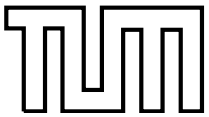


LEHRSTUHL FÜR REALZEIT-COMPUTERSYSTEME
TECHNISCHE UNIVERSITÄT MÜNCHEN
UNIV.-PROF. DR.-ING. G. FÄRBER

Graphics for a 3D Driving Simulator

Johannes Georg Brand

Bachelor Thesis

Graphics for a 3D Driving Simulator

Bachelor Thesis

Supervised by the Institute for Real-Time Computer Systems
Technische Universität München
Prof. Dr.-Ing. Georg Färber

Conducted at Robotics and Automation Lab
Center for Intelligent Information Processing Systems
University of Western Australia
Perth

Advisor: Assoc. Prof. Dr. rer. nat. habil. Thomas Bräunl
Adrian Boeing
Dipl.-Ing. Philipp Harms

Author: Johannes Georg Brand
Eulenweg 2
85356 Freising

Submitted 3rd March 2008

Contents

List of Figures

List of Tables

List of Symbols

RCS	Lehrstuhl für Realzeit-Computersysteme
UWA	University of Western Australia
NASA	National Aeronautics and Space Administration
DARPA	Defense Advanced Research Projects Agency
US	United States
AI	Artificial Intelligence
RARS	Robot Auto Racing Simulation
SDK	Software Development Kit
IDE	Integrated development environment
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GUI	Graphical User Interface
FLTK	Fast Light Toolkit
XML	Extensible Markup Language
HTML	Hypertext Markup Language
OSM	OpenStreetMap
UML	Unified Modeling Language
DLL	Dynamic Link Library
IP	Internet Protocol
GLSL	OpenGL Shading Language
HLSL	High Level Shading Language
PSD	Position Sensitive Device

Abstract

Due to the continuously growing amount of traffic on our roads, and with it, an increasing risk of car accidents, the theory and design of robotic cars that take away the risk of human driver errors has become an area of active research among car manufacturers, suppliers and universities. Even if various competitions for autonomous driving cars have introduced some excellent testing venues in real world environments, decent simulators for testing the robotic programs without hardware do not exist. In the light of this an extensible, modular and flexible open source component-based 3D driving simulator framework called AutoSim is presented in this thesis. The framework enables rapid and interactive development of robotic car algorithms and allows researchers to develop, test and experiment with autonomous vehicle software without the need for a physical vehicle. Using a testing environment like AutoSim can tremendously increase the speed of development and thus save costs by reducing time to market.

Constructing a simulator framework includes meeting a large range of demands for creating a high-level 3D environment. A large 3D scene has to be adapted for execution on limited computer hardware and nevertheless provide the user all features desired for interaction with the tested robots. This thesis follows the ideas of methods like *GeoMipMap* and *Geometric Clipmaps* by applying them to the simulator's requirements and performing parallel operations on the graphics card. The road system in AutoSim is designed in order to display an accurate representation of a real world street web, implemented through self-developed methods for procedurally constructing road meshes out of 2D world road data and transforming them by height data. Finally, the thesis introduces approaches for designing a 3D simulator framework which includes concepts for transferring and managing data as well as multiple rendering methods.

Zusammenfassung

Aufgrund einer beständig anwachsenden Verkehrsdichte auf unseren Straßen und eines damit verbundenen immer größer werdenden Unfallrisikos, ist das Entwickeln von Roboter Autos, welche die Fehler des menschlichen Fahrers als Unfallrisiko entfernen, ein aktiv umforschtes Gebiet von Fahrzeugherstellern, Lieferanten und Universitäten geworden. Auch wenn verschiedene Wettbewerbe schon exzellente Testumgebungen für autonome Fahrzeuge bereitgestellt haben, gibt es noch immer keine geeigneten Simulatoren um Roboter Programme ohne Hardware zu testen. Im Angesicht dessen wird in dieser Bachelorarbeit ein erweiterbares, modular aufgebautes und flexibel anwendbares Open Source 3D Fahrsimulator Framework mit dem Namen AutoSim präsentiert. Das Framework ermöglicht schnelles und interaktives Entwickeln von Programmen für Roboter und erlaubt Forschern ohne Gebrauch eines physikalischen Fahrzeugs zu entwickeln, zu testen und zu experimentieren. Eine Testumgebung wie AutoSim kann die Entwicklungszeit eines solchen Programmes enorm beschleunigen und auch wegen eines verkürzten Time-to-Market's Kosten einsparen.

