links and nodes will be inserted with no connection or incoming links respectively. After calculating these values the actual kernels will be called. In case of a simple benchmark-run no event data will be written; otherwise the events collected in this time step will be written to disk or sent into the Java-framework. The event writing will be discussed in more detail in the next section.

4.3.1 Asynchronous event handling

As seen in listing 4.2 the CPU does not have much to do while the GPU executes the kernels for simulation. On the other hand, the events need to be sent to their final destination, which is a duty of the CPU. To accomplish that task in an efficient way the CPU should do that while waiting for the GPU to finish the kernel execution. Apparently we cannot work on the events of the actual simulation step, as they reside in the GPU's memory. But the events saved from the last time step are safe to be treated by the CPU. To even run the downloading of the actual events and the writing of the older events by the CPU a double-buffering technique is installed. Listing 4.3 shows the same code as above, but this time including the event handling of the CPU.

The function call to `cudaThreadSynchronize()` synchronizes kernel and CPU execution. After this point we can be sure that all kernels above this

Listing 4.2: The main CPU loop executing the kernels

```
void CMicroSimRingEvent::simStep(int stepcount, int stepsize) {

  int lthread = min(THREADMIN,simdata.linkscount);
  int pthread = min(THREADMIN,simdata.planssize);
  int nthread = min(THREADMIN,simdata.nodecount);
  int lblock = ceil((double)simdata.linkscount / lthread);
  int pblock = ceil((double)simdata.planssize / pthread);
  int nblock = ceil((double)simdata.nodecount / nthread);

  for(int k=0; k < stepcount; k++) {
    // choose the other Event buffer for writing into from
       kernels
    ...

    insertWaiting_rev<<<pblock, pthread>>>(simtime,  *pEvent,
       deventpos, dagentscount);

    moveLink_rev<<<lblock, lthread>>>(simtime, dflow_accu, *
       pEvent, deventpos);

    moveNodes_rev<<<nblock,nthread>>>(simtime, *pEvent,
       deventpos);

     if(psimdata->writeEvents) {
      // Do the events writing here... see next section
      ...
     }
    simtime += stepsize;
  }
  // write last round of events
  ...
}
```

Listing 4.3: The CPU loop's event handling section

```
void CMicroSimRingEvent::simStep(int stepcount, int stepsize) {

  //size calculation
  ...
  for(int k=0; k < stepcount; k++) {
    // choose the other Event buffer for writing into from
       kernels
    Event* lastEvent = pEvent;
    int lastpos = eventpos;
    // double-buffer
    pEvent = (pEvent == &devent1) ? &devent2 : &devent1;

    // kernel execution
    ...

     if(psimdata->writeEvents) {
      // Parallel to kernel execution: Write events from last
         time
      writeEvents(simtime-1, lastEvent == &devent1 ? event1 :
         event2, lastpos);
      cudaThreadSynchronize();

      // get accumulated event count from this kernel execution
         , is used in next round
      cudaMemcpy(&eventpos, deventpos, sizeof(int),
         cudaMemcpyDeviceToHost);

      //Asynchronous to host, copy mem of this kernel exe to
         eventbuffer
      saveEvents(*pEvent, pEvent == &devent1 ? event1 : event2)
         ;

      int help = 0; // reset eventpos on device
      cudaMemcpy(deventpos, &help, sizeof(int),
         cudaMemcpyHostToDevice);
     }
    simtime += stepsize;
  }
  // write last round of events
  cudaThreadSynchronize();
  writeEvents(simtime-1, pEvent == &devent1 ? event1 : event2,
     eventpos);
}
```

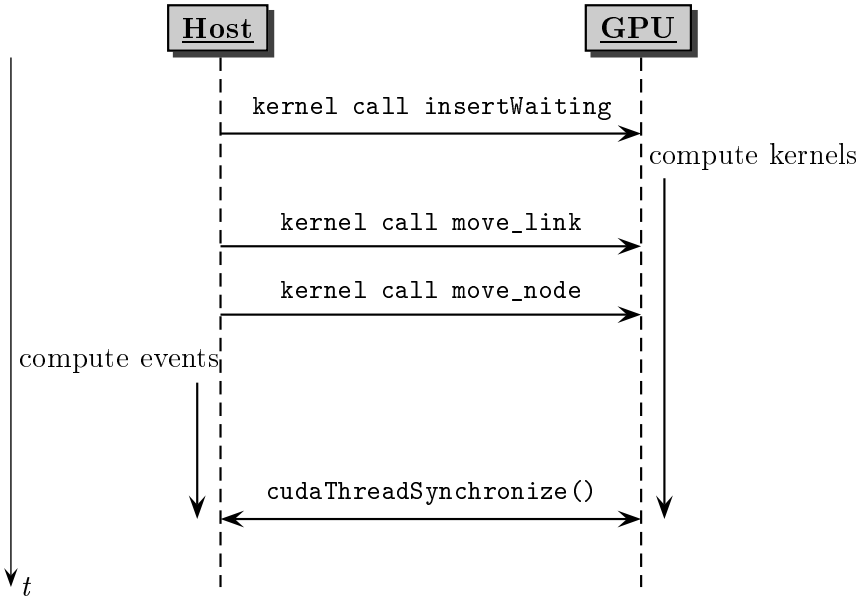


Figure 4.1: Parallel execution of CUDA simulation and CPU event handling

point have finished their path of execution. So all events of this iteration are guaranteed to have been sent into the GPU buffer by then. This buffer can then safely be copied to the host RAM. An asynchronous memory copy instruction is issued to the CUDA framework. Thereafter, the variable `eventcount` on the GPU is set to zero again. If there is another time step to run, the event-buffer will be exchanged with a second one (double-buffering technique) and execution of the next kernel is started.

As the CPU does not stop after starting the execution of the kernel, the `saveEvents()` function is called in parallel to kernel execution. This method handles the events from the previous frame, which have been completely copied to the CPU already as guaranteed by the last `cudaThreadSynchronize()` call. Finally the CPU and GPU execution paths will meet again at this barrier given by the `cudaThreadSynchronize()`. After the last iteration of this `simStep()` an additional `cudaThreadSynchronize()` and `saveEvents()` call is needed, to save the last iterations events. When the event handling of the CPU takes longer than the kernel execution, the whole simulation time is virtually annihilated by the event processing. This schedule is shown in Figure 4.1. Some results with this asynchronous execution can be found in Table 4.5.

4.4 JNI interface and coupling to Java

The Java language has been written with a tight integration of existing C-libraries in mind. Therefore the creators of Java, SUN, integrated the keyword `native` into the language. With the `native` keyword methods with no method body can be declared. Calling these methods yields the call of a particularly named C-method. This method can access the members of the wrapping classes object as well as the parameters delivered by the method call. This gives the Java programmer the ability to call functions written in C. As C++ is also capable of defining C-style functions, it is possible to call a C-style function that in turn makes use of C++ objects. More so, Java objects can be handed over to the C-functions, which again can call methods of these Java objects.

This enables a communication between Java and C that is mutual. The JNI enables us to program an interface between the Java-based MATSim framework and the C++ based CUDA simulation. Unfortunately, the actual programming of this JNI interface is rather cumbersome. Therefore a toolkit to assist in implementing the proxy classes is of great benefit.

4.4.1 Toolkits for coupling C++ and Java

Two toolkits for binding Java and C through the JNI are dominant on the Internet. The first is SWIG (simplified wrapper and interface generator) [3]. SWIG is a general approach at binding different high-level languages to C/C++. An intermediate language definition file has to be written, from which interfaces to several languages can be generated. SWIG generates stubs into 18 different languages, some more prominent examples would be Python, Lua, Ruby or Java. SWIG is a very flexible toolkit but it seems that Java is not the primary target of the SWIG developers. Very little documentation was found on using SWIG together with Java. Also, the necessity to write and maintain an extra set of files with the intermediate language definitions in it seemed time consuming and error prone. A second package that restricts itself to support binding of Java and C/C++ only is *cxxwrap*[5]. It generates wrapper classes directly from C/C++ header files. This eliminates the need for writing extra files. Another bonus feature of the *cxxwrap* package is the ability to overload C++ methods in Java nearly effortlessly. This feature is used to transfer the CUDA events to Java, as shown in the next section.

4.4.2 Transferring events to the MATSim framework

With the *cxxwrap* toolkit overloading C++ methods in Java is implemented as follows. Every method which is declared abstract in a C++ class can be overloaded in Java. This is done by simply extending it from the proxy class

of the given C++ class and implement a method with the same name and signature as the C++ abstract method. *cxxwrap* will generate a stub method which tries to call the Java method every time the stub method gets called from within the C++ code. Obviously this makes transferring events from C++ to Java a trivial task. The C++ base class has an abstract method `writeEvent(int type, int agentId, int linkId, int legNumber)`.

Two classes were already derived from this base class in C++ overloading the `writeEvent()`method:

- `CMicroSimRingEvent2File`, which writes the information to a file, and
- `CMicroSimRingEvent2Null`, which just drops the information by doing nothing.

To create a Java implementation of the CUDA simulation, the new class `JCudaSim` is derived from the *cxxwrap* generated stub class `JEventRingSim`. Additionally the appropriate `writeEvent(...)` method is implemented in Java. This method is then automatically called, whenever the base classes implementation calls `writeEvent(...)` from C++.

As the events are delivered as a primitive data type, not much conversion effort is needed. The Java method uses existing code from reading the output of the *DEQSim* to actually generate Java events. The Java events are then fed into the regular MATSim event loop.

Listing 4.4 shows the relevant parts of the Java implementation. In the static part of the Java class the C-library is loaded. This guarantees that the native part is in place whenever a `JCudaSim` instance is created, but not needed or loaded otherwise.

4.4.3 Implementing wrapper classes for the plans

Unlike the transport of events into the Java framework, getting the selected plans from the strategical layer into the CUDA simulation without relying to file-IO is more complex. As file reading of the plans is based on the C++ version of the *DEQSim*, so are the data structures in which the initial plans data is stored. When switching from the *C++ only* implementation to a version also capable of getting the plans directly from Java's JNI some changes had to be made.

An abstraction layer was introduced, which resembles the original C++ data structures as much as possible but breaks up the structure far enough to act as an generic interface to the Java version. To exemplify this, Figure 4.2 shows this process for the class holding the trip information of one leg. An abstract base class `CTripBase` was introduced. It basically offers the same methods the original `Trip` class had, but reduces the interface to a minimum of necessary methods. Another class `CTrip` was written to act as

Listing 4.4: The Java wrapper of the CUDA simulation

```
public class JCudaSim extends CMicroSimRingEventJ{
  static {
      try {
        System.loadLibrary("TraSimGL");
      } catch (Exception e) {
        System.out.println("TraSimGL lib not found!");
      };
    }

  ...

  public void writeEvent(double time, int agentId, int linkId,
      int eventId) {
    Integer legNumber = legscount.get(agentId); // do this
        later on in CUDA
    if ((legNumber == null)|| (legNumber ==0)) legNumber = 1;

    // next leg bei agentdeparture event
    if(eventId == 6) legscount.put(agentId, legNumber + 2);
    else legscount.put(agentId, legNumber);// same leg

    if(this.events != null) {
      BasicEvent ev;
      if(eventId == 1) {
        stuck++;
        //this event is not handled by EventsReaderDEQv1
        ev = new AgentStuckEvent(time, Integer.toString(agentId
            ), Integer.toString(linkId), legNumber);
      } else ev = EventsReaderDEQv1.createEvent(time, agentId,
          linkId, eventId, legNumber);
      events.processEvent(ev);
    }
    eventcount++;
  }

  ...
}
```

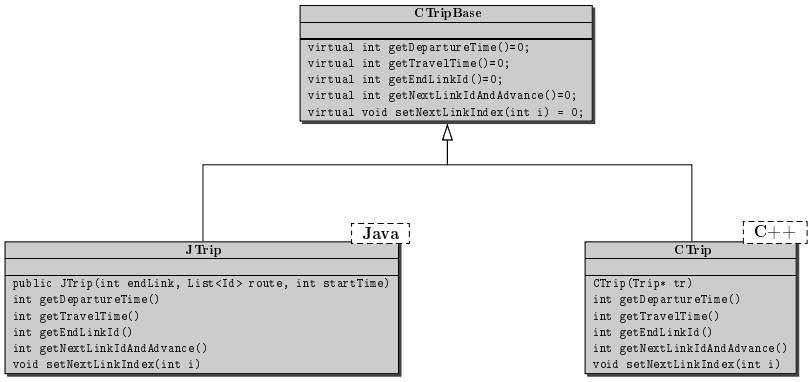


Figure 4.2: UML diagram of the relationship between the abstract and concrete implementations in Java and C++

a small wrapper for the original `Trip` class (the counterpart of the MATSims `Leg` class. This is done to avoid having to interfere with the original code of the *deqsim* implementation, so it is more likely to be able to upgrade to the next version rather painlessly. The three abstract classes are listed in listing 4.5

As already described, a Java class, derived from a C++ class containing abstract methods, can simply overload them. The actual implementation of a Java pendant to `CTrip` was thus straightforward. One main difference between the classes is the constructor that takes the actual structures to be abstracted from as an input parameter. Likewise, abstractions were written for plans and the complete population, `CPlanBase` and `CAllPlansBase`. Finally concrete implementations of these classes in C++ and Java were added, namely the classes `CPlan`, `CAllPlans` and on the Java side `JPlan` and `JAllPlans`. The Java's person data structure does not have a representation here, as the only person-related data needed in the context of the physical layer is the currently selected plan.

The plan conversion methods were rewritten using the above classes as *Template Pattern* and `CAllPlans::getNextPlan()` as *AbstractFactory Pattern* [41]. Finally the methods responsible for converting the plans into the CUDA format were refactored to solely rely on the abstract classes. After that it was completely transparent whether the plans were read by the Java or the C++ code.

Listing 4.5: The abstract classes for plans exchange

```
class CTripBase {
public:
  virtual int getDepartureTime()=0;
  virtual int getTravelTime()=0;
  virtual int getEndLinkId()=0;
  virtual int getNextLinkIdAndAdvance()=0;
  virtual void setNextLinkIndex(int i) = 0;
  virtual ~CTripBase() {};
};

class CPlanBase{
public:
  virtual int getAgentId()=0;
  virtual int getNumTrips()=0;
  virtual void perpareTrip(int i)=0;
  virtual const CTripBase* getTrip(int i) {perpareTrip(i);
      return trip;};
  void setTrip(CTripBase* tt) {trip = tt;};
  virtual ~CPlanBase() {};
protected:
  CTripBase* trip;
};

class CAllPlansBase {
public:
  virtual void restartAgentLoop()=0;
  virtual bool hasNextPlan()=0;
  virtual void prepareNextPlan()=0;
  virtual CPlanBase* getNextPlan(){prepareNextPlan(); return
      plan;};
  void setPlan(CPlanBase* pp) {plan = pp;};
  virtual ~CAllPlansBase(){};
protected:
  CPlanBase* plan;
};
```

4.5 Implementing further functionality on the CUDA device

Reducing the execution time of the physical layer will lead to a relatively higher contribution of the other components to the overall time the iteration takes to complete. By minimizing the execution time of the CUDA based mobility simulation, other parts of the calculation become prevalent and it is worth asking whether such components would be good candidates for implementation on the CUDA device as well.

Two distinct parts of the iteration, namely path generation and event handling are known to be rather time consuming. As an example of an area with potential for further optimization, the handling of events was examined. A simulation run will generate a large number of events on the course of execution. About 75% of these events are *LinkLeave/LinkEnter* pairs. These events are basically used to generate aggregated travel times for the links. The travel times will be used by the time-dependend router during the replanning process. The travel times are normally collected for certain time periods, e.g., every quarter of an hour, to keep the volume of data to be stored low.

To calculate the average travel times of a certain *time bin*, as these time period are often called, for any given link, the entrance time of all vehicles entering the link has to be stored until the vehicles leave the link again. At the moment of exit, the time difference can be calculated and stored in the appropriate time bin according to the entry time of the vehicle. This means that all incoming *LinkEnter* events need to be stored for later retrieval and mapped to the agents' IDs. When a *LinkLeave* event with a particular agent's ID occurs, the matching *LinkEnter* event needs to be fetched from the map so that the travel time of that agent can be calculated. Links on which the agent performed an activity need to be excluded from the calculation to avoid having the activity time added to the average travel time for that link.

The millions of store/retrieval pairs needed just to calculate the agents' travel times are rather time intensive. Therefore, it might be beneficial to have the travel times already aggregated in the mobility simulation, sending *aggregated travel time* events instead. This would load the burden of calculating the travel times upon the CUDA device, freeing the CPU from this task.

A first implementation in CUDA for a travel time aggregation proved to be disappointing. For a sample with 250.000 agents (25%) the number of sent events was reduced from about 12 million to a mere 3.2 million activity events, plus about 570.000 aggregated travel-time events. The CPU time for handling these events was therefore reduced by a few seconds, but the overhead for creating the events in the CUDA code cost more than the time savings.

Table 4.4 shows the running times of a second implementation. This im-

GeForce 8600GT (25%)	LinkEnter/Leave	link tt info	both
simulation	136s	142s	150s
event handling	46s	26s	28s
sum	**182s**	**168s**	**178s**
GeForce GTX280 (25%)	LinkEnter/Leave	link tt info	both
simulation	20s	24s	32s
event handling	46s	33s	34s
sum	**66s**	**57s**	**66s**

Table 4.4: Running times for different implementations for link time aggregation

plementation sets aside the complex and unfortunately very memory-access-intensive aggregation, sending the travel time of maximally one vehicle per link and time bin. For each vehicle leaving a link, the `moveNodes()` kernel identified the time the vehicle had entered the link. If the vehicle's agent executed an activity on the link, the entry time was set to -1, and the vehicle was ignored. If it had a valid travel time, the time bin according to the link enter time was calculated. Additionally, the last time bin for which a link's travel-time event had already been sent is stored for each link. If the link entry time of a vehicle was later than the end of that time bin, a new event was generated and the link's time bin was updated.

This is clearly not a perfect solution, but it is computationally less expensive and may be extended for use with more than one vehicle later on. The running times for this implementation can also be found in Table 4.4. The 25% sample was run on the slower system with the older GPU installed and also on the high-end configuration. The technical details of both systems can be found in Table 3.4 under the headings "LOW" and "HIGH". This time the implementation proved to be faster than the original implementation with the *linkEnter* and *linkLeave* events. On the other hand, the link events could be used for more than just the aggregation of travel times. In that case, they had to be generated in addition to the aggregated events. This seems to be a feasible option, however. As the last column in Table 4.4 shows, the creation of *linkLeave* and *linkEnter* events is not expensive compared to computing the aggregated travel time from these events. These number were computed by running the simulation two times. In the first run, events were created and the dismissed without further computations. This gives us the figure for the mere simulation and event cration times given in the row labeled "simulation". Then a second run was made, this time collecting all events in a list. Then the list of events was processes after finishing the simulation run. Only the time for processing the events was taken and listed in the row "eventhandling". The row "sum" shows the addition of both times. Between simulation run and the event processing a

GeForce 8600GT (25%)	LinkEnter/Leave	link tt info	both
overall time	**140s**	**170s**	**180s**
GeForce GTX280 (25%)	LinkEnter/Leave	link tt info	both
overall time	**68s**	**64s**	**70**

Table 4.5: Running times for different implementations for link time aggregation with CPU and GPU working simultaneously

Java garbage collection was enforced. This garbage collection took between six and nine seconds.

The speedup achieved does not scale linearly with the reduction of events. Although the event count was cut by more than half, the time to process these events was not. A closer inspection of the modules for event processing reveals that these modules completely relied on either the link-related events or the activity-related events. As the optimization only approached the link-related events, the runtime of the other modules remained untouched. Therefore, the possible speedup was restricted to the link-related modules which caused the asymmetry. Nevertheless, the running time of the event-related modules was greatly reduced, and putting the aggregation code on the CUDA side also means that it would benefit from faster CUDA devices. Reducing the number of events to be handled would additionally reduce the overhead of saving events to a file system, which is done by the frame work periodically. Another 30 seconds of computing time was saved through the reduced set of events when the events were written to a file.

4.5.1 Running event handling an simulation simultaneously

Finally another two runs were started. This time, event creation and handling were executed in the same loop, without artificially separating the two processes. This should benefit from the asynchronous execution of GPU and CPU as described in Section 4.3.1. As Table 4.5 shows, the regular version with the large number of events can benefit very well from the asynchronous execution taken, whilst the "tt info" version cannot benefit. This shows that the "tt info" is to a great extend GPU bound, whilst the regular version handles distribution of the workload more advantageous. These effect are more obvious on the system with the slower GPU as the CPUs of both systems are comparable, but the GPU is less powerful and "tt info" moves more work onto the GPU.

Chapter 5

Visualization and Interaction

An introduction to the general topic of visualization was already given in section 1.3. This chapter will discuss useful approaches to visualization of MATSim simulation runs. MATSim generates plans in an iterative process using evolutionary algorithms. The numbers from a simulation run can be validated against counting-station counts from real-world traffic. Computing a mean average error against real-world traffic can give further insights into the validity of a run.

These are crucial values to estimate the quality of a MATSim run. Nevertheless, it is important to have the option to look at the unaggregated simulation run to detect regions of implausible or bad performance. This can never be achieved with the aggregated data alone. One particular street link with an inappropriate capacity or some other small error in the map or the agents can substantially influence the aggregated data, thus ruining the populations performance. Therefore, visualizing the unaggregated data is a valuable tool for evaluating the actual simulation run and finding these small regions of erratic behavior. A second important function of visualization is certainly to visually reveal the impact of selected traffic measures to a broader public. Clarifying different effects of certain policies is more effectively presented with a visualization of the resulting traffic itself than by a column of numbers and percentages. So, the general benefits of a capable visualization are twofold:

- Bringing the developer means of easy and efficient debugging of the actual events taking place inside the simulation run.
- Displaying and visualizing the effects of measures in order to help policy makers and the broader public to gain a more intuitive understanding.

In the rest of this chapter the a visualizer will be defined with these general goal in mind. Additional subgoal and use cases will be defined to clarify the design choices. The next section will look into approaches

to visualize the output of simulations in the context of MATSim and of multi-agent systems in general. From these efforts the design goal for a new visualizer will be deducted and discussed. In the following section the necessary technical matter to implement these design goals will be discussed. The chapter's main section will then follow describing the actual design of the visualizer implemented. Finally, a conclusion is drawn and ways to further improve the design are shown.

5.1 State-of-practice in traffic visualization

In this section three visualizers used in the past to display MATSim results will be presented and discussed. In addition to that, two other multi-agent simulation systems and their visualizers will be described.

5.1.1 TRANSIMS

The multi-agent traffic simulation TRANSIMS [99] has been developed by the Los Alamos National Laboratory (LANL). It has been recently converted to an open-source and a commercial version. The development started in the 1990s and is still active. TRANSIMS is capable of demand generation as well as the simulation of the actual traffic using a cellular-automaton model. With the TRANSIMS package there comes the TRANSIMS visualizer *Vis*.

This visualizer is written in C++ and based on OpenGL. It reads a certain file format (an ASCII-file with row-oriented data in it) and visualizes the data found therein. The visualizer can change the colors and icons of the vehicles displayed based on the input file and some rules that are definable though configuration-files or the visualizer's GUI itself. The visualizer is not extensible by plug-ins or other code additions, but only by re-writing the monolithic application. The TRANSIMS visualizer's file format is output by the MATSim simulation by setting a flag in a configuration-file.

Discussion

The visualizer is written based on OpenGL, making use of the hardware acceleration of modern graphics devices and is therefore very fast. It is not a problem to display simulation runs with hundreds of thousands of agent. But apart from the raw speed the visualizer has not much to offer. It can only read "'post-mortem movie"' files and display these movies. The user can not ask questions about the state of each agent at a given time. Another drawback is the reduced capability to add extensions to the visualizer. These extensions, although possible, would have to be written in C++; given that all MATSim code is written in Java this does not seem an optimal situation. As most MATSim developers will want to extend the visualizer to their specific need, a change from MATSim's core language Java towards C++

seems unfavorable. The visualizer is not easy to maintain or port to other operating systems.

5.1.2 Visualizer of Christian Gloor

Christian Gloor [44] wrote a visualizer to display the results of his multi-agent simulation of hikers' movements in the open terrain of the alps. He uses many concepts underlying the MATSim framework to implement the travelers' choices and demands, as well as choosing and executing a preferable route from a mountain top to down in the valley.

He takes the notion of the two layers (strategical and physical) used throughout the MATSim framework, as presented in Figure 2.2 in Chapter 2, and extends it to segment the complete application into different modules. These modules interact through network communication. Therefore, it is possible to run each module on different computers connected through a network. This is also applied the visualizer.

Gloor introduces a visualizer module, which can be run on a computer separate from the actual simulation. This offers some benefits. First of all, dividing the extra computational load for rendering a visualization is taken from the simulation computer, leaving more room for larger scenarios. Second, more than one computer can visualize results. Any number of computers can connect to the simulation computer and render different visualizations of e.g. different areas.

One might implement visualizers with different perspectives, e.g. one with an "'ego-perspective"' of one particular agent and a 2D top-down view of a whole area. Gloor understands the visualizer modules as one another with many modules that might possibly attach themselves to the flow of events outpouring from the actual simulation. All of the modules participate in the simulation process taking place. It is even possible to add events before they enter the strategical layer again. Gloor demonstrates this by one visualizer module that enables the user to "draw" a number of "rain-events" into a landscape. These "rain-event" are then fed back into the strategical layer in an instant, yielding changes in the agents' plans.

Gloor uses low level network connections like TCP or UDP connections as well as UDP's broadcast modus to issue the events from the simulation. This low level functionality has to be ported for each operating system the simulation should run on. This proved to be a rather tedious and brittle strategy, as the implementational details proved to be different on the different operating systems, even if one merely wanted to use a different UNIX-system.

Discussion

Gloor introduces many interesting design decisions. The most outstanding achievements are:

- The option to interactively add events,
- The possibility to connect to a remote simulation,
- The option to render different aspects of one simulation simultaneously
- The option to use different renderers,
- The option to render and visualize the same (remote) simulation on different computers in parallel.

These options offer a wide range of use cases and re-usability of the the same modules for simulation. Unfortunately, there were some drawbacks as well. The need to handle platform-specific implementational details, like the specific implementation of sockets for network connections, for each supported operating system, turned out to be rather cumbersome. Additionally, like with the TRANSIMS visualizer, the implementation was based on C++, therefore not the first choice of MATSim's Java developer to program in and extend the visualizer. So, although giving many useful suggestions for implementing a visualizer, to start from the actual source of Gloor seemed not a viable decision. Furthermore, the experiences from the usage of low-level networking API was not a completely satisfying one, so it seems a good idea to be looking for a networking solution that promises to work "'out-of-the-box"'.

5.1.3 MATSim's NetVis

NetVis is a Java-based effort to visualize networks and agents. It has been implemented together with the re-implementation of the queue model in Java. The NetVis visualizer is based on the Java SDK's SWING classes. The SWING API offers several classes for drawing 2D elements and building a GUI for the input of values. The NetVis implementation was built with some flexibility in terms of the transmitted data in mind. It is possible to change for example the color-represented value that makes up the color of a link from the link's load to the average speed driven on the link. The NetVis visualizer's input file format is natively written by the MATSim simulation. Only these "'post-mortem"' files can be read into the NetVis and then be played.

Discussion

The main benefit of the NetVis visualizer is the fact that it is written completely in Java. Therefore, extending the visualizer should be fairly easy

for all of MATSim's Java programmer. It is extensible in terms of the data format written and already had some customizations to offer in terms of what to visualize. Nevertheless, it suffers from one big disadvantage: It was written in SWING, which is notoriously slow. The NetVis visualizer could only cope interactively with networks with a few thousand links and nodes as well as a few thousand agents, which is too little for MATSim's large-scale simulation scenarios. In addition to that it was lacking the possibility to look into the actual running simulation.

5.1.4 Other multi-agent simulations

The author has looked into two other multi-agent simulation packages to investigate their visualization capabilities. REPAST [83] and MASON [65] are two well known simulation frameworks. Both are based on a Java implementation.

The REPAST package proved in particular to be rather GUI bound. There is a good GUI package to design own input mechanisms to provide the user with ways to change the outcome of a simulation run by means of a GUI. This GUI even offers the possibility to "'look into"' each agent at random depth by utilizing the Java's Reflection API. This is a rather fascinating way to get to know the inner state of an agent. Unfortunately, REPAST does not offer support for execution of large-scale scenarios like MATSim does. It seemed unfeasible to visualize more than a few thousand agents at a time.

MASON seems to be a well thought package using Java2D as well as Java3D graphics to render contend on screen. As Java3D is known to be capable of making use of hardware acceleration, it seems that MASON should have sufficient resources to display event large networks and large numbers of agents. Unfortunately it is a rather monolithic package.

Discussion

REPAST just does not have sufficient power to visualize the amount of agents, links and nodes necessary fro the typical MATSim scenarios. The MASON package looks more promising on this behalf. But it seemed rather difficult to extract the visualizing parts from the rest of the simulation package. To fuse the existing MATSim code with MASON would have been a rather challenging undertaking, including the necessity to keep track of it for the lifetime of both products.

5.2 Goals and motivation

From the above findings, the author tried to induce a set of *use cases* to achieve a mixture of all good aspects of the visualizers examined.

The TRANSIMS visualizer does not have much to offer. But one thing is really good about it, it is fast. Therefore, implementing the visualizer in such a way that it would benefit from hardware acceleration seemed mandatory. From Gloor's approach many fruitful discoveries arose. The simulation should be running independently from the actual view onto it. More so, it should be possible to run the simulation on a remote computer and visualization on another one. Furthermore different computers should be able to visualize the simulation concurrently and different aspects or regions of the simulation should be accessible in parallel. Last but not least, the ability to interact or to send queries into the running simulation seems rather rewarding (and mandatory for the overall goal of being a debugging tool). From NetVis, finally, the implementation in Java seemed attractive. To use the same language as the MATSim core should reduce the barrier for the MATSim programmers to implement own ways to visualize the data. With these insights in mind, a list of possible design goals was written. This will be discussed in the next section.

5.2.1 An on-the-fly visualizer for the MATSim framework

The design goals for the new visualizer based on our insights from the survey above are listed below:

- Abstract data source (data collection) from data display (visualization)
- Easy to extend with own data types
- Simulation can be accessed over a network
- Simulation can be run locally on desktop computer
- Reduce the amount of sent data to a minimum
- Visualization can connect to running simulation (on-the-fly)
- Minimally-invasive to existing MATSIM code
- Fast enough for large scenarios
- Visualization can read from post-mortem dump (mvi-file)
- Read existing file formats (event files, T.veh files)
- Network connection, optionally ssl-encoded
- Multiple different views onto one data set
- Simple network infrastructure, easy to transfer to different connection types

- Take advantage of hardware support for drawing

With these design goals in mind, it should be possible to implement a flexible tool for both inspecting the inner states of a running simulation (e.g. for debugging purposes) and presenting an output of the simulation in a form easy to comprehend for the general public or decision makers.

Four different design principles and techniques were selected to optimally facilitate these design goals. First of all the MVC-pattern. It is the single most important design pattern when it comes to implementing a GUI in general. In our specific case it supports the modular approach that proved beneficial in Gloor's implementation. Also, choosing a multi-tier approach, namely a "'server-client"' architecture to separate data sources (the simulation) from the actual visualizer instances arise from the Gloor approach. Finally, two rather specific APIs were chosen to support the implementation of an open system. Java's own RMI (remote invocation interface) offers easy to establish, platform-independent network connections freeing the programmer from the hassles of low level network programming. OpenGL is the one platform independent interface for using graphic hardware to speed up rendering. It would have been a feasible way to choose Java3d for making use of hardware acceleration, but Java3d does not offer the high degree of freedom the OpenGL API offers and, last but not least, the author was more familiar with the OpenGL API.

Here is a short overview of which principle/technique supports which of the aboves goals, some goals are supported by more than one principle:

- MVC Pattern (Model-View-Controller)
  - Abstract data source (data collection) from data display (visualization)
  - Easy to extend with own data types
  - Minimally-invasive to existing MATSIM code
  - Multiple different views onto one data set
- Client-Server architecture
  - Simulation can be accessed over a network
  - Simulation can be run locally on desktop computer
  - Reduce the amount of sent data to a minimum
  - Visualization can connect to running simulation (on-the-fly)
  - Minimally-invasive to existing MATSIM code
  - Multiple different views onto one data set
  - Simple network infrastructure, easy to transfer to different connection types

- Java's RMI (Remote Method Invocation) interface and Serializable
  - Visualization can read from post-mortem dump (mvi-file)
  - Read existing file formats (event files, T.veh files)
  - Network connection optionally ssl-encoded
  - Simple network infrastructure, easy to transfer to different connection types
- OpenGL
  - Fast enough for large scenarios
  - Take advantage of hardware support for drawing

The MVC (Model-View-Controller) pattern

Though not part of the design pattern introduced by the "gang-of-four" [41], the Model-View-Controller (MVC) pattern is deemed one of the most important patterns when it comes to document-based design or simply any application with a dedicated User-Interface (UI). It was introduced in 1988 by Krasner and Pope [61] describing ways to design user interfaces for *Smalltalk-80*.

The MVC pattern describes a 3-tier applications layout. A model is defined as the actual application data to be displayed. The view is a representation of this data for on-screen presentation and the controller is responsible for the interaction of the user input and the application's data. As the MVC pattern is fundamental to nearly every GUI based application nowadays, it could also be applied to the new visualizer. In terms of the *OTFVis* the model consists of the simulation itself together with the server code. The controller is given by the client code and the classes responsible for user input and the view can be any implementation of the actual drawer classes, i.e. the OpenGL drawer or the SWING based drawer. Figure 5.1 illustrates this.

The MVC pattern itself could easily be decomposed into smaller patterns, as done by Gamma et al.[41]. Regarding to Gamma, the MVC can be decomposed into a selection of observer, decorator, factory method and composite patterns, many of which where also engaged in designing the architecture for the *OTFVis* visualizer. The pattern will be discussed in a later section.

Client-Server

Designing the new visualizer as a client-server architecture suggested itself based on the two design goals to separate data display from data collection and to be able to connect to a running simulation. This architecture is often found in literature when implementing visualizer applications. Paradise

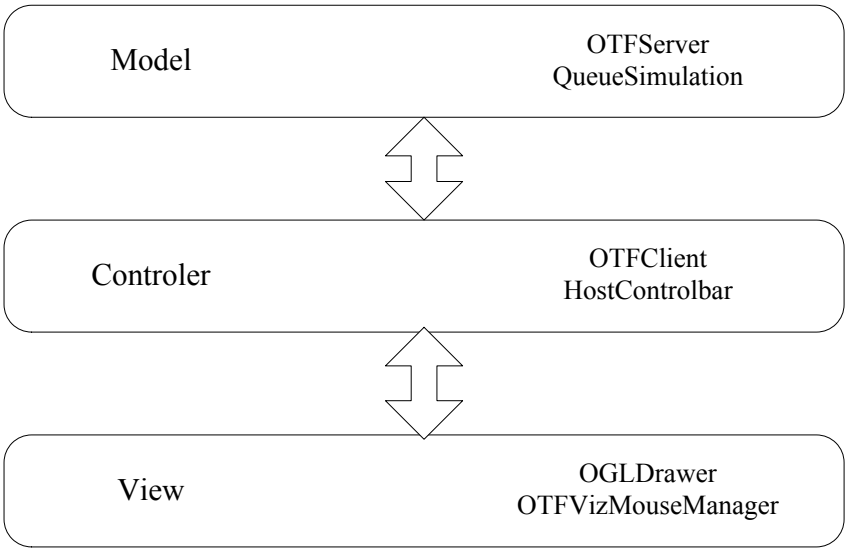


Figure 5.1: The MVC pattern for OTFVis

[35] is an example for a GIS application based on client-server architecture, Treinish et al. use it for data flow visualization [6], multi-user environments [40], streaming 3D graphics [86] or CAVE architectures [90].

Often a scenario is too large to be run on a desktop computer. To have the ability to connect to these simulations running on a remote computer a client-server-architecture was mandatory. Apart from that, separating the code responsible for displaying the actual data from gathering the data makes it easier to adopt to new technologies in visualizing hard- and software.

Java's RMI and Serializable Interface

RMI (Remote Method Invocation) and the *Serializable* interface are parts of the Java SDK. These were implemented for easing the effort to design application that can be run from remote computers over the Internet. RMI greatly reduces the overhead necessary to send data over a network connection, making it possible to use things like SSL-connections in a transparent way. The `Serializable` interface of Java plays an important role in preparing objects to be either sent over a network connection or be saved and re-created to and from a file. The latter one is used for creating the post-mortem *movie* files to be read from. RMI will be discussed in depth in

Section 5.4.3.

OpenGL

Over the last decade hardware-acceleration for 3D graphics has become a matter of course for modern computer systems. Even notebooks are most likely assembled with some sort of hardware-acceleration. For accessing these hardware features the OpenGL [76] interface is most common when using multiple operating systems. The OpenGL interface introduces an abstraction layer from the actual hardware present. Therefore the programmer does not need to interact with different hardware in terms of driver programming but can program it once for the OpenGL interface and –hopefully– run it everywhere. The OpenGL interface is supported by all relevant hardware vendors. For using OpenGL with the Java language, another small interface is needed to bridge from Java into the C -based OpenGL. This is supplied by the JOGL wrapper [4].

Apart from the OpenGL based visualizer a second client, capable of displaying data based on the 2D SWING package, was implemented, albeit with the drawback of being slow. It serves as an alternative for displaying at least smaller scenarios on computers not capable of OpenGL.

5.3 Architecture of an interactive system

Apart from the server-client architecture, other qualities are vital to interactively display large scenarios. In terms of data transport it is mandatory to reduce the amount of sent data to the bare minimum necessary to still attain the favored informative value. Three different constellations of data transfer had to be taken into account:

- Massive amounts of very simple data (e.g. positions of all agents)
- Small amounts of large data sets (e.g. user generated queries)
- Singular amounts of constant data (e.g. the network topology)

With dividing the data into these categories it is feasible to maintain the necessary frame rates for interaction while still being able to handle large scenarios.