Das Entwickeln eines Fahrsimulator Frameworks erfordert einer großen Menge von Ansprüchen einer hochwertigen 3D Anwendung gerecht zu werden. Eine große 3D Szene muss an die Limitierungen der Computer Hardware angepasst werden und trotzdem alle gewünschten Fähigkeiten zur Interaktion mit den Robotern bereitstellen. Diese Bachelorarbeit übernimmt dazu Ideen von Methoden wie *GeoMipMap* oder *Geometric Clipmaps* um sie an die Anforderungen des Simulators anzupassen und dabei parallele Operationen auf der Grafikkarte auszuführen. Das Straßensystem von AutoSim ist entworfen um das echte Straßennetz möglichst genau wiederzugeben, indem aus 2D Weltdaten durch selbst entwickelte Methoden prozedurale Straßenmeshes erstellt werden, um sie dann durch Höhendaten zu verändern. Abschließend führt die Arbeit noch einige Ansätze auf die das Entwickeln eines 3D Fahrsimulator Frameworks betreffen und geht dabei auf Konzepte wie das Übertragen und Verwalten von Daten sowie mehrere Render Methoden ein.

1 Introduction

This introductory chapter aims to show the underlying motivation of the thesis topic, explains its objectives, and finally gives a brief overview of the thesis outline.

1.1 Motivation

The Mars Pathfinder mission, launched by the NASA on 4 December 1996 [20], was one of the most successful NASA missions after landing on the moon. The Pathfinder lander reached Mars on 4th July 1997 and deployed a small rover (named Sojourner) that first rolled onto Mars' surface on 6th July.

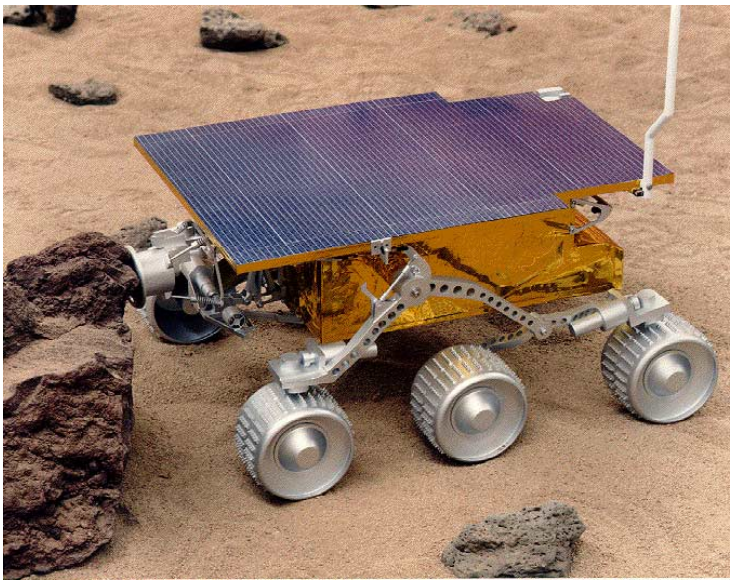


Figure 1.1: Sojourner, Mars rover from Pathfinder mission [29].

The Mars rover is a six-wheeled vehicle which is controlled by an Earth-based operator, using images obtained by both the rover and lander systems. Because the time delay is about 10 minutes to Mars, the rover requires some autonomous control. As the connection of the Sojourner was done via the lander, it could only discover a small area. In 2003 the NASA sent two new rovers to the planet that were now able to have direct contact with the control center on Earth. These two robots are still operational, which demonstrates

the huge success of the mission, as it had been expected they would be driving on Mars for only a few months.

Throughout the whole history of NASA space missions and also throughout all its means of transportation, the Pathfinder mission initiated a new era of locomotion. For the first time a nevertheless still mostly remote controlled NASA driving vehicle had to find its own way for a short time on an unexplored terrain until it received further instructions from the commander.

Even if the NASA Pathfinder mission was one of the first applications of an autonomous vehicle, already in 1980 Ernst Dickmanns and his group at Universität der Bundeswehr Munich (UniBW) built the world's first robot car *VaMoRs* by using saccadic vision, probabilistic approaches such as Kalman filters, and parallel computers [3]. This team continuously improved their vehicles and had highly recognized publications of their work in the ongoing years. In 1995 a S-class mercedes benz (Figure 1.2(b)) of Dickmanns and UniBW was autonomously driving 1678 km on public Autobahns from Munich to Denmark and back, at up to 180 km/h by automatically passing other cars.

(a) First Robot Car VaMoRs [15]

(b) VaMP 1995 [15]

As autonomous driving vehicles are already finding their way into space missions and more recently into some military operations, an active research among manufacturers, suppliers and universities has started to bring these new means of transportation into our daily life. In 2004 the United States Department of Defense started a sponsorship of the DARPA Grand Challenge [9] and thus introduced a competition for driverless vehicles. The Grand Challenge was set up at the Mojave Desert region and the teams from all over the world had to show the navigation capability of their robotic cars on an obstacle filled desert track. While in 2004 no team could successfully complete the race, in 2005 five teams managed the course. In 2007 the DARPA Urban Challenge was initiated, which means the vehicles now have to drive through a street course filled by urban obstacles.

As development of the autonomous driving cars continues, the wish to drive in a real city moves more and more into the focus of the challenging teams. The aim is to replace the driver, who is the biggest threat in daily traffic. This could avoid more than 90% of all the accidents on the streets today [24].

Figure 1.2: Stanley, 2005 Grand Challenge winner from Stanford University [4]

Teams who research in driverless vehicles have to spend a lot of time in testing their programs. The Grand Challenge is already a good venue for testing today, as it provides very realistic conditions for the vehicles. However, a car that should compete in a challenge has to be tested throughout the whole development process. Unfortunately testing of programs that run on hardware are always dependent on the development of that hardware. Vice versa those programs can obstruct the hardware development if they produce damage to the newly built hardware by software errors (very likely to happen).

Testing environments for artificial intelligence (AI) developers need to visualize the reactions of the AI as close to the real world as possible and provide sensor information for evaluating the AI programs. Sensors could be, for example, camera pictures to do image processing, PSD Sensors to specify the distance to a car in front or a Gyroscope Sensor to measure the dihedral angle. To simplify and accelerate the work of the developers a simulator that creates a reproduction of the world would be a desirable improvement. The advance of faster testing could reduce the costs of development, increase development speed and thus make the project more competitive to other teams.

1.2 Objectives

This thesis is part of a project whose goal is to create an open source simulator for any kind of robot, mainly focused on automobiles. A robot represents an interacting object in the simulator's world that should be able to be controlled by programs developed by the user. Therefore these control programs will need to obtain sensor information of the simulated world and especially the robot. The AutoSim framework has to be flexible as it should allow the user to create robots designed for custom requirements, which will be limited by the underlying physics engine. To ensure this limitation is not an obstruction in taking part in physics engines' future development progress AutoSim uses a Physic Abstraction Layer (PAL) [6] to let the user easily swap to other physics engines.

Another objective is to display the simulated world in order to help the user debug their user programs by introducing a further sensor, vision. The visualization is very important in a testing environment like AutoSim, as it is usually the easiest and sometimes also the only way to find errors in the simulation or in the created world. As the framework sets itself the task to simulate a large testing area it requires an intelligent graphics scene that recognizes which objects to render. Furthermore the underlying graphic engine has to be fast in rendering speed, but also capable of providing the newest rendering technologies for presenting a decent world.