5.3.1 Data formats and interfaces

To implement the above mentioned modes of transport for the data from server to client, three distinct ways to ask the server for data were implemented. First, directly after the client-server connection is established, all constant data is transfered at once and for one time only. After that, the server will transport dynamic data to the client, whenever the simulation

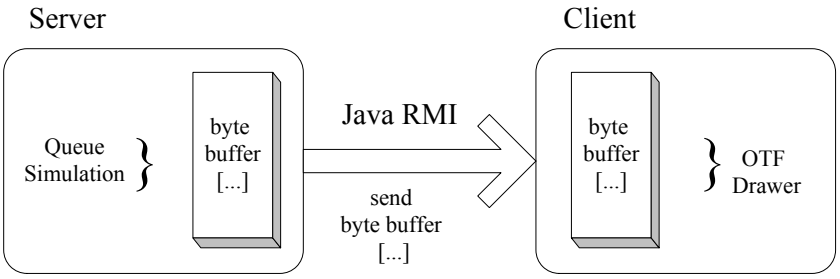


Figure 5.2: Sending dynamic data from server to client over RMI using byte buffers

changes. Additionally to that a third mode of operation is determined by sending `OTFQuery` instances from client to server and vice versa. These three modes of transport will be discussed in detail in later sections. The first implementation of the necessary data structures for serializing data to a stream was done by serializing the extracted data to a file stream. To implement this, a byte buffer was filled with the data. Everything was dumped into a byte buffer and read from it on the client side. This way of transporting data was maintained throughout the further development of the on-the-fly version of the server, as it turned out to be the fastest way to transport large data sets over the RMI interface. The RMI framework used for transporting the data over the network will be discussed in Section 5.4.3. For the less time critical parts, like the initial sending of the data structures and the sending of `OTFQuery` objects, the more convenient RMI object proxy methods were used. The basic outline of the transportation is illustrated by Figure 5.2.

Writer and Reader

All data extracted from the simulation for visualization is written to a byte array as described in the last section. To maintain the highest possible speed and the minimum memory footprint this data is written sequentially into a byte buffer, with no separators between the different sections. Apparently it is most imperative to have readers reading the exact amount of data from the stream as the distinct writers have written to the stream. It is up to the programmer to maintain this synchronicity. In case of our Java implementation this is achieved by implementing the writer class as an inner

class of the reader class, combining both codes for writing and reading data in the same text file, therefore, maintaining the consistency of both should be fairly manageable.

Apart from having all readers and writers running analogously the same sequence of writer/reader pairs has to be called. To achieve this, a data structure for easy access to rectangular region has been implemented. Two exact copies of this structure are being maintained on each client and server side, except for the server's side holding the writers and the client's side holding the readers. Whenever a region needs to update, the sequence for going through the writers is guaranteed to concur in the sequence of the readers. The data structure used is a *quadtree* and the data exchange is discussed in detail in Section 5.5.1.

Receiver

After submitting data from the simulation to the visualization it needs to be drawn. The classes responsible for drawing data have two distinct responsibilities: First, they need to be able to receive data from the reader classes. Secondly, they need to be able to draw on-screen or transmit the received data to classes capable of drawing on screen. The reader classes have to agree with a set of receiver/drawer classes over which data needs to be exchanged. This is done by employing the Java's *interface* paradigm for describing the interaction. This offers the benefit of being flexible in buildup but still being very fast in execution, as calling an interface method is as expensive as calling a virtual method. The implementation details are discussed in Section 5.4.3.

Visualizer

There is no distinct class being the visualizer class. The visualizer consists of different modules working together to provide the on-screen display. Apart from the drawer classes mentioned in the last section, the visualizer will have to provide a framework with elements for user interaction as well as the interaction with the server and a canvas for the drawer to paint on. Different elements for interaction are needed, depending on the source of data provided. The actual implementation of this will be discussed in Section 5.5.2 and Section 5.4.1.

5.3.2 Quadtrees for spatial data storage

Quadtrees have been studied thoroughly in the domains of GIS and computer graphics. In 1974 Finkel and Bentley [39] introduced the notion of a quadtree to sort spatial data. It has proven itself a valuable tool for storing spatial data since then. Many variations of the quadtree can be found. Samet [89, 88] gives a very exhaustive overview of derivations from

the quadtree. Many variants on the quadtree were implemented for the different spatial structures, like the R-Trees [85] or PR-trees [77]. In this thesis a PR-quadtree is used. A PR-tree is a tree structure that stores two-dimensional points respecting their spatial position. It is a specialized version of the rectangular quadtree presented by Klinger [60] .

A quadtree divides a two-dimensional space into rectangles. These rectangles are stored in the tree structure. Each node of this tree agrees on the following two conditions:

- A *leaf*, i.e. a node with no child nodes, can hold up to a predefined number m of objects.
- An *internal non-leaf node* has exactly four child nodes, dividing the quads space equally. One might call them NorthEast, NorthWest, SouthEast and SouthWest.

Whenever an element is added to a leaf, so that the maximum number of objects is exceeded, this node becomes an internal node and its objects get distributed among the new child nodes according to their position. A search for all elements of a rectangular area becomes easy with a quadtree. One needs to find the uppermost node fully containing the sought-after rectangle and then step through all objects in all leaves of this tree section. A quick test can be performed to test if a child rectangle is fully contained in the questioned rectangle, in which case all objects can be added to the result set. Only if the node's rectangle is included only partially, additional comparisons have to be made.

Fig. 5.3 shows a space decomposition and the associated quadtree.

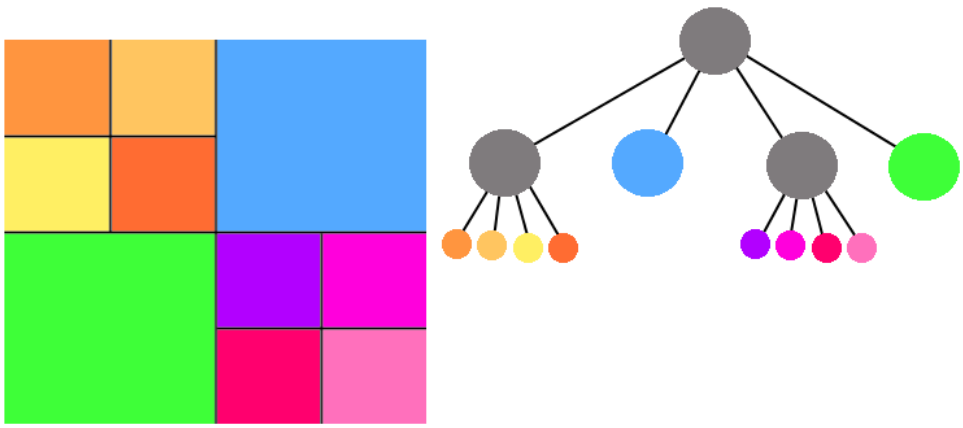


Figure 5.3: Space decomposition and quadtree

All other branches of the quadtree do not need to be inspected at all. The quadtree is well suited to cover areas with heterogeneous space assignment. Other than with a grid structure of tiles covering the area in question, the quadtree has a predefined maximum number of objects per leaf node and is able to cover areas with many objects with a finer granularity than inane areas. This greatly reduces the storage demand of the quadtree compared to a highly resolved grid of tiles.

For implementing a quadtree for handling the data necessary for the visualization an already existing quadtree implementation was used. Albeit, the implemented quadtree being a point quadtree, hence not capable of storing outspread data, like street sections, it seemed sufficient for our purpose. Street sections (link) were inserted into this point quadtree using their respective center point.

The quadtree is known for its $O(logn)$ behavior in terms of data insertion and retrieval. Only the deletion of nodes is expensive. Therefore the quadtree is best used for data that is mainly static. In the *OTFVis* implementation it is used for storing the nodes and links of the network underlying the simulation run. This network is known to be unchanged for the entire simulation and, thus, a good candidate for being maintained in a quadtree.

One of the great benefits of a quadtree is that the quadtree simplifies spatial range queries. These are basically ubiquitous in the domain of visualization. Every time the user zooms into a certain range of the network, it is necessary to re-collect all data that resides in this range. Optimizing the visualization for speed means to be prepared to transmit the smallest possible amount of data that still fully represents the area to be drawn. The quadtree is used for finding this smallest amount of data. The line data is stored with their respective center point. This is not the optimal implementation, a quadtree representing line data like the PM or PMR quadtrees [74, 75] would have been a better choice for the links, but as no performance loss was perceptible from the point quadtree and a second data structure had to be maintained in the other case, the links were left in the point quadtree. This could be a further improvement to the *OTFVis* implementation.

5.3.3 Patterns used

Gamma [41, 42] first used the notion of "design pattern" in computer science to describe and name recurring ways of accessing and deconstructing many problem domains. Since then design patterns have proven themselves to be a reliable way to communicate architectural issues. In this section the primary pattern used to design the *OTFVis* application will be discussed.

Proxy

For any client-server architecture the *proxy* pattern is vital. The proxy pattern describes a class acting as surrogate or placeholder for another object, to exhibit ways to control the other object. In the context of the *OTFVis* the proxy pattern is used by the *Java RMI* networking SDK. Virtual instances of the server's quadtree on the client side or the client's queries on the server side are good examples of proxy classes.

Factory method

The *factory method* pattern describes a way to create new objects without detailed knowledge about the object's class. This pattern is used heavily in the context of creating the reader and writer classes for the simulations objects, as well as in creating the receiver and drawer instances on the client side. This pattern made it possible to design a hierarchy of responsibilities for streaming the data from the simulation over the server and the client to the actual displaying class, without having to make any decision about what data should be transferred and in which way the data should be visualized. These pieces of information are provided in terms of a "late binding" as described in Section 5.4.3.

Visitor

When further functionality needs to be added to objects contained in another data structure without actually changing or inheriting the data structure, the *visitor* pattern is useful. It specifies how an external class can be executed upon by members of another data structure. It is used together with the quadtree implementation. Each quadtree can "execute" visitor objects on rectangular areas of its member objects. In this way any future operation can be implemented on the quadtree's elements, without changing the actual quadtree code.

Bridge

The *bridge* pattern is the pattern equivalent to Java's *Interfaces*. As the interfaces are the most used structural elements in nowadays Java code, the bridge pattern is used throughout the client and server code of the *OTFVis*.

Decorator and Observer

Again, the *decorator* pattern is constitutional for a part of the Java framework. The *SWING GUI Framework* used in the visualizer to implement all elements of the graphical user interface, i.e. all buttons, drop-down lists, text fields, etc, is based on using decorators to add functionality to the GUI's

objects. Together with the decorator pattern, the *observer* pattern is used in the SWING classes to maintain ways to address objects, when they need to update or redraw.

5.4 Implementation

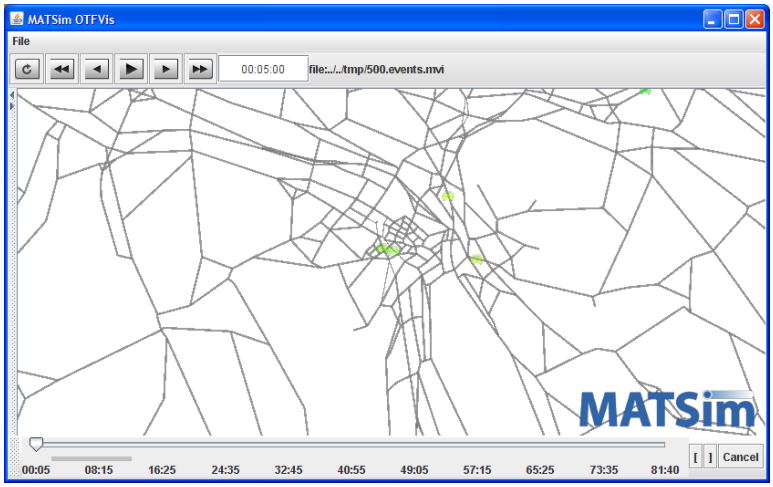
This section will describe the classes actually written for implementing the architecture described in the last section. Several different variations of server implementations needed to be written for serving data from the various sources. Only two different clients were implemented, based on either Java's own graphics library SWING or on the OpenGL graphics library. The SWING version of the client is very reduced in its capabilities and capacity; it can merely display a small network with agents. Its main purpose is to have a fall back solution for users with a computer not capable of displaying OpenGL graphics. Another aspect of implementing a SWING client was to serve as a proof of concept that it is feasible to build different clients without having to implement special servers for these clients.

5.4.1 Client implementation

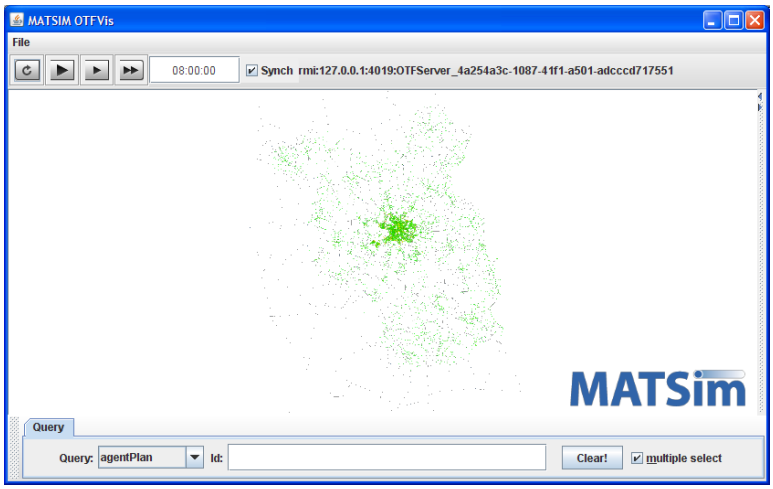
The client is responsible for several different tasks. It will initiate a connection to the appropriate server. The server's address is given to the client by an unique *URL*. This will be discussed in more detail in Section 5.4.3. The client will manage a connection to the server with the aid of an `OTFHostControlBar` instance. Depending on the chosen server, the client's visual appearance will differ. When reading from a file, the client will not be capable of interrogating the server about the inner state of the simulation or the agents, as it is only a post-mortem – one might say *read-only* – view onto the simulation run. Therefore, the toolbar responsible for doing queries into the server, the `OTFQueryBar` instance, will be missing. But instead, there will be an `OTFTimeLine` instance. This enables the user to scroll through the actual time line of a pre-recorded simulation run, as the complete time structure of this run is already known. When running an on-the-fly-version of the server, the end-time of the simulation is not known, making the drawing of a timeline impractical. Nevertheless the user can step forward in time by using the controls given by the `OTFHostContorlBar`. Figure 5.4 depicts the two different looks of the OTFClient.

OGLClient

The `OGLClient` class is an implementation of a client based on the OpenGL graphics interface [76]. This framework is the industry-standard framework for displaying 3D-graphics. As on modern computers most 3D-graphics



(a) Reading from file



(b) on-the-fly visualization

Figure 5.4: Different layouts of the OTFVis.

class	function
interface OTFGLDrawable	interface for OpenGL related drawing
OTFGLDrawableImpl	default implementation of the interface above
OTFOGLDrawer	main class for drawing on OpenGL canvas
SimpleStaticNetLayer	draws a net without dynamic colors
ColoredStaticNetLayer	draws a network with dynamic data
OGLSimpleBackgroundLayer	the layer for background images & features
QuadDrawer	draws link with capacity based data
QuadSpeedDrawer	draws link with average speed based data
OGLAgentPointLayer	draws the agents / cars
AgentArrayDrawer	optimizes agent drawing with vertex lists
AgentPointDrawer	relies on above to draw agents
AgentPadangTimeDrawer	specialized version of above
AgentPadangRegionDrawer	specialized version of above
SimpleQuadDrawer	receives and draws one rectangle
NoQuadDrawer	receives but does not draw a rectangle
SimpleBackgroundDrawer	draws bitmaps as background pattern
SimpleBackgroundFeatureDrawer	draws geotools features as background

Table 5.1: List of OpenGL related classes

rendering is supported by dedicated hardware, this 3D-graphics is tremendously fast compared to traditional CPU based rendering. This holds true not only in terms of three-dimensional rendering but also when it comes to high-performance 2D-rendering. The OpenGl-based client outperforms the SWING-based client by magnitudes. Given the broad circulation and superior performance of OpenGL, this client was chosen as the standard implementation of the more sophisticated features of OTFVis. The`OGLClient` still uses the SWING-based components for drawing the user interface. Only the visualization of the simulation is rendered in an OpenGL context. This affects the class handling the overall rendering of the screen `OTFOGLDrawer` as well as all related layer and drawer classes. A list of implemented classes for OpenGL-based rendering is given in Table 5.1. Apart from these classes, all queries (as discussed in Section 5.4.1) include code to render to OpenGL contexts.

SwingClient

As mentioned in the introduction, the SWING version of the client does not have much functionality at hand. It is based on Java's own SWING toolkit, provided with the Java SDK. Its main benefit is that SWING is guaranteed to be existent whenever a Java SDK is given, therefore, it is as platform independent as Java itself. The main drawback of the SWING im-

plementation is being too slow to effectively visualize the large networks MATSIM is used for. Still it can be used for smaller networks. This client is implemented in the class `OnTheFlyClientQuadSwing` in the package `org.matsim.utils.vis.otfvis.executables`.

Interaction

The user can interact with the simulation in multiple ways. The most intuitive is the interaction with the view via mouse handling. With the mouse the user can zoom and pan the viewed region. He/she can also save the current view settings for later reuse. Preferences for agent size or link width and many more can be saved either to a *.mvi* file or to a separate *config*-file.

Depending on the capabilities of the server implementation the *OTFVis* differs in the interaction capabilities offered. When the server executed is not a *live-server*, the interaction is rather restricted. No queries into the data base of the server can be issued, as the server can only rely on the pre-recorded data at hand. Anything that was not recorded at runtime can not be retrieved anymore. Albeit any information provided by a writer/reader pair can also be present in a pre-recorded session, the ability to actually query the agent database is missing. Nevertheless, some additional features are implemented for the non-live server. As the server already knows how many time steps were recorded during the simulation, it can present the user with a time line, representing the whole simulation's recording. The `OTFTimeLine` class provides a convenient time line representation as illustrated in Fig. 5.5. It offers the functionality to use a slider for stepping in time as well as buttons to cancel the reading of the pre-recorded data into a RAM buffer. It is also possible to specify bounds for looping the animation.

Figure 5.5: The timeline bar for user interaction.

When a live simulation is run, the end time of that simulation is not known, therefore, the time line is replaced by a different toolbar. With the `OTFQueryControlBar` the user can issue queries into the running simulation. This powerful mechanism will be discussed in depth in Section 5.4.2 and 5.4.2. An illustration of this tool bar can be found in Figure 5.6

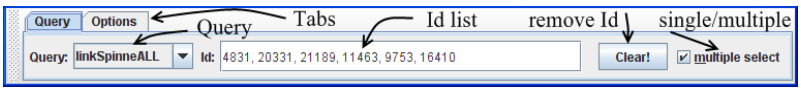


Figure 5.6: The query bar for user interaction.

All queries will be collected in the tool bar's drop-down list box. Selecting a particular query might result in additional GUI elements to provide the user with more options for this query. A query can indicate the need for an additional option pane by implementing the `OTFQueryOptions` interface shown in listing 5.1. In case of additional options a new tab is added to the query tool bar. The component returned by the `getOptiondGUI()` method will be included into this tab called "Options". The *Id list* shows all selected identifiers for the particular query's selection type. The *"Clear"* button clears the selection. The check box on the right hand side indicates whether multiple *Id* selection is enabled or not. Clicking into the view will select the appropriate object nearest to the mouse pointer. If multiple selection is not enabled each click will clear the previous selection.

Listing 5.1: Interface for a query options tab

```
public interface OTFQueryOptions {
  public JComponent getOptionsGUI(JComponent mother);
}
```

Mouse handling

As seen in the last section, not only panning and zooming is done via the mouse or touchpad input device, but also selecting entries that will be sent to the `OTFQueryControlBar`. Two interfaces are responsible for passing the mouse events around. The handling of any mouse action is done by the `VisGUIMouseHandler` which inherits respectively implements the class `MouseInputAdapter` and the interface `MouseWheelListener`. This object holds a reference to an object implementing the `OTFDrawer` interface. This handler takes care of all general actions regarding changes of the viewable region. It will call the `redraw()` or `invalidate()` methods of the `OTFDrawer interface` whenever it suits the situation. In addition to that, the `OTFDrawer` interface ensures the presence of the following two methods:

```
  public void handleClick(Point2D.Double point, int mouseButton
      , MouseEvent e);
  public void handleClick(Rectangle currentRect, int button);
}
```

These methods are called whenever the mouse handler is instructed to select a point or a rectangular region. The selection is then passed along to whatever `OTFDrawer` implementation is in charge of the actual view. The drawer decides whether it can handle the request on its own or passes the click on to an object implementing the `OTFQueryHandler` interface. This interface, again, includes the following two methods:

```
  public void handleClick(String id, Double point, int
      mouseButton);
  public void handleClick(String id, Rectangle2D.Double
      origRect, int button);
}
```

The additional `String id` indicates which particular view received the request. This information might be useful for choosing the correct server, although in the actual implementation only one server per *OTFVis* instance is used.