As the major goal of the AutoSim framework is the testing of autonomous vehicles, the simulator's world street system is constructed out of real world street data obtained by the open source framework OpenStreetMap. The user can download one part of street data, modify it in an editor, add height data and import terrain textures to the simulated world. Every part of the earth with available data can be constructed.

A framework whose task is to simulate a real world should also be able to provide the user's robots with unexpected actions of other robots in the world. Unforeseen events occur most likely if the world contains many different behaving objects. To perform this task the AutoSim Framework is based on Networking to let many different types of testings be executed in the same simulation environment. A AutoSim server loads and simulates the physics world and transfers the information to the clients that can display the world and control their robots. The server as well as the client program are designed to benefit from the ongoing multi-core processor development, as they are multi threaded and run on more than one CPU.

1.3 Thesis Outline

The main work on the AutoSim framework is distributed into two theses where this mainly concentrates on the client part of the networked simulator, the implementation of the terrain and the construction of the roads. Torsten Sommer completes the description of AutoSim in his thesis *Physics for a 3D driving simulator* [28] that is focused on the server-side implementation and the integration of the user programs. However, as some parts of the project apart from the terrain and the roads also do not need to distinguish between server and client, those are included in this thesis as well.

For generating some AutoSim concepts this thesis endeavored to understand and evaluate existing methods concerning the implementation of 3D applications and that's why it includes many theoretical sections and chapters. Therefore implementation sections explain why certain methods were chosen and related theory provides the user with the necessary background knowledge.

The following chapter (2) will give a short explanation of related work concerning existing 3D driving simulators and review resources evaluated for AutoSim. The third chapter (3) describes the framework, gives an overview of all framework forming programs and finally explains the client software design. Chapter 4 explains the world and robot creation procedures of the AutoSimClient in detail. After that, chapter 5 describes the concept of the terrain including its mesh creation, its implementation on the GPU in a shader and its positioning of further static objects. The construction of the road meshes out of OSM data files is explained by chapter 6. Chapter 7 is concerned with rendering methods used for visualizing the 3D scene and its objects. Finally, a conclusion of the achieved goals and perspectives on future work are presented in chapter 8.

2 Related Work

During the 1990's processor speed in personal computers grew exponentially and for the first time it was possible to create realtime 3D content without spending most of the development costs into computer hardware. Benefitting from the progress, companies and universities started to create driving simulators used for entertainment as well as driver's education courses. Meanwhile heaps of driving simulators can be found either in research or commercial field. This chapter will give a short review of helpful literature for developing the AutoSim software design and methods for implementation and present an assortment of projects and products related to the simulator.

2.1 Literature Review

The following literature and researches have been evaluated and used in the AutoSim-Client.

2.1.1 Design Patterns

Design Patterns: Elements of Reusable Object-Oriented Software [12] is a software engineering book describing recurring solutions to common problems in software design. The book's authors Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides describe a series of classic software design patterns and include examples in *C++* and *Smalltalk*. The book is a classic in the field of software design and even if some of the examples are old and some design patterns are better described in newer articles, it is still a good reference if it comes to search a solution for a software problem. During AutoSim development this book was used for getting a general overview of how to design a software, understanding techniques like the Model/View/Controller concept and adapting various patterns to the AutoSim framework.

Thinking in C++, Vol2: Practical Programming [11] is a book of Bruce Eckel and Chuck Allison that tries to emphasize on the most important practically used programming topics by explaining them in every detail. The book is structured into 3 main parts: *Building Stable Systems*, *The Standard C++ Library* and *Special Topics*. In this thesis the book was helpful in order to prevent AutoSim from the most commonly occurring errors in programs, applying helpful *Standard Library* functions to the framework and

implementing some of the explained software design patterns to the AutoSimClient.