5.4.2 Server implementation

One aspect prominent in the design decisions was to provide the possibility to read data from multiple input sources. Movie files written during a simulation run, conversions of event files or *t.Veh* data of the old TRANSIMs format should be visualized as well as the actual running simulation. Thus different classes were implemented providing these servers. Only the visualization of events files was implemented differently as discussed in the following section. Providing different sources for visualization is an important goal as it improves flexibility in case the complete set of data is not given. Many older runs of the simulation were only run with *T.veh* output activated. Maintaining the ability to compare these older runs to more recent ones by providing ways to visualize the older runs the same way as the newer runs helps to classify them. All servers implement a common interface that enables any client implementation to communicate with each of them. The server interface *OTFServerRemote* is illustrated in Fig. 5.7

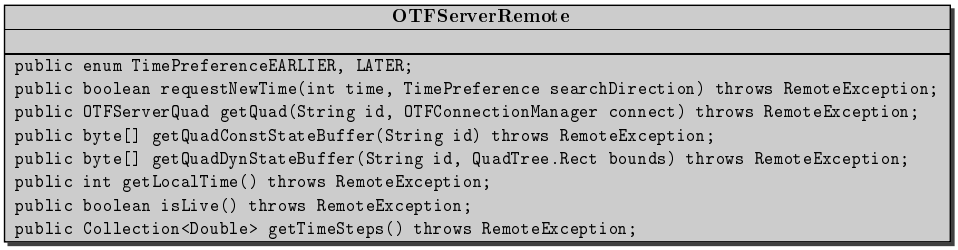


Figure 5.7: UML diagram of the OTFServerRemote interface

The method `requestNewTime()` is an input into the actual server implementation. The client can state what time steps representation it would like to receive by the next call to `getQuadDynStateBuffer()`. The server might not be able to actually provide the exact time step, so additionally the client can state whether it wants to receive the next earlier time step or the subsequent time step after the requested one. This feature had to be

provided as there is no single obvious solution to what time step is wanted in absence of the actual requested time step. If, for example, one wants to jump to the beginning of the simulation, the first time step *later* than midnight will have to be targeted. On the other hand, if one wants to inspect the simulation at a distinct time of day, this user will most likely want to start at the first time step previous to the requested one. The actual time the server has chosen can be requested with the method `getLocalTime()`.

The method `isLive()` will tell the client, if this server is a running simulation or a prerecorded movie. In the latter case, all time steps stored in the record can be retrieved with the `getTimeSteps()` method. This is used for pre-caching data from movie files. A live server returns `null` instead.

The method `getQuad()` is used to receive a copy of the server's data structure representing the spatial data of the simulation. This contains all server-side handlers responsible for sending data to the client. This method has to be issued in the very beginning of the client-server conversation. Directly after receiving the quad data structure, it is converted to a structure holding the receiving handlers that complement the server's handlers. Now server and client can exchange data. The first data to be exchanged is the *constant data*. This will include all data that stays unchanged over a simulation run and therefore needs to be transfered only once.

Subsequently, the server is waiting for a sequence of calls to the methods `requestNewTime()` and `getQuadDynStateBuffer()`. The process of exchanging data between client and server is illustrated in listing 5.2.

The actual implementation of the different servers will be discussed in the following sections.

Server for OTFVis movie files

The most "natural way" to visualize a simulation run is to provide a *.mvi*-file, rendered during the actual simulation run. The Java version of the simulation is capable of providing the dumping of the events of a simulation run into so called *snapshots* in different formats. Possible snapshot formats are:

- *transims*, the old TRANSIMS format (*T.veh*)
- *googleearth*, the format necessary to display runs in *Google Earth* (kmz)
- *netvis*, the format of an older visualizer
- *otfvis*, the OTFVis movie format

Any combination of these snapshot formats can be written by implementations of the interface *SnapshotWriter*. The corresponding implementations are:

Listing 5.2: Initializing and updating data in OTFVis

```
public class OTFClient{
private OTFServerRemote server;
...

  public void initializeConnection(OTFServerRemote server) {
    this.server = server;
    OTFServerQuad servQ = host.getQuad(id, connect);
    OTFClientQuad clientQ = servQ.convertToClient(id, host,
        connect);
    clientQ.createReceiver(connect);
    clientQ.getConstData();

    hasNext  = server.requestNewTime(0, OTFServerRemote.
        TimePreference.LATER);
    clientQ.getDynData();
    }

  public int gotoTime(int time) {
    hasNext  = server.requestNewTime(time, OTFServerRemote.
        TimePreference.EARLIER);
    clientQ.getDynData();
    return server.getLocalTime();
  }
  ...
}
```

- *transims*: `TransimsSnapshotWriter`
- *googleearth*: `KmlSnapshotWriter`
- *netvis*: `NetStateWriter`
- *otfvis*: `OTFQuadFileHandler.Writer`

An additional information given to these snapshot writers is the time period between two snapshots to designate the granularity of time in which a snapshot is written. While writing snapshots every second might be helpful for taking a closer look at a small time period, for runs over an entire day time steps of 5 minutes are more appropriate. This helps keeping the visualizer files small and manageable.

The OTFVis movie file provided by the `OTFQuadFileHandler.Writer` class can be read directly by the `OTFQuadFileHander` server class.

Event file conversion

For posterior visualization of a simulation run on the basis of a given event file, no server was implemented. A tool for converting an event file to any snapshot format has already been present in the MATSim framework. Therefore, any event file can conveniently be converted to a movie file. To provide a server for event files, the structure of the class `Events2Snapshot` containing the conversion routines would have to be completely refactored to fit the time-step-based approach of the server. Instead of rewriting the class, another class `OTFEvent2MVI` was implemented for convenient conversion of event files to *.mvi* files.

Server for the TRANSIMS format

The data format known from the TRANSIMS project [99] was a long time acquaintance of the MATSim project. Many older runs exist in the form of the TRANSIMS output format *T.veh*. The option to view older runs was provided by implementing a server for this format, the `OTFTVehServer`. Furthermore, a pendant to the `OTFEvent2MVI` class was provided for converting the *T.veh* file into a *mvi* file. This class is called `OTFTVeh2MVI`.

On-The-Fly-Server for interactive simulation runs

The `OnTheFlyServer` class is a server implementation, which is run in parallel to a simulation run of the queue simulation. The `QueueSimulation` class has been extended to include an instance of the `OnTheFlyServer` class. The new class `OnTheFlyQueueSimQuad` adds little change to the original `QueueSimulation` class. Three methods of `QueueSimulation` have been overloaded. The changes are outlined in listing 5.3. Apart from initializing

and deconstructing the server in the respective methods, the server intervenes only with one line of code in the method `afterSimStep()`. There the server is given an opportunity to update its internal status. This update includes blocking the actual thread of execution, when the client sends a command to hold the simulation. This control over the simulation is supplied by another interface. The `OTFLiveServerRemote` interface extends the `OTFServerRemote` interface used before and adds the functionality necessary to interact with the running system. Apart from the option to pause or run the simulation, an option to obtain information from the *living* system is given. An outline of the interface is given in Fig. 5.8. The method `answerQuery()` will enable the client to send any kind of query into the simulation and receive an answer to display.

The `OnTheFlyQueueSimQuad` can, therefore, be stopped in execution at each time step and the inner state of any agent can be queried and inspected at will. Interactions between the agents can be visualized and interrogated. Even changes to the agent's inner state can be made, as well as changing the network's state. This is provided by another interface. The notion of a *query* is discussed in depth in the next section.

Listing 5.3: Class OnTheFlyQueueSimQuad holds an instance of OnTheFlyServer

```
public class OnTheFlyQueueSimQuad extends QueueSimulation{
  private OnTheFlyServer myOTFServer = null;

  protected void prepareSim() {
    this.myOTFServer = OnTheFlyServer.createInstance("<Server␣
        name␣or␣UUID>", this.network, this.plans, events, false
        );

    super.prepareSim();
  }

  protected void cleanupSim() {
    this.myOTFServer.cleanup();

    super.cleanupSim();
  }

  protected void afterSimStep(final double time) {
    super.afterSimStep(time);

    this.myOTFServer.updateStatus(time);
  }
 ...
}
```

class	function
interface OTFQuery	interface for OpenGL related drawing
QueryAgentActivityStatus	find the activity the given agent is doing
QueryAgentEvents	draw events related to this agent onto the road
QueryAgentId	given a coordinate, find fitting agent
QueryAgentPan	draws the plan of given agent
QueryAgentPTBus	a variation of above
QueryLinkId	given a coordinate, find fitting link
QuerySpinne	draws a partial network based on certain options
QuerySpinneNOW	a variation of above

Table 5.2: List of queries implemented for the visualizer

Querying the OnTheFlyServer

The `OTFQuery` interface given to and returned by the `answerQuery()` method of the `OTFLiveRemoteServer` interface is outlined in Fig. 5.9.

The interface's main method `query()` is called by the `OnTheFlyServer` instance. This method is provided with all relevant data concerning the actual simulation run. It is free to extract any data from the given objects and collect this data in any –serializable– format of choice. After collecting all data, the query object is returned to the client.

Queries can be of two different types. A continuous query will return `true` in the method `isAlive()` indicating that the query must be triggered at every tick until it is removed by calling the method `remove()`. In case the query is a one time query, `isAlive()` returns `false` and the query is executed only once on the server. The client side will then integrate the query into its drawing scheme, calling `draw()` everytime the display needs a refresh. A call to the method `draw()` is supplied with the actual `OTFDrawer`, which requests redrawing, as a parameter. This makes it easy to implement different drawing routines for different implementations of the client's GUI. A *SWING*-based GUI can be updated with the appropriate *SWING*-commands and an *OpenGL*-based GUI will receive GL-based drawing instructions. It is up to the `OTFQuery` class what drawer classes it supports. Only a small number of queries have been implemented to test the framework. List 5.2 sums them up.

5.4.3 Managing the data flow

As we have already seen in the previous sections, flexibility in the use of the visualizer was one important design goal. The visualizer is merely an empty framework. Classes must be supported for data generation, transportation and finally visualization. Each of these stages is designed to be overloaded

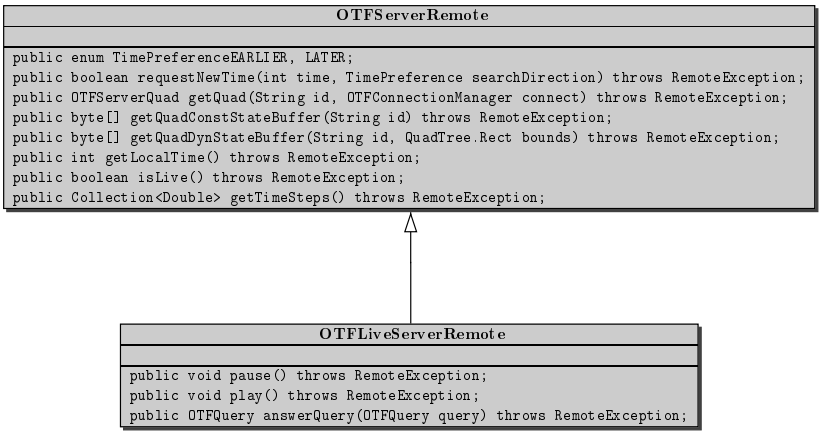


Figure 5.8: UML diagram of the OTFLiveServerRemote interface

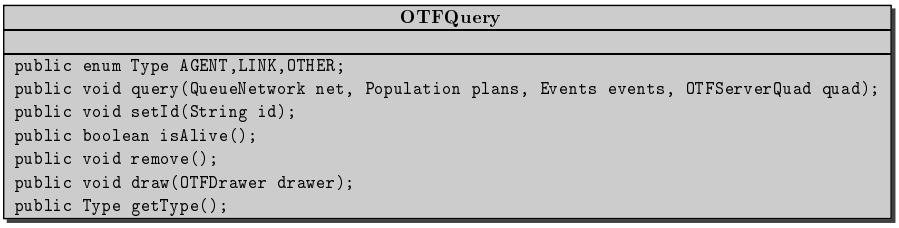


Figure 5.9: UML diagram of the OTFQuery interface

and extended with small owner-written classes, to provide additional functionality. These classes need to interoperate to build a continuous connection from the sink –usually some data source inside the queue simulation– to the display of data on the screen. A description of the connectivity of all different classes for the different stages must be found. The information about the interplay of the classes is joined in one class, the `ConnectionManager` class. An instance of the `ConnectionManager` class describes the road the data takes from data generation to visualization. The connection manager mainly consists of a map of key-value pairs. Each pair describes a distinct interplay between an intermediate data source and the corresponding sink. The data flow is normally defined by four stages of interaction. An example description of the data flow from a link with agents to the actual visualization thereof is found in listing 5.4.

Listing 5.4: A sample OTFConnectionManager instance

```
public void prepareView1() {
    private OTFConnectionManager connect = new
        OTFConnectionManager();

    connect.add(QueueLink.class, OTFLinkAgentsHandler.Writer.
        class);
    connect.add(OTFLinkAgentsHandler.Writer.class,
        OTFLinkAgentsHandler.class);
    connect.add(OTFLinkAgentsHandler.class,
        SimpleStaticNetLayer.SimpleQuadDrawer.class);
    connect.add(OTFLinkAgentsHandler.class,  AgentPointDrawer.
        class);

    connect.add(SimpleStaticNetLayer.SimpleQuadDrawer.class,
        SimpleStaticNetLayer.class);
    connect.add(AgentPointDrawer.class, OGLAgentPointLayer.
        class);
    ...
  }
```

The first entry marks the transition from the actual MATSim framework class `QueueLink` to a class capable of extracting data from a `QueueLink` and –if necessary– preparing this data for sending it over the network stream.

This class is mapped to the appropriate reader class which waits for the data at the end of the stream. This reader class is then connected to one or more classes responsible for the actual drawing of elements on screen. In the example the reader class is the `OTFLinkAgentsHandler` class. It is connected to two drawer classes, `SimpleQuadDrawer` and `AgentPointDrawer`, being separately responsible for drawing the link itself and the agents thereon. There is a last and optional stage to establish a connection between a drawing class and a layer class which bundles drawers and might provide ways to optimize the visualization of the drawers.

Each of these stages represents only small sections of the overall data flow. Most of the classes are small and easy to understand and lastly to extend.

The connection manager holds this map of relations and provides enough information to do the actual process of building the functional net of objects automatically. It also provides means to check if the given configuration is sound. In the next section a closer look is taken on how the visualization is build up using the connection manager.

Building a visualization view

First the connection between client and server is established, which is the duty of a single object of the class `OTFHostControlBar`. This class also provides the main GUI for the interaction with the server, i.e. buttons for playing and pausing the simulation and for displaying the simulation's current time. While there might be several views into the simulation data active at the same time, only one object of HostControlBar can have control over the simulation. From the client's view all interaction with the server must be initiated by contacting the appropriate host controlbar. To establish a new view onto the simulation, a connection manager object must be constructed to provide the semantic of the connection. With this at hand and an unique identifier string for the new view, the method `OTFHostControlBar.createNewView()` can be called. This returns a quadtree structure holding all reader objects to correspond to an quadtree on the server-side filled with writers.

The `OTFHostControlBar` instance calls the method `createView()` of the `RemoteServer` interface, to receive the server's quadtree. The quadtree is constructed on the server-side by stepping through all links and nodes. For each object, the `ConnectionManager` instance is interrogated to provide the actual `WriterFactory` class. This `WriterFactory` class is expected to provide an implementation of the `OTFDataWriterFactory` interface. From this factory actual data writers are requested. It is up to the factory to either create new writer instances or use a singleton writer for writing. Each `OTFDataWriter` needs to store the source's reference, which was provided during the initial building process by a call to the writer's `setSrc()` method. Later the writer is responsible for providing the necessary updates, the `QueueNetwork` will not be iterated again. When all writers have been created and sorted into the `ServerQuad` quadtree implementation, this quadtree is kept in the server, together with the *ID* of the corresponding view. The inner working of the method `createNewView()` is summed up in listing 5.5.

Listing 5.5: Creating a new View onto the simulation

```
public OTFClientQuad createNewView(String id,
    OTFConnectionManager connect) throws RemoteException {
```

```
    OTFVisConfig config = (OTFVisConfig)Gbl.getConfig().
        getModule(OTFVisConfig.GROUP_NAME);

    if(config.getFileVersion() < OTFQuadFileHandler.VERSION
        ...) {
      // go through every reader class and look for the
          appropriate Reader Version for this fileformat
      connect.adoptFileFormat(OTFDataReader.getVersionString(
          config.getFileVersion(), config.getFileMinorVersion()
          ));
    }

    System.out.println("Getting␣Quad");
    OTFServerQuad servQ = host.getQuad(id, connect);

    System.out.println("Converting␣Quad");
    OTFClientQuad clientQ = servQ.convertToClient(id, host,
        connect);

    System.out.println("Creating␣receivers");
    clientQ.createReceiver(connect);

    readConstData(clientQ);

    this.quads.put(id, clientQ);
    return clientQ;
  }
```

A copy of this quadtree is sent to the client's `OTFHostControlBar` instance using the RMI interface as discussed in Section 5.4.3. This class constructs an exact copy of the server's quadtree exchanging all writer objects by the analogous reader classes, again relying on the knowledge of the connection manager and Java's capability to create instances from class objects. This having been done, a data connection between the server and the client with the necessary data for this particular view is established. This connection is on the client-side represented by the `OTFClientQuadTree` object. Any update of the view's data will take place with the aid of this object.

The last necessary step is to actually generate the classes for the visualization of the data. Again the connection manager provides the necessary informations. The client quadtree's method `createReceiver()` recursively tracks through the tree structure and for every reader found it generates a list of corresponding drawer classes. For this to take place the drawer implementations implement one or more interfaces inherited from the `OTFData.Receiver` interface. For each receiver class a call to the reader's `connect()` method is made. The receiver will instantiate one or more objects of the `Receiver` class if it fits its needs. The receiver classes normally provide

a set of *setter*-methods to provide the necessary information for drawing. As an example the `OTFDataQuad.Receiver` interface provides `setQuad()` and `setColor()` methods for setting coordinates and color of one rectangular area. These receivers are the classes responsible for the actual drawing of information on the screen. This procedure is the topic of the next section.

The data flow from server to screen

After having established all connections, the sending of data can start. Data is classified into two different types: *Constant* and *dynamic* data. For keeping the network traffic minimal, constant data should be sent only once. Typical constant data are, for example, the coordinates of nodes or the link and node identifiers. Other data has to be updated at every tick like the agent's positions.

The constant fraction of data is sent before the first dynamic data is transmitted and right after the connection is established. The code for this is exemplified in listing 5.5. The server collects all data in a simple byte buffer. This buffer is transferred to the client side by RMI, and the client's quadtree reads the data from the byte buffer. This should make it relatively easy to transfer the mechanism to another network platform in case RMI should not be the preferred interface anymore.