The following passage explains the two most important design patterns used by the AutoSim software:

- **Singleton**: A Singleton design pattern ensures that a class has only one instance and provides a global point of access to it. This is achieved by creating a template class with a private constructor and a global defined pointer to a function returning the *Singleton's* instance. The private constructor ensures the class cannot be created from outside and the static *Singleton* return function either creates a new instance if none is existing or simply returns a pointer to the object. Creating the new *Singleton* instance on the heap and by using compiler instructions for not interrupting the creation process makes the *Singleton* pattern thread safe. A singleton in AutoSim is useful for classes like the controller, which should be accessible from all threads, but only exist once. Therefore every class using this pattern simply has to inherit from the *Singleton* template class.
- **Builder**: The goal of a builder design pattern is to separate the construction of a complex object from its representation so that the same construction process can create different representations. Figure 2.1 visualizes this concept in AutoSim. The abstract *Builder* class always provides an interface that can be called by the *Loader* without knowing its actual implementation. Thus multiple *Concrete Builders* can create particular implementations to one and the same *Builder* interface. This can be used in order to ensure two different implemented programs follow an equal construction sequence by using the same *Builder* interface.

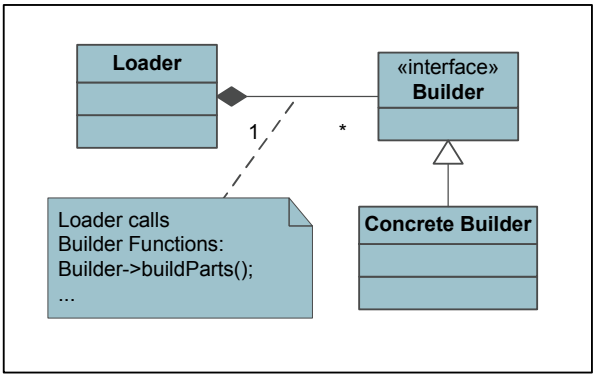


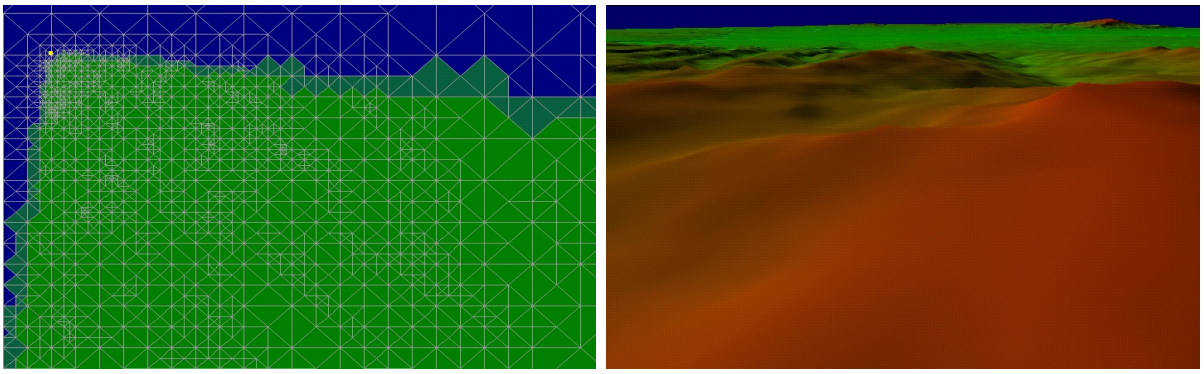
Figure 2.1: Builder Design Pattern

2.1.2 Terrain Rendering

Terrain rendering is a difficult problem for applications requiring accurate representations of large terrain datasets at high frame rates, such as flight or car simulators. On current graphics hardware, the problem is to maintain dynamic, view-dependent triangle

meshes and texture maps that produce good images at the required frame rate. Since the visualization of 3D terrain is possible in computer hardware, the wish to enlarge and accelerate the viewable area led to many researches in this topic. Heckbert and Garland [14] presented their general treatment of multiresolution LOD control in 1994, which was followed by a period of active research in terrain rendering.

One of the most recognized and successful algorithms was published by *Mark Duchaineau et al.* In their method *Real-time Optimally Adapting Meshes (ROAM)* [18] they are presenting an algorithm for constructing triangle meshes that optimizes flexible view-dependent error metrics, produces guaranteed error bounds, achieves specified triangle counts directly and uses frame-to-frame coherence to operate at high frame rates for thousands of triangles per frame. Following researches from Jonathan Blow [5] and Andreas Ögren [13] were based on the achievements of ROAM in modifying the number of triangles of a terrain mesh during runtime while improving the method in some details.

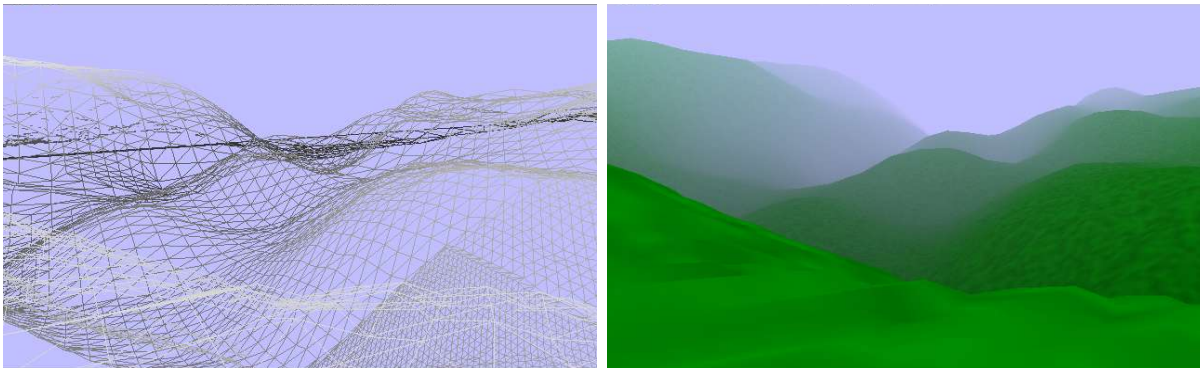


(a) Algorithm [32] (b) Rendered Terrain [32]

Figure 2.2: ROAM Method

Willem H. de Boer [7] presented a simple method for *Fast Terrain Rendering Using Geometrical MipMapping*. Different to his predecessors he is not concerned with optimizing the number of rendered triangles through complex algorithms during runtime, but optimizing the terrain meshes for the high parallelism of the meanwhile improved graphics hardware. Through saving a lot of calculation speed by not changing the actual terrain mesh, this terrain rendering method concedes the GPU with more time for rendering. Before the rendering process is started the algorithm divides the terrain into tiles, constructs different levels of detail for the meshes and at runtime changes the detail level dependent on the distance to the point of view.

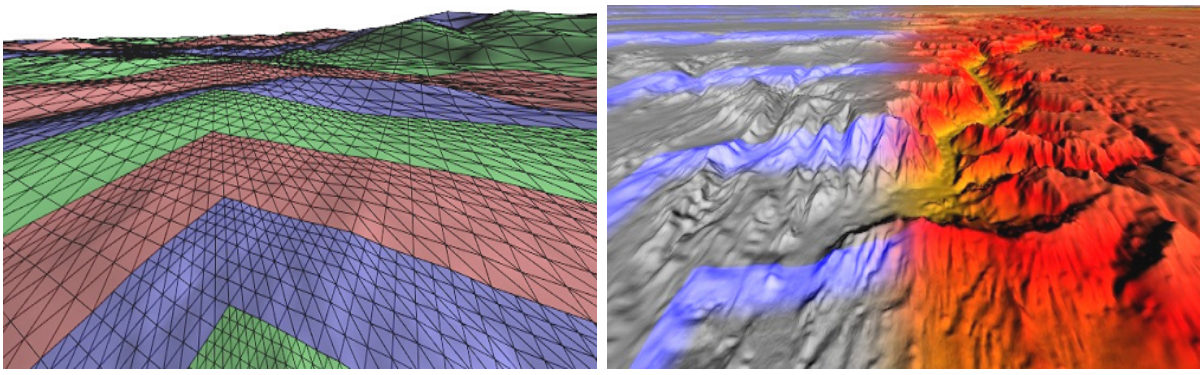
A totally different idea is expressed by Frank Losasso and Hugues Hoppe [17] in using a generated mesh gliding over an invisible terrain data skeleton while updating its vertices from arrays every time shifting to a new position. The so called *Geometry Clipmaps* method is very simple to implement and produces a good rendering output as the area