The server-side writer objects are responsible for extracting the information needed from their respective source objects and putting this data onto the byte buffer. The client's reader classes do not store any data, but are solely responsible for reading the data from the byte stream in an appropriate manner and pushing them directly into the *setter*-methods of their drawer or receiver classes. This is done to keep the memory print of the data on the client-side minimal. The drawer classes could again convert or strip the data to the smallest necessary set of data already using OpenGL data structures like *DisplayLists* or *VertexObjectBuffers*, possibly relocating the data onto the graphics card, if feasible.

A sample *Reader* class is illustrated in listing 5.10. The `readDynData()` and `readConstDta()` methods do not store any data locally, but directly push them into the receiver classes. The receiver objects have to implement one or more interfaces inherited from the `OTFData.Receiver` interface.

A sample *Receiver* interface for receiving an rectangle for drawing is the `OTFDataQuad.Receiver` interface from listing 5.6. As soon as the reader class has been equipped with an object implementing this interface it can directly call a method of choice like a virtual method call, maintaining a fast and safe execution path at runtime. With this approach one issue still remains when it comes to sending different data with the same primitive type. An example receiver interface named `OTFDataFloat.Receiver` defining the anticipated `getFloat(float value)` method is not a good idea. Given a reader wants to submit two different float values with distinct meaning, e.g.

a flow capacitiy and a space capacity, with the `OTFDataFloat.Receiver` interface, there is no way to reconstruct which float value belongs to which receiver class. This could be avoided by creating two interface classes, e.g. `OTFDataSpeedCap.Receiver` and `OTFDataFlowCap.Receiver` and naming the methods appropriately. Using these `Receiver` interfaces will make each access unique again. The `Receiver` interfaces need to be named after their data's usage, not after the actual type of data transferred.

Listing 5.6: A receiver interface for quad data

```
public interface OTFDataQuad extends OTFData{
  public static interface Receiver extends OTFData.Receiver{
    public void setQuad(float startX, float startY, float endX,
        float endY);
    public void setQuad(float startX, float startY, float endX,
        float endY, int nrLanes);
    public void setColor(float coloridx);
    public void setId(char[] idBuffer);
  }

}
```

Java RMI

As the design of the visualizer was defined as a client-server-architecture, some sort of protocol to exchange data from a remote computer to the client and vice versa had to be found. As the programming framework was the Java SDK, Java's own RMI (Remote Method Invocation) framework for exchanging data seemed natural. Java's original purpose was to be the *lingua franca* of the world wide web. Even though this goal has never really been met, many parts of the Java SDK are still dedicated to web services and communication. The RMI framework is one of these packages. It provides all necessary procedures and structures to send "living" objects over the net from one Java VM (virtual machine) to another as long as some kind of network connection is established.

The RMI interface has been developed together with the Java language itself and is, therefore, tightly integrated into the Java framework. It obeys all security constraints of Java, making it easy to guarantee a secure and encrypted connection if necessary and offering a fast and simple implementation when security is not an issue, like in an internal network. Using RMI makes the network connection nearly transparent to the user. Accessing a remote object is not much different from accessing a local objects, apart from catching the `RemoteException` if necessary.

The RMI framework is using Java's `Serializable` interface to send, marshal and un-marshal data on different computers. To enable an object to be send over the Internet it must be serializable. To call a remote object over

the network, this object must implement the `Remote` interface. Furthermore, every method which might be called over RMI must declare to throw a `RemoteException`.

Establishing a remote connection

All access to a remote object is managed by a registry. The Registry has to be accessed via either a call to the static method `getRegistry()`, or by creating a new registry with `createRegistry()`. For the OTFVis a registry is created at a certain port number (4019). The first instance of a `RemoteServer` will create a registry at this port, further instances will use the existing registry. There is no problem with putting this registry anywhere else, fitting the needs of the network topology. Running a beowulf-cluster, typically connected to the Internet by a single host computer, one might place the registry on this host computer so that the registry is reachable from the Internet as well. Over the so-placed registry access from the Internet to the cluster's inner computer's simulations is possible.

To make an object known to the registry, it will be bound to a certain registry by calling the registry's method `bind()`. A unique name must be given for every object. For the OTFVis server this is achieved by either using a user-defined unique name or by creating an UUID with the java.util.UUID class provided by the 1.5 Java SDK. This UUID is appended to the String `"OTFVisServer_"` to make it easier to identify the object as an `OTFServer`. After the binding, the object is ready to be accessed remotely.

The client will have to ask the registry for access to the object. To do so, the client needs to know the UUID of the server. This could be achieved by letting the client start the server itself with a client-chosen UUID, or by telling the client the UUID of the server to attach to. The client's constructor takes a sort of URL to connect to different servers. The URL `"rmi:localhost:4019:OTFServer_<UUID>"` will for example open a local connection to a server with the given UUID. Another example might be `"file:../studies/berlin/500.plan.mvi"` which will open a QuadFileServer instance and read from a movie file. For "live" connections, either `"rmi:"` or `"ssh:"` type of URL could be used, establishing an unsecured or secured connection to the OTFVis server.

After connecting to the RMI registry, the client can receive a list of remote objects via a call to the registry method `list()`. From this list it has to pick the appropriate server by the UUID. It will then request a proxy of the found object by a call to `lookup()`. From then on the client can use the server as if it is a local object, except for the `RemoteExceptions`. In case of the `OTFServer`, the client can rely on the `LiveRemoteServer` interface. It is inherited from the `RemoteServer` interface and offers the additional functionality of sending queries to the server.

OTFVis and RMI

After a client has connected to a host, that is, after retrieving the server's proxy object given as an `OTFRemoteServer` interface, the client can ask for this server's quadtree representation of the network. This quadtree representation is then converted from a `OTFServerQuad` (containing all `OTFDataWriter` objects) to an exact copy containing corresponding `OTFReader` objects. Sending objects over the RMI interface is rather expensive. Therefore the main communication between server and client after exchanging the quadtree structure is reduced to the exchange of arrays of bytes. All writer on the server side will write their data into the byte buffer and the readers on the client side will read the data from the buffer into the actual objects responsible for drawing. This ensures the smallest possible amount of data being send over the network. How this is achieved is described in Section 5.4.3.

Listing 5.7: Interface for the remote server classes

```
public interface OTFServerRemote extends Remote {
  public enum TimePreference{EARLIER, LATER};
  public boolean requestNewTime(int time, TimePreference
      searchDirection) throws RemoteException;

  public OTFServerQuad getQuad(String id, OTFConnectionManager
      connect) throws RemoteException;
  public byte[] getQuadConstStateBuffer(String id) throws
      RemoteException;
  public byte[] getQuadDynStateBuffer(String id, QuadTree.Rect
      bounds) throws RemoteException;

  public int getLocalTime() throws RemoteException;
  public boolean isLive() throws RemoteException;

  public Collection<Double> getTimeSteps() throws
      RemoteException;
}
```

The full `OTFServerRemote` interface is illustrated in listing 5.7. Apart from the methods discussed in Section 5.4.2 it mainly consists of the methods necessary to send the quadtree and the byte buffers from server to client. The most frequently called method of this interface is `getQuadDynStateBuffer()`, which returns the aforementioned buffer of byte data representing the actual state of the data in the rectangle given by the `bounds` parameter. The `id` string parameter is needed as one server can serve multiple views onto the actual simulation. This is discussed in the Section 5.2.1

5.4.4 Extensibility

The visualizer can be extended in many ways. Starting with smaller issues like replacing the agent's drawer by some other way to draw. Assume one wants to change the coloring scheme for the agents' icons to run from blue to yellow instead of the standard coloring. All there is to be done is to write a small new drawer for the agents, as given in listing 5.8.

Listing 5.8: New coloring scheme drawer for agents

```
  private static OTFOGLDrawer.FastColorizer colorizerBlue = new
       OTFOGLDrawer.FastColorizer(
      new double[] { 0.0, 50.}, new Color[] { Color.BLUE, Color
          .YELLOW});

  public class AgentPointDrawerBlue extends AgentPointDrawer {
    public void setAgent(char[] id, float startX, float startY,
         int state, int user, float color) {
        drawer.addAgent(id, startX, startY, colorizerBlue.
            getColor(0.1 + 0.9*color), true);
    }
  }
}
```

Then replace the appropriate entry in the `OTFConnectionManager` by one referring to the new class and finally associate the drawer with the correct layer by adding another entry to the connection manager. The two lines will look like this:

```
connect.add(OTFLinkLanesAgentsNoParkingHandler.class,
    AgentPointDrawerBlue.class);
connect.add(AgentPointDrawer.class, OGLAgentPointLayer.class);
```

For more sophisticated operations one might want to add a customized layer. For example, to assure a certain drawing order, one might introduce an additional layer, overloading the layers `getDrawOrder()` method as described in Section 5.5.2. If one wants to add more or different data from the server to the visualization process, there are two alternative approaches. Either a new query is defined and collects and displays the data, or a new writer/reader pair is written. This, again, needs to be inserted into the `OTFConnectionManager`s mapping. New ways of interaction can be provided by overloading the `VisGUIMouseHandler` class handling all mouse movements. Finally, even the layout of the views can easily be adapted to new needs. The `OnTheFlyClientQuad` is defined as a `BorderLayout` with a `JSplitPane` split layout window in the center, capable of drawing two different views of a simulation run. An example for a split view visualization is depicted in Figure 5.10.

This could easily be replaced by any combination of Java's layout managers and panes. The two indispensable elements of the visualization, the

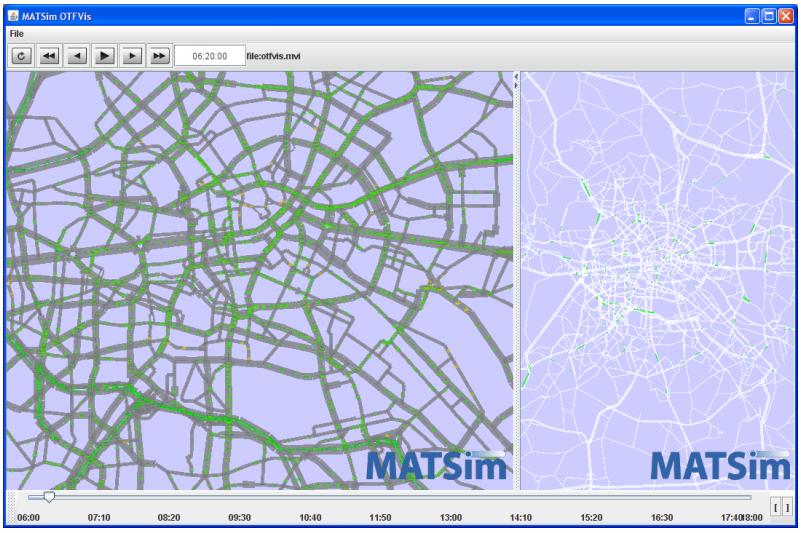


Figure 5.10: A visualization using the split view pane.

`OTFControlToolBar` and the view represented by an `OTFOGLDrawer` instance, are just plain Java `Components` to be inserted into any SWING GUI.

The visualizer is easily extended into any direction and the actual state of affairs should be understood as a starting point for new and more sophisticated ways of mining into the queue simulation's data. The next chapter will introduce an extension to the visualizer, enabling the visualizer to show results from the CUDA runs by accessing the graphics hardware memory directly without resorting to the CPUs memory, thereby, saving time and memory.

5.4.5 Version management

Having an open system like the one implemented with the *OTFVis* also implies that one has to be prepared for change. A change of a reader or writer class might be necessary to add features or remove unnecessary data. As we write the binary stream of information that travels through RMI into a plain byte buffer and dump this to a file, this file format is fragile in terms of breaking with classes changing. Three levels of control have been implemented into the OTFVis to avoid breaking backwards compatibility, each addressing another aspect of possible change. The next three sections will explain each level and what safety it achieves for the file format. Only the *.mvi* files are affected by changes in the class structure. If you change a class structure and then run the "live" version of the OTFVis all changes will be in place anyway.

Moving classes

Unfortunately, the classes saved to file via the Java's `Serializable()` interface are being saved with their full path. For example, `org.matsim.utils.vis.otfvis.handler.DefaultAgentHandler.Writer` might be such a class. At some later point in development, the entire package might need to be refactored and to be moved one level up out of the `utils` package. No class or functionality has been changed, however, the class path has. So when reading our old visualization from file, Java is unable to find and construct the original class.

Fortunately, the Java creators have foreseen this situation and exposed the mechanism in charge of resolving the classes to the user. Whenever a stream is used to read such a serializable object from a file, a special stream is used. This stream is derived from the `DataObjectInputStream` normally used for reading the objects. This new `OTFDataObjectInputStream` overloads the method `resolveObject()` which is in charge of creating and presenting a class ready to be filled with the data read from the file. Thus, any change in the class-path can be kept track and taken care of in this method. Listing 5.9 shows such a method for the example above and, additionally, exchanges a very old implementation of the quadtree with the current one.

Listing 5.9: Resolver for Java classes

```
      protected Class resolveClass(final ObjectStreamClass desc
          )  throws IOException, ClassNotFoundException {
        String name = desc.getName();
        System.out.println("try␣to␣resolve␣" + name);
        if (name.equals("playground.david.vis.data.
            OTFServerQuad")) {
          return OTFServerQuad.class;
        } else if (name.startsWith("org.matsim.utils.vis.otfivs
            ")) {
          name = name.replaceFirst("org.matsim.utils.vis.otfivs
              ",
              "org.matsim.vis.otfvis");
          return Class.forName(name);
        }
        return super.resolveClass(desc);
      }
    }
}
```

So this mechanism will ensure a high level of security when classes are moved around without being changed.

Adding or removing data

Another situation that might appear is that we want to add data to our class or remove data from it. One viable idea is certainly to just put the additional data into an inherited class of the original or into a completely separate class altogether. If that is, for some reason, not feasible, then the changing of the class can be performed in a broad manner without breaking compatibility. The Java documentation [57] offers the following options for changing a class-definition without breaking compatibility:

> The compatible changes to a class are handled as follows:
>
> - Adding fields - When the class being reconstituted has a field that does not occur in the stream, that field in the object will be initialized to the default value for its type. If class-specific initialization is needed, the class may provide a `readObject` method that can initialize the field to nondefault values.
> - Adding classes - The stream will contain the type hierarchy of each object in the stream. Comparing this hierarchy in the stream with the current class can detect additional classes. Since there is no information in the stream from which to initialize the object, the class's fields will be initialized to the default values.
> - Removing classes - Comparing the class hierarchy in the stream with that of the current class can detect that a class has been deleted. In this case, the fields and objects corresponding to that class are read from the stream. Primitive fields are discarded, but the objects referenced by the deleted class are created, since they may be referred to later in the stream. They will be garbage-collected when the stream is garbage-collected or reset.
> - Adding `writeObject/readObject` methods - If the version reading the stream has these methods then `readObject` is expected, as usual, to read the required data written to the stream by the default serialization. It should call the `defaultReadObject` first before reading any optional data. The `writeObject` method is expected as usual to call `default\-WriteObject` to write the required data and then may write optional data.
> - Removing `writeObject/readObject` methods - If the class reading the stream does not have these methods, the required data will be read by default serialization, and the optional data will be discarded.

- Adding `java.io.Serializable` - This is equivalent to adding types. There will be no values in the stream for this class so its fields will be initialized to default values. The support for subclassing nonserializable classes requires that the class's supertype have a no-arg constructor and the class itself will be initialized to default values. If the no-arg constructor is not available, the `InvalidClassException` is thrown.
- Changing the access to a field - The access modifiers `public`, `package`, `protected`, and `private` have no effect on the ability of serialization to assign values to the fields.
- Changing a field from static to *nonstatic* or *transient* to *nontransient* - When relying on default serialization to compute the *serializable* fields, this change is equivalent to adding a field to the class. The new field will be written to the stream but earlier classes will ignore the value since serialization will not assign values to static or transient fields.

The documentation also indicates changes that will actually break the backwards compatibility:

> Incompatible changes to classes are those changes for which the guarantee of interoperability cannot be maintained. The incompatible changes that may occur while evolving a class are:
> Deleting fields - If a field is deleted in a class, the stream written will not contain its value. When the stream is read by an earlier class, the value of the field will be set to the default value because no value is available in the stream. However, this default value may adversely impair the ability of the earlier version to fulfill its contract.

- Moving classes up or down the hierarchy - This cannot be allowed since the data in the stream appears in the wrong sequence.
- Changing a *nonstatic* field to *static* or a *nontransient* field to *transient* - When relying on default serialization, this change is equivalent to deleting a field from the class. This version of the class will not write that data to the stream, so it will not be available to be read by earlier versions of the class. As when deleting a field, the field of the earlier version will be initialized to the default value, which can cause the class to fail in unexpected ways.

- Changing the declared type of a *primitive* field - Each version of the class writes the data with its declared type. Earlier versions of the class attempting to read the field will fail because the type of the data in the stream does not match the type of the field.
- Changing the `writeObject` or `readObject` method so that it no longer writes or reads the default field data or changing it so that it attempts to write it or read it when the previous version did not. The default field data must consistently either appear or not appear in the stream.
- Changing a class from `Serializable` to `Externalizable` or vice versa is an incompatible change since the stream will contain data that is incompatible with the implementation of the available class.
- Changing a class from a *non-enum* type to an *enum* type or vice versa since the stream will contain data that is incompatible with the implementation of the available class.
- Removing either `Serializable` or `Externalizable` is an incompatible change since when written it will no longer supply the fields needed by older versions of the class.
- Adding the `writeReplace` or `readResolve` method to a class is incompatible if the behavior would produce an object that is incompatible with any older version of the class.

These rules seem rather difficult to cope with, but most of them describe a rather natural usage of the classes. All simple changes to classes will be mostly harmless.

Version management of the Buffer I/O

As safety measures mentioned above only affect the classes read from file via `OTFDataObjectStream`, but not the plain data written into the byte buffer, a third and last frontier is build into the OTFVis file format to handle changes. Each *.mvi* file gets tagged with a version number, the moment it gets saved to disk. This version number of the format *MajorVersion.MinorVersion* can be used as an indicator of the particular reader class to use for reading objects from the binary stream. The *MajorVersion* number is thought of as indicating an actual break of backwards compatibility, i.e. when the major number from the file is smaller than the actual major number stored in the `OTFVisConfig`, the file is no longer loadable. This is implemented to give the MATSim coordinators the possibility to –at some time in the future– clean out compatibility code that is no longer needed in a safe and documented manner. If only the minor number changes, a non-breaking

change will be indicated. If, for example, additional data is to be written by the `OTFLinkAgentsHandler` and the actual file version is *1.1*, the minor version will thus be increased setting the version number to *1.2*. The old handler's functionality has to be preserved in another class, e.g., an inner class of the `OTFLinkAgentsHandler`.

This might lead to the following class description:

```
  /***
   * PREVIOUS VERSION of the reader
   */
  public static final class ReaderV1_1 extends
      OTFLinkAgentsHandler {
    @Override
    public void readDynData(ByteBuffer in, SceneGraph graph)
        throws IOException {
      agents.clear();

      int count = in.getInt();
      for(int i= 0; i< count; i++) readAgent(in, graph);
    }
  }
}
```

This functionality and the version number up to which the legacy code is responsible for reading data written for the `OTFLinkAgentsHandler` need to be communicated to finalize the change. This is done by including a static block in the class declaring the necessary information:

```
public class OTFLinkAgentsHandler extends OTFDefaultLinkHandler
    {

  static {
    OTFDataReader.setPreviousVersion(OTFLinkAgentsHandler.class
        .getCanonicalName() + "V1.1", ReaderV1_1.class);
  }

...}
}
```

This information is visible to the `ConnectionManager`. Before the connection manager assembles the necessary classes for reading data from a file, it will fetch the version number from the file and replace the `Reader` class with a previous version if necessary. Therefore, it is possible to change the reader classes for the plain byte stream without breaking the backwards compatibility to older files. Another viable method would be to just inherit from the previous version and overload the writing that needs updating. But this might not be a suitable idea in all cases, as one might want to change the functionality of a specific reader/writer-pair that is, from there on, obsolete in his older version. In this case the versioning is a good tool

for avoiding cluttering the namespace with several similarly named classes.