(a) Wireframe Terrain [16] (b) Rendered Terrain [16]

Figure 2.3: Geometrical MipMapping

next to the viewpoint is always highly detailed and the transitions to lower detail regions are never reachable.



(a) Illustration [17] (b) View of Dataset [17]

Figure 2.4: Geometry Clipmap

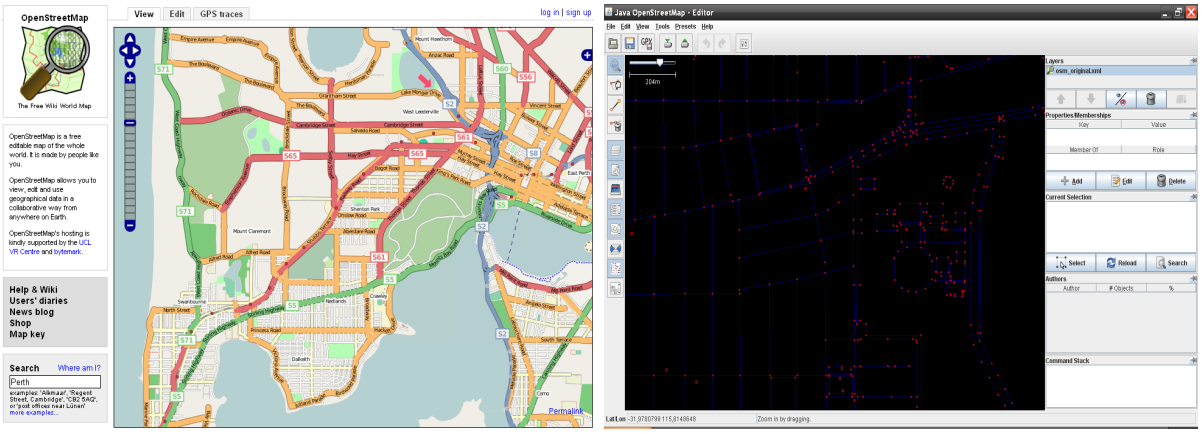
2.1.3 Geometric Data Systems

In the last couple of years many geometric data systems like *Google Maps* arose and offer now everyone free rendered road maps together with satellite data. These services could be very useful for AutoSim as it requires a lot of street, height and satellite image data. Unfortunately the data these services use for creating their maps is usually not available for free or sometimes cannot be retrieved at all.

Luckily in 2004 the *OpenStreetMap* project was founded, which collects and publishes user created data similar to the web encyclopedia *Wikipedia* and makes it available for

download under a license that allows it to be used in other open source projects. Everybody can take part in the creation and modification of road data by simply using GPS devices to create new maps or edit the existing maps. Additionally to the data *OpenStreetMap* provides rendering map images and route planning that can be obtained directly from their website [23].

In the meantime *OpenStreetMap* has already formed a big community and the users created some good tools for editing the road data. Above all a free Java editor, called *JOSM* has been very useful during AutoSim development. *JOSM* can directly download data from the *OSM* server and modify it. Changes on the street web are simply done by dragging the street nodes through the 2D view to the desired new position (Figure 2.5(b)). If two users upload the same modifications of a map to the server, those will be saved in the *OpenStreetMap* database and published on the server. A description of the data structure is available from within the good *OSM* documentation [23].



(a) The Website [23] (b) JOSM [23]

Figure 2.5: OpenStreetMap

2.2 Simulators

Many 3D Driving Simulators have been developed in the last couple of years. The following section presents some of the most interesting for AutoSim.