5.5 Optimizations

In this section two strategies to optimize the data transmission and the display of data on screen will be presented. Both have been largely used in the implementation of 3D graphics applications but prove valuable in the context of large-scale 2D visualization as well. *Quadtrees* (or in the 3D world octrees) help to spatially structure data to make it simple to access certain areas only. So, whenever the display shows only a portion of the overall network, the speed of the application will benefit from the use of quadtrees. Another optimization strategy is to re-use already rendered sections of the data. This is done by carefully organizing these Sections in *SceneGraphs*. Whenever the screen needs a redraw (without new data from the server) the SceneGraph can be used to speed up the display. The next two sections will discuss the implementation of these optimitzations.

5.5.1 Data exchange via quadtrees

The main attention, when constructing the client server architecture, was payed to carefully reducing the amount of sent data to a minimum. Sending redundant data apparently collides with the goal to implement the fastest possible way to visualize great amount of data. To avoid sending redundant data, the server must know, what information is to be drawn on the actual clients display. Therefore, it is mandatory to know the exact rectangular sector visible on screen to exclusively evaluate and send visible data to the client. Fortunately, this is what the quadtree structure is capable of with a very small administrative overhead. A spatial query is sent to the server's quadtree to let all writer objects of a certain region write their data into a byte buffer. This buffer is then sent over to the client via RMI and the client asks its quadtree to read exactly the same rectangular area from the byte buffer. So only the minimum amount of data is written and read. Fig. 5.11 illustrates this procedure.

So each writer class must be complemented by the appropriate reader class. For structural reasons this is done by uniting both classes in a *Handler* class, proving the reading capabilities itself and having an inner class that implements the writer class. By keeping the code of reader and writer in one class and, thus, source-code file it is easier to maintain the synchronicity of both reading and writing, which apparently is mandatory, as the stream itself has no information whatsoever of when the data dedicated to one reader ends and the next reader has its turn. A rather simple class is presented by listing 5.10. This writer simply writes one float value per link. Likewise the reader class picks one float from the stream. This has to be done regardless if there is a visualizer waiting for the data or not. Otherwise the stream

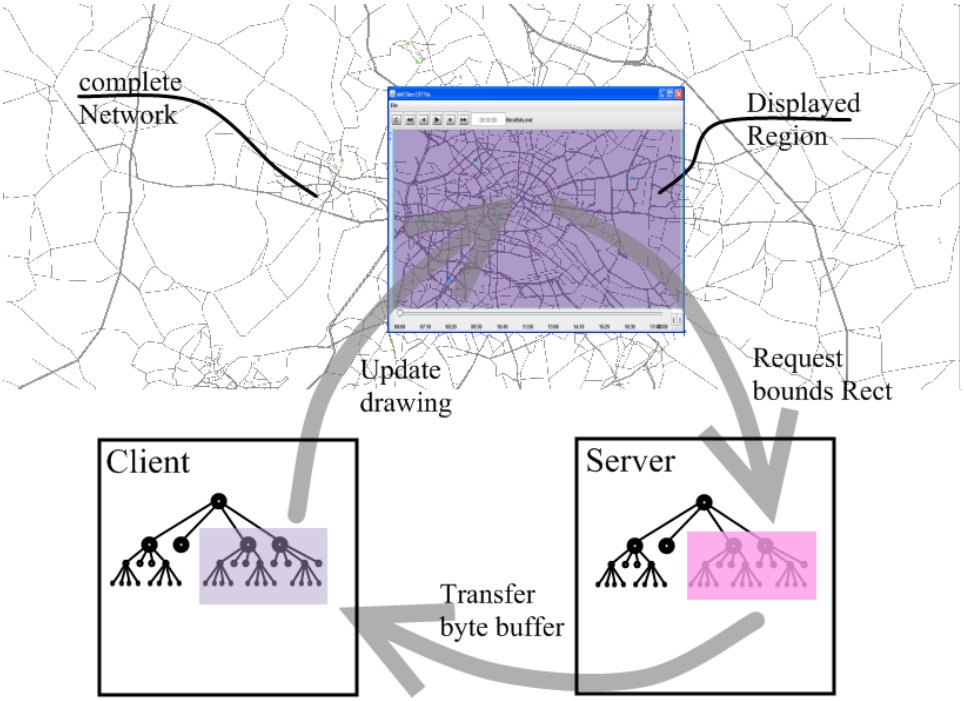


Figure 5.11: The update process per quadtree

would be out of synchronicity. Just like the `ServerRemote` interface each writer class provides a method for sending the constant data bits as well as the dynamic ones. No time step is given here, though. Choosing the right time step is up to the server. All writers will then write the same time step.

Listing 5.10: Implementation of a simple LinkHandler class

```
public class OTFDefaultLinkHandler extends OTFDataReader
    implements OTFDataQuad.Provider {

  protected OTFDataQuad.Receiver quadReceiver = null;

  static public class Writer extends  OTFDataWriter<QueueLink>
      {

    private static final long serialVersionUID =
        2827811927720044709L;

    @Override
    public void writeConstData(ByteBuffer out) throws
        IOException {
      out.putFloat((float)(this.src.getLink().getFromNode().
          getCoord().getX() ...));
      out.putFloat((float)(this.src.getLink().getFromNode().
```

```
            getCoord().getY() ...));
        out.putFloat((float)(this.src.getLink().getToNode().
            getCoord().getX() ...));
        out.putFloat((float)(this.src.getLink().getToNode().
            getCoord().getY() ...));
      }

      public void writeDynData(ByteBuffer out) throws IOException
          {
        out.putFloat((float)this.src.getVisData().
            getDisplayableTimeCapValue());
      }
    }

    public void readConstData(ByteBuffer in) throws IOException {
      float xs = in.getFloat();
      float ys = in.getFloat();
      float xe = in.getFloat();
      float ye = in.getFloat();
      this.quadReceiver.setQuad(xs, ys, xe,ye);
    }

    public void readDynData(ByteBuffer in, SceneGraph graph)
        throws IOException {
      this.quadReceiver.setColor(in.getFloat());
    }
}
```

5.5.2 Caching

Every time the user pushes a button on screen, uses zooming or panning, opens the context menu or does any other interaction with the application, an update of the screen's content is necessary. In many cases no connection or update to the server is necessary, as all data is available in the client already. Providing this data to avoid having to resort to the server is called *caching*. This technique has been used in other visualization tools as well, e.g. Treinish et al. [6]. A sophisticated caching strategy has been implemented in the visualizer ensuring that all data is transferred only once. The steps of refreshing the display and the caching used to optimize this is discussed in this section.

Refreshing the screen view

When the screen rectangle changes, which might occur frequently, i.e. whenever the user zooms in or out, resizes the window, puts the application to the back or translates the view, the screen's content needs to be redrawn.

In many cases some or all of the data already transferred stays valid. It should, therefore, not be transferred again. If the user zooms in on an area, for example, no additional data will need to be transferred, as a lager part of the screen has already been transferred. To avoid getting the information again, it needs to be cached. If the screen is moved, additional data will need to be added to the present data. Therefore, a data structure must be found that offers the capability to add new data later.

SceneGraph

Scene graphs have been used widely in computer graphics to store a compact yet flexible representation of the data necessary for visualization. The notion of a graph representation of 3D scenes goes back into the nineties when Strauss and Carey [95] illustrated with *IRIS inventor* a new approach for interactive representation of data by means of a scene graph. A little later Rohlf and Helman [84] presented the *IRIS performer*, an optimization of the *IRIS inventor*. From this *OpenInventor* evolved. A more recent example would be the *OpenSceneGraph* by Burns, et al. [22].

With the OTFVis a rather simple scene graph was used to store the actual representation of the data on screen. As mentioned in the last section, a way to intermediately store the graphics data was needed, capable of the assimilation of later additions to the graphics as the user explores more of the visualized network. Using the quadtree ensures the bits of information immediately needed are transferred from server to client. As the quadtree is already a highly sophisticated data structure the actual scene graph to store the drawer is rather simple. The data coming from the quadtree is not stored in the reader classes, but directly sent to the appropriate receiver objects. The receiver objects are responsible not only for storing the data, but also for drawing a representation of the data on screen. Keeping track of these receiver objects enables us to redraw the scene by simply calling the appropriate receiver objects again.

SceneLayer

To enable synergies between the different receivers, these can be bundled in *layers*. Layers were mentioned in the section on the `ConnectionManager` class already. Additional entries correlating drawer classes with layers were given there. These layers have a two-fold duty in the visualization process. First, they serve as a factory for drawer objects. That is to create a new drawer object, the actual layer responsible for these drawer objects is asked for an instance of the drawer class giving the layer the opportunity to decide whether to create a new instance or reuse an existing instance. This gives the layer enough flexibility to centralize certain drawing actions, e.g. use one big OpenGL `VertexArray` object to hold the data of all vehicles to be drawn.

Each drawer not assigned to a distinct layer will end up in the automatically generated default layer. The default layer will simply return a new instance of the object's class and will call the respective `draw()` routine every time the object needs to redraw itself. Therefore, execution of the drawing of all objects is secured, albeit not essentially with the optimal drawing strategy.

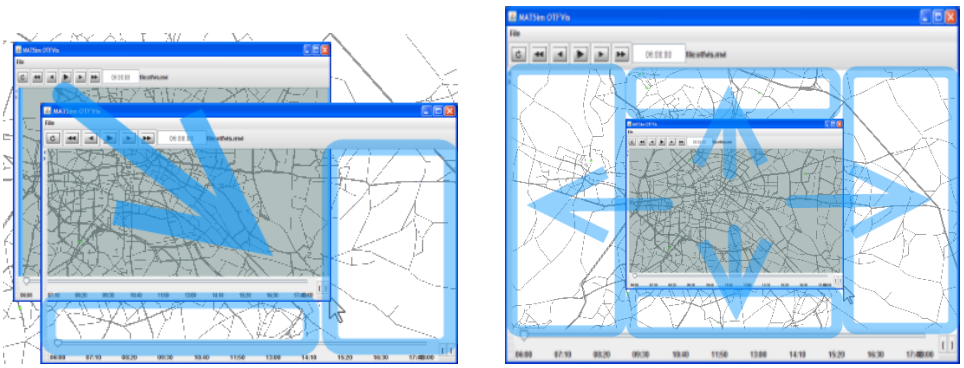
Additionally, each layer provides a `getDrawOrder()` method. This method is responsible for the sequence in which the layers will be drawn by the scene graph. The background layer is defined to have an order of zero. The layer drawing the agents is defined to have the order of *100*. The higher the order number is, the later – i.e. further up – a layer will be drawn.

The drawer implementation

One last object missing in our chain of responsibility is an `OGLDrawer` instance. This class is responsible for managing the actual OpenGL canvas and accepting redraw commands from the OpenGL framework. As mentioned before, there are two distinct reasons for redrawing the screen. One being the ongoing change in the simulation as one or more time steps pass. In this case the `OTFHostControlBar` will issue a command for redrawing to all views attached to it. The other reason for the need to redraw is that the user changes the zoom or clipping region displayed or resizes the window. The latter cases are reported directly to the OpenGL framework by the operating system. The framework will in turn call the `display()` method of the `OTFOGLDrawer` instance. There are two ways to issue a redraw in the `OGLDrawer`. The simple and less time consuming option is to call the `redraw()` method. This is only feasible when it is guaranteed that no additional data is needed. For example using the selecting or zoom rectangle on screen needs all data on screen to be redrawn every time the selection box moves. But at the same time it is guaranteed that no additional data is needed, as the screen section is not moved. Therefore, a simple redraw of the scene graph is sufficient.

If, on the other hand, the screen region changes, the already accumulated data might not be sufficient anymore. In this case the drawer's `invalidate()` method is used to indicate that the actual data stored in the scene graph might not be valid anymore.

The `invalidate()` method knows a single parameter: a *time* value. If this time value equals the time of the actual scene graph, or if -1 is given, indicating that the time has not changed, the `OGLDrawer` will re-use the existing scene graph. It will receive the updated rectangle describing the region visible on screen and pass this information on the related client quadtree to update the scene graph accordingly. The quadtree object will analyze the correlation between the existing and already loaded rectangle and the new region to display, generating up to four new rectangles of unread sections.



(a) Paninng (b) Zooming

Figure 5.12: Different unread regions resulting from user interaction.

The actual number of sections to be read depends on the action the user performs. A zooming into the current clipping does not trigger any re-loading of data at all, whereas moving the section around can trigger up to two new rectangles to read data from. Zooming out of the current section finally triggers reading four new sections at each corner of the former clipping region. This is illustrated in Fig. 5.12. The blue sections must additionally be read from the server. As the scene graph is capable of handling the extra data, the new sections are being read into the existing scene graph and this is returned to the `OGLDrawer` object. As a final operation the drawer object calls its own `redraw()` method.

The chain of responsibility

To sum up the aforementioned, the `OTFOGLDrawer` of each view is responsible for keeping the actual screen display up to date. The drawer does so by maintaining a cache object containing the elements to be drawn. Four distinct classes might ask the `OTFOGLDrawer` to update. The `OTFHostControlBar` will call the drawer's `invalidate()` method, whenever the time on the server has changed. This leads to the construction of a new `SceneGraph` object containing the actual screen rectangle's representation of the current time step. The second class is the `OTFQueryBar`, which is the toolbar on the lower end of the screen that handles user triggered queries into the simulation. The third class is the `VisGUIMouseHandler` class, which represents any change inflicted by user interaction via mouse or keyboard. The last possible change comes from the OpenGL context itself, e.g., when changing the screen size.

When the drawer is asked to redraw the screen, it is sufficient to simply call the drawing routine of the actual scene graph again. The drawer is asked

to invalidate, because either the user is doing an interaction that brings up a new region on the screen, or the host is receiving a new time step. The `OTFOGLDrawer` will ask his client quadtree to read in the new dynamic data received and then invalidate itself. This causes all drawers to get updated with the latest information.

These drawers will be collected in the new scene graph, if a new time step is requested. If only the view region is changed, the drawer objects triggered by the call to `invalidate()` will be collected in the existing scene graph, additionally to the already established drawer items. This scene graph is handed back to the `OTFOGLDrawer`.

The drawer then does a simple redraw of the new scene graph including the newly added elements. The source code in listing 5.11 illustrates the steps necessary to do an update of a region. This code is distributed over different methods in the actual implementation. A new `SceneGraph` instance is initiated and all data necessary to display is retrieved from the server side in an `ByteBuffer` object. After that, the affected region of the client quad is traversed, issuing read operations on all reader objects. As mentioned before, these read operations do not store the information in the reader object, but pass it along to the related receiver/drawer objects immediately.

The last step is to traverse the region once again with the `InvalidateExecutor`, which will add all drawer objects to the new scene graph. Having executed these steps, the new scene graph is ready to draw the updated content in an optimized way. The whole process is shown in Figure 5.13.

The server side of this operation is somewhat simpler. It basically contains one line executing a `WriteDataExecutor` on the appropriate region of the server's quadtree, which will write all data to a `ByteBuffer`.

Listing 5.11: The principal steps for updating a region in the client

```
    Rect bound = {the rectangle to be read}
    String id = {The identifier for the quadtree to read from}

    SceneGraph newScene = new SceneGraph(rect, time, connect,
        drawer);

    ByteBuffer in = ByteBuffer.wrap(host.getQuadDynStateBuffer(
        id, bound));
    clientquad.execute(bound, new ReadDataExecutor(in,
        readConst, result));

    clientquad.execute(bound, new InvalidateExecutor(newScene))
        ;

}
```

As the `VisGUIMouseHandler` can only initiate changes that are local to one particular `OTFOGLDrawer`, only this drawer is made known to it. The

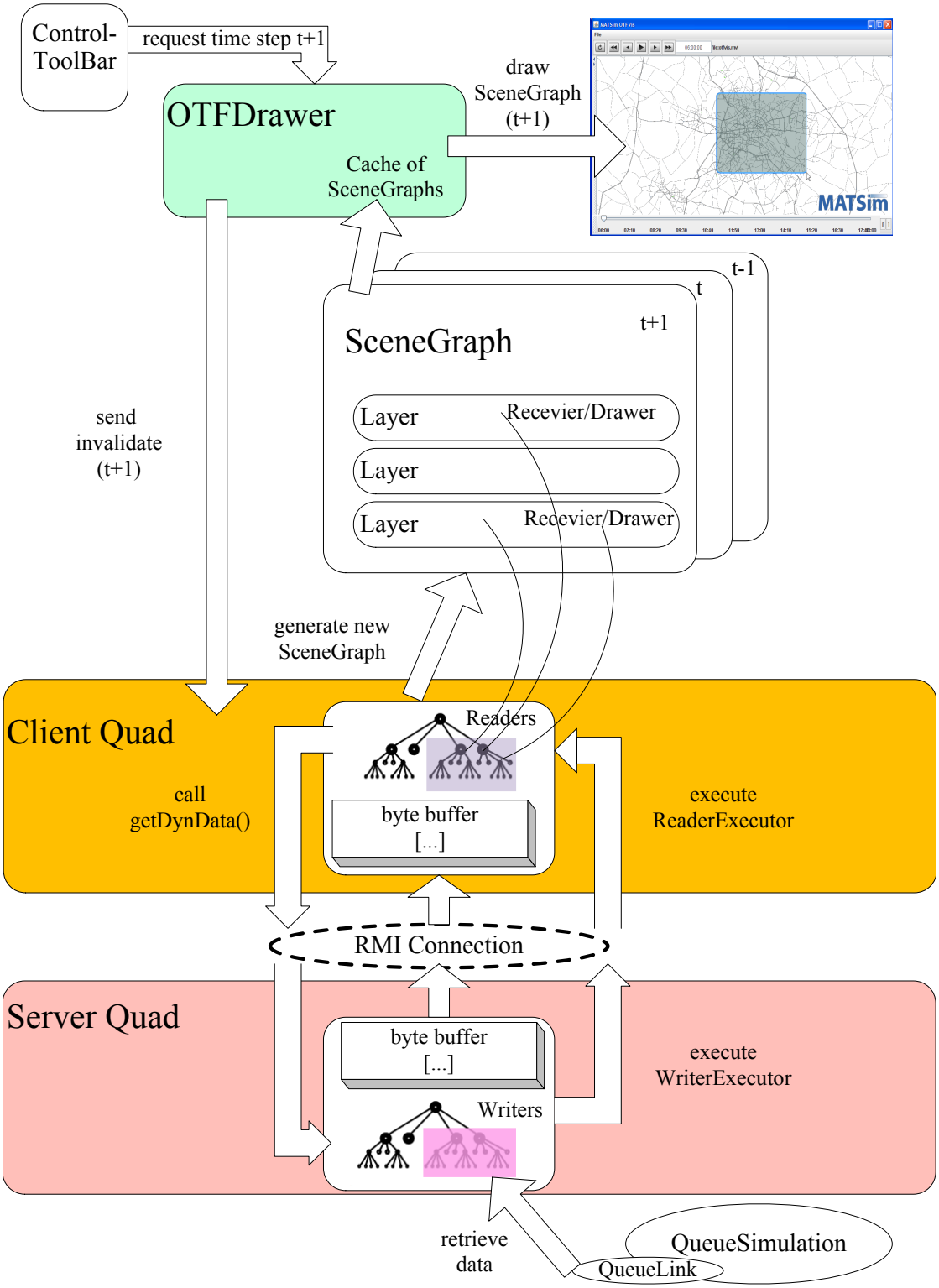


Figure 5.13: Requesting a new time step t+1.

OpenGL framework knows all drawers, as they are `GLCanvas` objects, and will ask them to redraw individually. The `OTFQueryBar` does not know any drawer objects, but can ask the `OTFControlBar` to update all views. The `OTFControlBar`, finally, knows all drawers established.

Working with the quadtree

Many of the abovementioned steps require traversing the quadtree structure for a distinct rectangular region only. The implementation of different behavior upon the data stored within a quadtree employed a *Visitor*-pattern, as defined by Gamma, et al.[42]. The *Visitor*-pattern allows additional functionality to be implemented on a certain data structure without infringing the structural integrity of this structure. The *Visitor* adding functionality to the quadtree implements the `QuadTree.Executor` interface. The interface consist of only one method:

`execute(double x, double y, T object)`.

This method implements the actual functionality. An instance of the derived `Executor` class can be executed on a certain region of the quadtree by calling the tree's `execute(Rect region, Executor<T> executor)` method.

5.6 Conclusion

The dissection of the visualization techniques used in conjunction with MATSim in the past helped to get a good overview over what should be found in a new visualizer and what should be avoided. A flexible design was chosen based on *use cases* that outlined the range of application for such a visualizer.

The two most important features to implement were the ability to run the visualizer remotely and to issue queries into a running simulation.

Running the visualizer remotely proved important as the longer iterations of the simulation normally are being run on some remote computer in a cluster and not locally on one's desktop computer or notebook. Being able to connect to a remote simulation to take a look into the actual iteration is helpful to get an impression whether it is worth to continue the iterating. The other effect of moving the visualizer to a separate computer is that all resources on the simulation computer can be used for simulation.

The ability to connect to a "living" simulation, asking questions about every aspect of the simulation's state is the other important feature. These queries enable the modeler to efficiently debug their simulation by inspecting every aspect at every second. This inspection can be done in a visual way, drawing the state into the network on screen which is often more effective than a column of numbers.

Apart from that, it is also possible to record movie files from simulation runs to e.g. show different outcomes of different policies, tolls or changes in

the road infrastructure.

Client-server architecture, quad trees and scene graphs were adapted for the usage with MATSim. The overall design proved to be useful to implement queries. If all queries can be expressed with the given implementation is still an open question. The development of MATSim is still a very agile and ongoing process, so the visualizer will have to prove if it is capable to adapt changes in MATSim fast enough. But as the actual interface between MATSim and the visualizer is rather small, the author is positive about that. The visualizer has been accommodated positively by the rest of the MATSim developers. Extensions for the use with evacuation simulation (display of tsunami wave, shelters) and public transport (display of bus stops and buses) have already been implemented.

Eventually it can be said, that the OTFVis has proved to be sufficiently fast to meet the demands of the MATSim framework and it has shown to be a stable application. It managed to stay backwards compatible to older movie files and still be open to new extensions.

Chapter 6

OTFVis and CUDA

The visualization framework and the fast CUDA-based simulation which have been discussed in the two previous chapters can be combined to construct the interactive visualization system proposed in the introduction to this thesis.

As the CUDA simulation is already embedded into the MATSim framework and the visualization is capable of running the controller of the MATSim framework, regardless of the simulation used, the "missing link" for drawing the results of a CUDA run is easily spotted:

- A reader/writer pair for transporting the generated data to the client
- A drawer class for drawing the CUDA data
- A client/server pair maintaining instances of the above classes

The rest of this chapter will discuss the changes necessary for running the visualizer and the CUDA simulation together. Three distinct modules have been changed: the simulation itself, the client and the server.

6.1 Additions to the server side

The server can be changed rather easily. As the `OnTheFlyServer` class maintains an extra list of objects not included in the quadtree, the newly created `OTFWriter` object for the CUDA vehicle data can be inserted as an additional object into the server's object list. The changes needed to have a client ready to display the CUDA results –given the classes explained in the next section– are simple. Listing 6.1 shows the necessary changes to the client class implemented as a static method that adds the entries to the `ConnectionManager` instance.

On the server side, all that is left to do is to insert the writer object after an instance of the server has been created. How these reader, writer and drawer classes have to be constructed will be explained in the next section.

Listing 6.1: OTFClient class for CUDA

```
public class OTFCUDAClient extends OnTheFlyClientQuad {
  public static OTFCUDAClient createSim(String url) {
    OTFConnectionManager connect = new OTFConnectionManager();
    connect.add(QueueLink.class, OTFNoDynDataLinkHandler.Writer
        .class);

    connect.add(OTFAgentsVBOHandler.Writer.class,
        OTFAgentsVBOHandler.class);
    connect.add(OTFAgentsVBOHandler.class, OTFExternVBODrawer.
        class);
    connect.add(OTFDefaultNodeHandler.Writer.class,
        OTFDefaultNodeHandler.class);

    connect.add(OTFNoDynDataLinkHandler.Writer.class,
        OTFNoDynDataLinkHandler.class);
    connect.add(OTFNoDynDataLinkHandler.class,
        SimpleStaticNetLayer.SimpleQuadDrawer.class);
    connect.add(SimpleStaticNetLayer.SimpleQuadDrawer.class,
        SimpleStaticNetLayer.class);
    return new OTFCUDAClient(url, connect);
  }
}
}
```

6.2 Using device memory to exchange data

To visualize the CUDA simulation runs, both the visualizer and the CUDA simulation have to be extended.

In the regular `QueueSimulation` all links are responsible for informing the visualizer where to draw the agents currently driving upon themselves. It is possible to extend this mechanism to the CUDA simulation by having all links output the current vehicle position every time step or on request. Unfortunately, this would include transferring the necessary position data from the CUDA device to the host's main memory and then back to the graphics devices as *OpenGL* arrays.

This is clearly not the most efficient way to handle the given situation. As all calculations already take place on the graphics device, it would be more efficient to leave the data on the device and just send a pointer to the location where the data resides to the *OpenGL* framework, which can then display it. This –rather obvious– demand has been foreseen by the NVIDIA engineers, and the necessary functionality for transferring data from the CUDA side of the graphics device to the graphics part is already available as part of the CUDA SDK.

With the `cudaGLMapBufferObject(void** p, GLint vbo)` method, any *OpenGL* vertex buffer array can be mapped to a CUDA memory-area pointer. Therefore, it is easy to register a VBO (vertex buffer object) on the *OpenGL* side and use this to store values actually being computed by CUDA kernels.

Listing 6.2 shows the code for generating a VBO capable of holding a Vertex2 structure for every vehicle in the simulation and binding it to a CUDA pointer. After the CUDA related operations have been finished, the memory location has to be released again by the CUDA context by calling the `cudaGLUnmapBufferObject(GLint vbo)` method. This is mandatory so that the VBO object can be used from the *OpenGL* side again.

6.3 Calling the kernel to render the positions into a VBO

Given the premises of the last section, it was self-evident to implement the updating routine for the agents' –respectively vehicles'– positions as another CUDA kernel, writing the position data into a VBO. Sending the VBO's identifier together with the number of vehicles to be drawn to the visualizer is sufficient to draw from this buffer. Therefore, it is not necessary to transfer any vehicle data back to the host's memory.

The complete host code for calling the kernel is given in listing 6.3

Listing 6.3: Calling the kernel for the VBO update

```
void CMicroSim::updateVBO() {
  if(vbo <= 0) return;
```

Listing 6.2: Use of a VBO on the CUDA device

```
  GLint bsize = sizeof(Vertex2) * simdata.planssize;
  glGenBuffersARB(1, &vbo);
  glBindBufferARB(GL_ARRAY_BUFFER_ARB, vbo);
  glBufferDataARB(GL_ARRAY_BUFFER_ARB, bsize,
          NULL, GL_DYNAMIC_DRAW_ARB);
  glGetBufferParameterivARB(GL_ARRAY_BUFFER_ARB,
      GL_BUFFER_SIZE_ARB, &bsize);
  if (bsize != (sizeof(Vertex2) * simdata.planssize)) exit(1);

  glBindBufferARB(GL_ARRAY_BUFFER_ARB, 0);
  cudaGLRegisterBufferObject(vbo);
  CUT_CHECK_ERROR("cudaGLRegisterBufferObject failed");

  Vertex2 *p;

  unsigned long error = cudaGLMapBufferObject((void**)&p, vbo);
}
```

```
  int lthread = min(THREADMIN,simdata.linkscount);
  int lblock = ceil((double)simdata.linkscount / lthread);

  unsigned long error = cudaGLMapBufferObject((void**)&p, vbo);

  vehpoints = 0;
  cudaMemcpy(dvehpos, &vehpoints, sizeof(int),
      cudaMemcpyHostToDevice);
  calcVehPos_r<<<lblock,lthread>>>(simtime, p, dvehpos);
  cudaMemcpy(&vehpoints, dvehpos, sizeof(int),
      cudaMemcpyDeviceToHost);

  cudaGLUnmapBufferObject(vbo);
}
}
```

The kernel responsible for drawing the vehicles runs over all links. Its code is rather lengthy and was thus omitted here. The kernel calculates how many agents' positions have to be drawn for a particular link and chooses a color for the agents depending on their status in the queue. If a link and the associated buffer are empty, the kernel will bail out right away. Otherwise, vehicles on the link will be drawn in green, while vehicles residing in the link's buffer will be drawn in purple.

The position of the vehicles is calculated as equidistant over the whole link. Therefore, links with less allocation will look less crowded in the visualizer than congested links. For each vehicle, a 2D coordinate and a color value is saved into the vertex buffer. The number of agents to be drawn

later by the *OpenGL* framework is stored in `dvehpos`. This is done by an `atomicAdd()` operation. The actual number of agents and the VBO identifier are the only information transmitted via the host memory.

6.4 The handler for client-server communication

On the server side a new `OTFDataWriter` class was written for transferring the VBO-related data. The class `OTFAgentsVBOHandler` serves as a handler class for reading and writing the data. The inner class `Writer` writes the VBO's identifier and the actual number of vehicles that need to be drawn on screen at a particular time step as dynamic data. That is, it transfers only two integer values as data. This is a grat improvement over the regular drawer, which had to send the color and position data of every agent over the network at every time step drawn.

6.5 The drawing on the client side

Two more classes had to be implemented on the client side for the drawing of the VBO's content. The interface `OTFDataVBO` derived from `OTFData` to define the methods of the `OTFData.Receiver` class. These are methods for receiving the VBO's *ID* and the vehicle count. Finally, a drawer class which could display the VBO in the *OpenGL* context had to be written. The `OTFExternVBODrawer` class consists of only a few lines of code and is given in listing 6.4

With these classes installed, the CUDA simulation can draw the agents by sharing a common area of memory between *OpenGL* and CUDA.

6.6 Result from interactive runs

To see if any speedup was achieved when running the CUDA based simulation inside the OTFVis, the times for two configurations were taken. The GTX280 system was run with the 10% sample. One run was started with the regular `QueueSimulation` serving as the "physical reality" the other run was started with the `JCudaSim`.

In the OTFVis it is possible to "jump" directly to a new timestep by entering time time into the time field. Or one could press the play button and watch the simulation progress in time. To get an impression of the run time of both ways to step forward two different times were tested in both runs: The running time for stepping 24 hours forward with and without displaying the graphics. The results can be found in Table 6.1. As the results show, the `JCudaSim` is about eight times faster than the Java in all variations. This is nearly an order of magnitude faster in execution speed. Therefore, the overall goal of reaching interactive frame rates with the aid

Listing 6.4: The drawer for drawing from the VBO

```
public class OTFExternVBODrawer extends AgentArrayDrawer
    implements OTFDataVBO.Receiver {
  private int count = 0;
  private int vbo = 0;

  protected void drawArray(GL gl) {
    int vertex2size=12;
    gl.glBindBufferARB(GL.GL_ARRAY_BUFFER_ARB, vbo);
    gl.glVertexPointer(2, GL.GL_FLOAT, vertex2size, 0);
    gl.glColorPointer(3, GL.GL_BYTE, vertex2size, 0);
    gl.glInterleavedArrays(GL.GL_C4UB_V2F, vertex2size, 0);
    gl.glDrawArrays(GL.GL_POINTS, 0, count);
    gl.glBindBufferARB(GL.GL_ARRAY_BUFFER_ARB, 0);
  }

  public void setVBO(int vbo) {
    this.vbo = vbo;
  }
  public void setVehCount(int count) {
    this.count = count;
  }
}
}
```

GTX280	CUDA	Java
10% w/o graphics	18s	157s
10% with graphics	28s	207s
25% w/o graphics	29s	235s
25% with graphics	42s	335s

Table 6.1: Running times for stepping 24 hours forward with the OTFVis using CUDA and Java

of modern graphics hardware has been accomplished. Although the peak performance of executing the simulation in C++ with an speedup of 38 could not be maintained after the integration into the Java framework, a reasonable speedup still remains, making it possible to shift the time to execute the simulation into the time frame required to maintain interaction.

Chapter 7

Conclusion and Outlook

This chapter comprises a brief summary of the work presented in this thesis as well as an account of some of the latest developments of the visualizer. As the visualizer is an open and "living" project, additions are constantly being made to it. Some contributions by the author of this thesis will be described.

7.1 Running the server as an external module

As seen in Section 5.4.2 the `OnTheFlyServer` class needs to be called from within the *QueueSimulation*. The *QueueSimulation* had to be extended in order to overload the `afterSimStep()` method and the server called from there. This is rather inconvenient as one must decide before starting the simulation whether to look into the simulation or not by choosing either the original *QueueSimulation* or the derived class.

A solution to this has recently been added to the MATSim project. By "firing" certain events, signaling the different states of the simulation, external classes can be informed of the state changes. One of the events fired is called `QueueSimulationAfterSimStepEvent`. This event is sent whenever a simulation step has finished. By making the `OnTheFlyServer` aware of this event and registerinf it as an event listener to the currently running *QueueSimulation*, the need to overload the method called after each simulation step is eliminated. It is no longer necessary to use a special version of the *QueueSimulation* to cooperate with the `OnTheFlyServer`.

How this is done and how to take this approach even one step further is shown in the next section.

7.1.1 Running several iterations

Given the high-speed simulation of the CUDA device, it is feasible to run the complete controller framework in the visualizer, giving the user the pos-

sibility to run more than one iteration inside the visualizer's view. The class responsible for executing MATSim iterations is the `Controler` class. To use the `JCudaSim` instead of the regular `QueueSimulation` class, the `Controler` class needs to be extended. By inheriting and overriding the `doMobsim()` method the regular simulation can be replaced by the `JCudaSim` that is running the simulation on the graphics device. This small change makes it possible to run iterations on the CUDA device.

The second step towards running and experiencing the MATSim iteration process through the visualizer is to enable the visualizer to cope with more than one run of the "physical" layer and to provide meaningful information while the "strategical layer" is occupied with the replanning process.

To start with, the `OnTheFlyServer` instance which, up to this point, was created inside the start-up code of the *QueueSimulation*, had to be created by the `Controler` class and held persistent over several iterations of the "physical" layer. Because the MATSim framework offers the opportunity to register external classes as "listeners" to both an internal event inside a *QueueSimulation* and inside the `Controler` class, the following approach was chosen.

An `ExeViaListener` class implementing several interfaces was created and inserted as a listener to both the simulation and the controller. Furthermore, the `OnTheFlyServer` interface was extended to hold another method, `getControllerStatus()`. This method sends status information whenever the controller or the strategic layer is active. Listing 7.1 shows the `ExeViaListener` class reacting to several events in the controller and simulation and taking meaningful action. The code should be self-explanatory; notice how the overloaded `notifyAfterMobsim()` method is used to issue the server's `getStatus()` method, which used to reside inside the QueueSimulation itself.

Listing 7.1: Running the server as an external module

```
public static class OTFControlerListener implements
StartupListener,BeforeMobsimListener,
AfterMobsimListener, QueueSimulationInitializedListener,
QueueSimulationAfterSimStepListener {

  private QueueNetwork queueNetwork;
  private OnTheFlyServer myOTFServer;
  private Population population;
  private Events events;

  public void notifyBeforeMobsim(BeforeMobsimEvent event) {
    Controler cont = event.getControler();
    myOTFServer.setControllerStatus(OTFVisController.RUNNING +
        cont.getIteration());
  }
```

```
  public void notifyAfterMobsim(AfterMobsimEvent event) {
    Controler cont = event.getControler();
    myOTFServer.setControllerStatus(OTFVisController.REPLANNING
        + cont.getIteration()+1);
  }

  public void notifySimulationAfterSimStep(
      QueueSimulationAfterSimStepEvent e) {
    int status = myOTFServer.updateStatus(e.getSimulationTime()
        );
  }

  public void notifySimulationInitialized(
      QueueSimulationInitializedEvent e) {
    QueueSimulation q = e.getQueueSimulation();
    myOTFServer.events = QueueSimulation.getEvents();
    myOTFServer.replaceQueueNetwork(q.getQueueNetwork());
  }

  public void notifyStartup(StartupEvent event) {
    UUID idOne = UUID.randomUUID();
    Controler cont = event.getControler();
    this.population = cont.getPopulation();
    this.events = cont.getEvents();
    this.queueNetwork = new QueueNetwork(cont.getNetwork());
    this.myOTFServer = OnTheFlyServer.createInstance("
        OTFServer_" + idOne.toString(), this.queueNetwork, this
        .population, this.events, false);
    myOTFServer.setControllerStatus(OTFVisController.STARTUP);
  }
}
```

The listing shows the actual implementation for the regular *QueueSimulation*. For the CUDA simulation to run, the controller still has to be derived to override the `runMobSim()` method and instantiate a `JCudaSim` object instead of the regular one. Running the simulation over several iterations also means that the plans of the agents will change and a score for each agent will be calculated. This makes it even more interesting to be able to inspect the inner state of an agent.

7.2 Adding reflection

The *Java reflection* API is part of the Java SDK. With this API it is possible to inspect every aspect of each class and object without having any previous knowledge about the object. This API is particularly valuable with regard to the objects' fields and their states. One can recursively step through each field of an object and display its value without any implementational

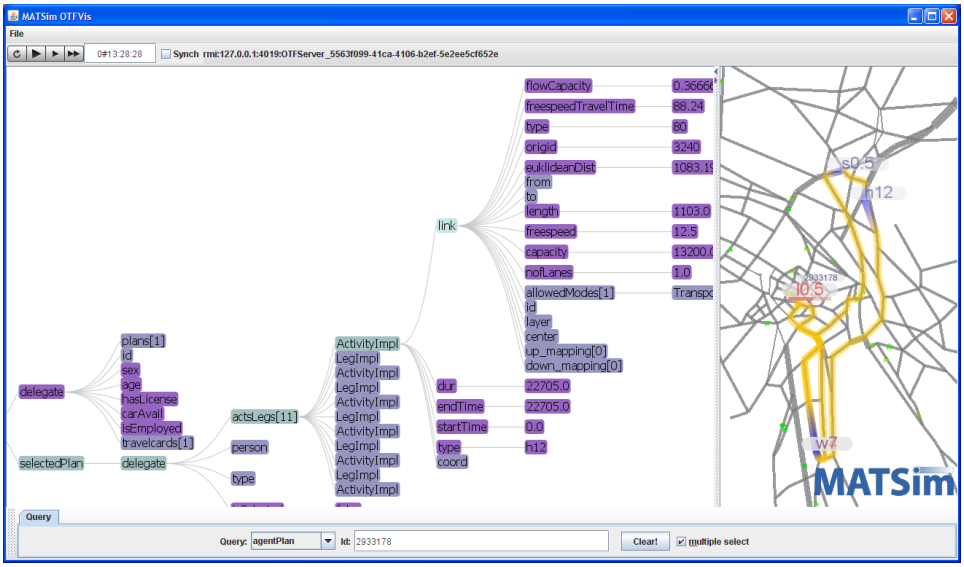


Figure 7.1: An agent's plan visualized using PREFUSE

premises like providing `getValueXX()` methods. The *Java reflection* API was used together with the remarkable *PREFUSE* visualization library to display the inner state of each agent, regardless of its class. By stepping recursively through the fields of the agent and collecting the state of the agent's fields, a tree is constructed reflecting the complete inner state of the agent database. It is, for example, possible to interactively browse all nodes of an agent's current route or investigate the score of an agent's plan. Figure 7.1 shows an expanded view of an agent.

7.3 Summary

Investigating the technical issues of an interaction with mobile intelligent particles was the mission of this thesis. The challenges to be solved were diagnosed as

- providing means to explore the huge data sets representing the agents and the network
- providing visualizations to help the user understand the inner states of the agents and
- providing sufficient speedup to narrow the time domain of the problem to an interactively manageable time frame.

A visualization and interaction framework was designed with extensibility and flexibility in mind. The main features were :

- MVC-pattern-based
- client-server architecture to run over the network
- separate data-representation and drawing
- easy to extend with self-written classes
- high-speed visualizer for displaying hundreds of thousands of agents
- fall-back solution running on every computer with Java SDK installed

The given implementation of the *QueueSimulation* proved to be too slow to interactively handling a physical simulation of transport. A new generation of graphics hardware was used to decrease the execution time necessary to run the simulation. The NVIDA CUDA framework offers tremendous calculation power for very little money. A €300 graphics card was chosen, and a variant of the *QueueSimulation* was implemented and run on the graphics device using the CUDA framework. The real-time ratio possible with the CUDA device is on the order of 10^4.

This was deemed fast enough to generate interactive frame rates even for larger agent samples.

The implementations described in the previous chapters brought together the CUDA simulation and the new *OpenGL*-based visualizer. Furthermore, whole iterations of simulations could be run interactively.

By using the CUDA simulation and the speedup possible with this implementation, it is feasible to run iterations of samples with hundreds of thousands of agents in less than a minute, enabling the researcher to examine the simulation "live" on screen instead of having to rely on pre-recorded or aggregated data. Given the fast execution times, it is feasible to conduct research interactively simply by re-running a certain iteration to inspect the events leading to an observed (mis-) behavior. Additionally, all kinds of queries can be sent into the population database to further narrow down the causes. As writing new kinds of queries is rather straightforward, it is hoped that, after a certain period of time, a widespread toolbox of queries and additional visualization modules will become available for data mining into the behaviour of the agent and into the complex, emergent group behavior shown by multi-agent simulations.

The opportunity to visually interact with the spatial data of a traffic simulation, will hopefully reveal new trails of emergent behavior that would have been left unaccounted for by aggregated data. Even the *post-mortem* usage of a recorded movie file cannot deliver the bandwidth of research opportunities that the interaction with the living simulation system can offer.

Bibliography

[1] Paper, Transportation Research Board Annual Meeting, Washington, D.C.

[2] *Proceedings of the meeting of the International Association for Travel Behavior Research (IATBR)*, Kyoto, Japan, 2006. See www.iatbr.org.

[3] SWIG: Simplified Wrapper and Interface Generator. http://www.swig.org/, accessed 02/2009.

[4] JOGL: Java Binding For OpenGL. https://jogl.dev.java.net/, accessed 04/2009.

[5] CXXWrap: Java JNI wrapper generator. http://cxxwrap.sourceforge.net/, accessed 08/2008.

[6] G. Abram and L. Treinish. An extended data-flow architecture for data analysis and visualization. In *Proceedings of the 6th conference on Visualization'95*. IEEE Computer Society Washington, DC, USA, 1995.

[7] T. Arentze, F. Hofman, H. van Mourik, and H. Timmermans. ALBATROSS: A multi-agent rule-based model of activity pattern decisions. Paper 00-0022, Transportation Research Board Annual Meeting, Washington, D.C., 2000.

[8] V. Astarita, K. Er-Rafia, M. Florian, M. Mahut, and S. Velan. A comparison of three methods for dynamic network loading. *Transportation Research Record*, 1771:179–190, 2001.

[9] ATI FireStream www page. Firestream: DAAMIT GPGPU framework. ati.amd.com/technology/streamcomputing/index.html, accessed 08/2008.

[10] K. Axhausen. *Eine ereignisorientierte Simulation von Aktivitätenketten zur Parkstandswahl*. PhD thesis, Universität Karlsruhe, Germany, 1988.

[11] K. Axhausen and T. Gärling. Activity-based approaches to travel analysis: conceptual frameworks, models, and research problems. *Transport Reviews*, 12(4):323–341, 1992.

[12] K. Axhausen and R. Herz. Simulating activity chains: German approach. *Journal of Transportation Engineering*, 115(3):316–325, 1989.

[13] M. Balmer. *Travel demand modeling for multi-agent transport simulations: Algorithms and systems*. PhD thesis, Swiss Federal Institute of Technology (ETH) Zürich, Switzerland, 2007.

[14] M. Balmer and K. Nagel. Shape morphing of intersection layouts using curb side oriented driver simulation. In J. Van Leeuwen and H. Timmermans, editors, *Innovations in Design & Decision Support Systems in Architecture and Urban Planning*, pages 167–183. Springer, Heidelberg/Berlin, 2006.

[15] M. Balmer, M. Rieser, K. Meister, D. Charypar, N. Lefebvre, K. Nagel, and K. Axhausen. MATSim-T: Architektur und Rechenzeiten. In *paper presented at Heureka '08*, Stuttgart, Germany, March 2008.

[16] Y. Berra. Yogiisms of lawrence peter berra. http://www.umpirebob.com/DATA/yogiisms.htm, accessed 04/2009.

[17] C. Bhat, J. Guo, S. Srinivasan, and A. Sivakumar. A comprehensive econometric microsimulator for daily activity-travel patterns. *Transportation Research Record*, 1894:57–66, 2004.

[18] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of political economy*, 81(3):637, 1973.

[19] J. Bottom. *Consistent anticipatory route guidance*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 2000.

[20] J. Bowman, M. Bradley, Y. Shiftan, T. Lawton, and M. Ben-Akiva. Demonstration of an activity-based model for Portland. In *World Transport Research: Selected Proceedings of the 8th World Conference on Transport Research 1998*, volume 3, pages 171–184. Elsevier, Oxford, 1998.

[21] I. Buck. Brook specification v.02. http://merrimac.stanford.edu/brook/brookspec-v0.2.pdf, 10 2003. accessed 08/2008.

[22] D. Burns and R. Osfield. Open scene graph a: Introduction, b: Examples and applications www.openscenegraph.org. In *Proceedings of the IEEE Virtual Reality 2004*. IEEE Computer Society Washington, DC, USA, 2004.

[23] C. Cantarella and E. Cascetta. Dynamic process and equilibrium in transportation network: Towards a unifying theory. *Transportation Science A*, 25(4):305–329, 1995.

[24] N. Cetin. *Large-Scale parallel graph-based simulations.* PhD thesis, Swiss Federal Institute of Technology (ETH) Zürich, Switzerland, 2005.

[25] N. Cetin, A. Burri, and K. Nagel. Parallel queue model approach to traffic microsimulations. In *In Proceedings of Swiss Transportation Research Conference*, 2002.

[26] D. Charypar, K. Axhausen, and K. Nagel. An event-driven parallel queue-based microsimulation for large scale traffic scenarios. 2007.

[27] D. Charypar, K. Axhausen, and K. Nagel. Event-driven queue-based traffic flow microsimulation. *Transportation Research Record*, 2003:35–40, 2007.

[28] D. Charypar and K. Nagel. Generating complete all-day activity plans with genetic algorithms. *Transportation*, 32(4):369–397, 2005.

[29] C. Chen and T. Rhyne. Visualization viewpoints. *IEEE Computer Graphics and Applications*, 2005.

[30] Y. Chen, M. Rieser, D. Grether, and K. Nagel. Improving a large-scale agent-based simulation scenario. Paper, Transportation Research Board Annual Meeting, Washington, D.C., submitted.

[31] CUDA www page. CUDA: NVIDIA GPGPU framework. www.nvidia.com/object/cuda_home.html, accessed 08/2008.

[32] C. Daganzo. Queue spillovers in transportation networks with a route choice. *Transportation Science*, 32(1):3–11, 1998.

[33] A. de Palma and F. Marchal. 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.

[34] T. DeFanti, M. Brown, and B. McCormick. Visualization: expanding scientific and engineering researchopportunities. *Computer*, 22(8):12–16, 1989.

[35] D. DeWitt, N. Kabra, J. Luo, J. Patel, and J. Yu. Client-server paradise. In *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*, pages 558–558. INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE), 1994.

[36] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269 – 271, 1959.

[37] DYNAMIT/MITSIM www page. http://mit.edu/its, accessed 2008.

[38] J. Ferber. *Multi-agent systems. An Introduction to distributed artificial intelligence*. Addison-Wesley, 1999.

[39] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, March 1974.

[40] T. Funkhouser. RING: A client-server system for multi-user virtual environments. In *Proceedings of the 1995 symposium on Interactive 3D graphics*. ACM New York, NY, USA, 1995.

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[42] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Pattern: Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series. Addison-Wesley, 2001.

[43] C. Gawron. *Simulation-based traffic assignment*. PhD thesis, University of Cologne, Cologne, Germany, 1998.

[44] C. D. Gloor. *Distributed Intelligence in Real World Mobility Simulations*. PhD thesis, ETH - SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH, 2005. thesisType=phd; department=Technical Sciences;.

[45] T. Gormen, C. Leiserson, and R. Rivest. *Introduction to algorithms*. MIT Press/McGraw-Hill, 1990.

[46] J. Hackney and K. Axhausen. An agent model of social network and travel behavior interdependence. In *Proceedings of the meeting of the International Association for Travel Behavior Research (IATBR)* [2]. See www.iatbr.org.

[47] T. Hardware. Cpu comparison and benchmark. http://www.tomshardware.com/de/testberichte-235198-75.html, accessed 05/2009.

[48] hardware infos.com. ATI AMD graphic devices list. http://www.hardware-infos.com/grafikkarten_ati.php, accessed 05/2009.

[49] hardware infos.com. Nvidia graphic devices list. http://www.hardware-infos.com/grafikkarten_nvidia.php, accessed 05/2009.

[50] M. Harris. CUDA workshop pre-ISC2008, dresden, Juni 2008.

[51] D. Helbing and K. Nagel. The physics of traffic and regional development. *Contemporary Physics*, 45(5):405–426, 2004.

[52] D. Hensher and K. Button, editors. Elsevier, Oxford, 2nd edition, 2008.

[53] J. Holland. *Adaptation in Natural and Artificial Systems.* Bradford Books, 1992. Reprint edition.

[54] J. H. Holland. Using classifier systems to study adaptive nonlinear networks. In D. Stein, editor, *Lectures in the sciences of complexity.* Addison-Wesley Longman, 1989.

[55] L. Howes and D. Thomas. Efficient random number generation and application using CUDA. *GPU Gems*, 3, 2007.

[56] Intel coorperation. Intel processors sorted by release date. http://www.intel.com/pressroom/kits/quickrefyr.htm, accessed 05/2009.

[57] Java Documentation. Java Serialization documentation and whitepaper. http://java.sun.com/javase/6/docs/platform/serialization/spec/serialTOC.html, accessed 04/2009.

[58] C. Jones. *Visualization and optimization.* Kluwer Academic Pub, 1996.

[59] D. Kaufman, K. Wunderlich, and R. Smith. An iterative routing/assignment method for anticipatory real-time route guidance. Technical Report IVHS Technical Report 91-02, University of Michigan Department of Industrial and Operations Engineering, Ann Arbor MI, May 1991.

[60] A. Klinger. Patterns and search statistics. In *Optimizing Methods in Statistics: Proceedings*, page 303. Academic Press, 1971.

[61] G. Krasner and S. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object-oriented programming*, 1(3):26–49, 1988.

[62] E. Kutter. Integrierte Berechnung städtischen Personenverkehrs – Dokumentation der Entwicklung eines Verkehrsberechnungsmodells für die Verkehrsentwicklungsplanung Berlin (West). Arbeitsberichte zur Integrierten Verkehrsplanung, TU Berlin, Berlin, 1984.

[63] N. Lefebvre and M. Balmer. Fast shortest path computation in time-dependent traffic networks. In *Proceedings of Swiss Transport Research Conference (STRC)*, Monte Verita, CH, September 2007. See www.strc.ch.

[64] D. Lohse, G. Bachner, B. Dugge, and H. Teicher. Ermittlung von Verkehrsströmen mit n-linearen Gleichungssystemen unter Beachtung von Nebenbedingungen einschließlich Parameterschätzung (Verkehrsnachfragemodellierung: Erzeugung, Verteilung, Aufteilung). Technical Report 5/1997, Institut für Verkehrsplanung und Straßenverkehr, Dresden, Germany, 1997.

[65] MASON www page. MASON multi agent framework. http://www.cs.gmu.edu/ eclab/projects/mason/, accessed 02/2009.

[66] MATSim www page. MATSim multi agent transport simulation framework. http://www.matsim.org, accessed 02/2009.

[67] J. D. McCalpin. STREAM: sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream/, accessed 04/2009.

[68] B. McCormick, T. DeFanti, and M. Brown. Visualization in scientific computing (special issue). *Computer Graphics*, 21(6), 1987.

[69] K. Meister. Erzeugung kompletter Aktivitätenpläne für Haushalte mit genetischen Algorithmen. Master's thesis, IVT, ETH Zürich, 2004.

[70] K. Meister, M. Balmer, K. Axhausen, and K. Nagel. planomat: A comprehensive scheduler for a large-scale multi-agent transportation simulation. In *Proceedings of the meeting of the International Association for Travel Behavior Research (IATBR)* [2]. See www.iatbr.org.

[71] MITSIM, 1999. Massachusetts Institute of Technology, Cambridge, Massachusetts. See its.mit.edu.

[72] K. Nagel. *High-speed microsimulations of traffic flow.* PhD thesis, University of Cologne, 1995. See www.zaik.uni-koeln.de/~paper.

[73] J. Nash. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences of the United States of America*, pages 48–49, 1950.

[74] R. Nelson and H. Samet. A consistent hierarchical representation for vector data. *ACM SIGGRAPH Computer Graphics*, 20(4):197–206, 1986.

[75] R. Nelson and H. Samet. A population analysis for hierarchical data structures. *ACM SIGMOD Record*, 16(3):270–277, 1987.

[76] OpenGL www page. OpenGL graphics framework. http://www.opengl.org, accessed 02/2009.

[77] J. Orenstein. Multidimensional tries used for associative searching. Inf. *Proc. Lett*, 14(4), 1982.

[78] J. d. D. Ortúzar and L. Willumsen. *Modelling transport*. Wiley, Chichester, 1995.

[79] B. Park and J. Schneeberger. Microscopic simulation model calibration and validation: A case study of VISSIM for a coordinated actuated signal system. *Transportation Research Record*, 1856:185–192, 2003.

[80] R. Pendyala. Phased implementation of a multimodal activity-based travel demand modeling system in florida. volume II: FAMOS users guide. Research report, Florida Department of Transportation, Tallahassee, 2004. See www.eng.usf.edu/˜pendyala/publications.

[81] PTV. Traffic Mobility Logistics. webpage, accessed 2008. See www.ptv.de.

[82] B. Raney and K. Nagel. An improved framework for large-scale multiagent simulations of travel behaviour. In P. Rietveld, B. Jourquin, and K. Westin, editors, *Towards better performing European Transportation Systems*, page 42. Routledge, London, 2006.

[83] Repast www page. Repast, Java multi agent framework. http://repast.sourceforge.net/, accessed 02/2009.

[84] J. Rohlf and J. Helman. Iris performer: a high performance multiprocessing toolkit for real-time 3d graphics. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 381–394, New York, NY, USA, 1994. ACM.

[85] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. In *SIGMOD '85: Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, pages 17–31, New York, NY, USA, 1985. ACM.

[86] S. Rusinkiewicz and M. Levoy. Streaming QSplat: A viewer for networked visualization of large, dense models. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 63–68. ACM New York, NY, USA, 2001.

[87] P. Salvini and E. Miller. ILUTE: An operational prototype of a comprehensive microsimulation model of urban systems. *Network and Spatial Economics*, 5(2):217–234, 2005.

[88] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.

[89] H. Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann, August 2006.

[90] N. Sawant, C. Scharver, J. Leigh, A. Johnson, G. Reinhart, E. Creel, S. Batchu, S. Bailey, and R. Grossman. The tele-immersive data explorer: A distributed architecture for collaborative interactive visualization of large data-sets. In *Proceedings of the Fourth International Immersive Projection Technology Workshop*, 2000.

[91] S. Schnittger and D. Zumkeller. Longitudinal microsimulation as a tool to merge transport planning and traffic engineering models - the MobiTopp model. In *Proceedings of the European Transport Conference*, Strasbourg, October 2004.

[92] Y. Sheffi. *Urban Transportation Networks: Equilibrium Analysis with Mathematical Programming Methods*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1985.

[93] H. Simão and W. Powell. Numerical methods for simulating transient, stochastic queueing networks. *Transportation Science*, 26:296–311, 1992.

[94] W. Stallings. Queuing analysis. ftp://shell.shore.net/members/w/s/ws/Support/ QueuingAnalysis.pdf, 2000.

[95] P. S. Strauss and R. Carey. An object-oriented 3d graphics toolkit. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 341–349, New York, NY, USA, 1992. ACM.

[96] H. Sutter. The free lunch is over. *Dr. Dobb's Journal*, 3, 2005.

[97] H. Sutter. Machine architecture: Things your programming language never told you. Northwest C++ User Meeting, 11 2007.

[98] T. Titze. Entwicklung eines ÖV-Routingmoduls für Multiagentensimulationen. Term project report, TU Berlin, 2007.

[99] TRANSIMS. TRansportation ANalysis and SIMulation System, Dec. accessed 2008. See transims.tsasa.lanl.gov.

[100] K. Vaughn, P. Speckman, and E. Pas. Generating household activity-travel patterns (HATPs) for synthetic populations, 1997.

[101] W. S. Vickrey. Congestion theory and transport investment. *The American Economic Review*, 59(2):251–260, 1969.

[102] VISSIM www page. www.ptv.de, accessed 2004. Planung Transport und Verkehr (PTV) GmbH.

[103] P. Vovsha, E. Petersen, and R. Donnelly. Microsimulation in travel demand modeling: lessons learned from the New York best practice model. *Transportation Research Record*, 1805:68–77, 2002.

[104] P. Waddell, A. Borning, M. Noth, G. Ulfarsson, N. Freier, and M. Becke. UrbanSim: A simulation system for land use and transportation. Working Paper, University of Washington, Seattle, 2001.

[105] J. Wardrop. Some theoretical aspects of road traffic research. *Proceedings of the Institute of Civil Engineers*, 1:325–378, 1952.

[106] R. Wiedemann. Simulation des Strassenverkehrsflusses. Schriftenreihe Heft 8, Institute for Transportation Science, University of Karlsruhe, Germany, 1974.

[107] R. Wiedemann. Beschreibung des Staus. In H. Keller, editor, *Beiträge zur Theorie des Straßenverkehrs*. Forschungsgesellschaft für Straßen- und Verkehrswesen, Köln, Germany, 1995.

[108] wwwBlackScholes. http://www.cse.ucsd.edu/~goguen/courses/130/SayBlackScholes.html, accessed 04/2009.