2.2.1 RARS

RARS is the Robot Auto Racing Simulation, a competition for programmers and an ongoing challenge for practitioners of Artificial Intelligence and real-time adaptive optimal control. It consists of a simulation of the physics of cars racing on a track, a graphic

display of the race, and a separate control program (robot *driver*) for each car. RARS was published in 1995, the year also the first RARS race was announced and performed. From 1995 to 2003 many races and even complete racing seasons were carried out by the RARS team. The robot programs of the participants were sent to a local machine, simulated and published on the website [26].

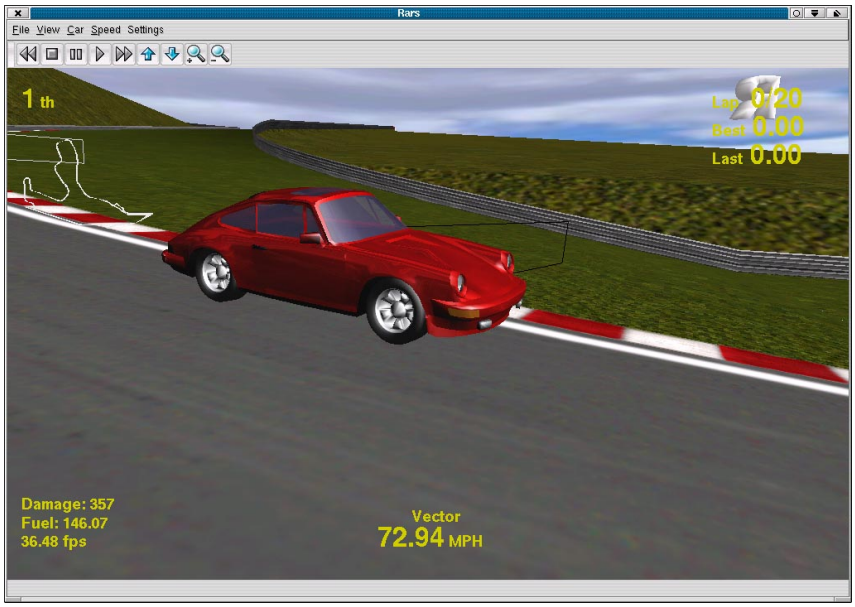


Figure 2.6: RARS Screenshot [26]

Even if RARS has 3D graphics in the meantime, the graphic was first developed to visualize a 2D racing track system. On that account it is still limited by the old 2D system as the project was not completely restructured into a 3D system. The Physics system is simple as well and thus gives the user only a few possibilities of obtaining and processing data inside the robot programs. The simulator's capabilities of providing the user with robot data are more related to writing AI programs for computer games than for a real world. Almost no real world sensors exist.

2.2.2 TORCS

TORCS, The Open Racing Car Simulator is a racing simulator which allows users to drive races against computer controlled opponents and to develop own computer-controlled robots. The concept of TORCS is derived from RARS [26], but allows the user now to control one of the robots by an input device like keyboard, mouse or steering wheel. The Graphics system is in good shape and is able to visualize lighting, smoke, skidmarks and glowing brake disks. In addition to a common physics model the simulator features a simple damage model, collisions and copious car properties. The gameplay allows different types of races from the simple practice session up to a whole championship.

Just as RARS, TORCS has a racing board where race competitors meet, upload their robot programs and view racing results. The software uses cross-platform libraries like OpenGL, Mesa 3D and OpenGL Utility Toolkit, to be able to run on many platforms (e.g. Linux, PowerPC Architectures, FreeBSD, Microsoft Windows).

Figure 2.7: TORCS Screenshot [31]

Similar to RARS, TORCS is intended for driving on a race course, rather than in a city environment and in the light of this the physics model does not behave like reality. Furthermore TORCS also does not have network capability, which means challenges between robots have to be simulated on one local machine.

2.2.3 Racer

Racer is a free cross-platform car simulation project (for non-commercial use), using professional car physics to achieve a realistic feeling and an excellent render engine for graphical realism [25]. Racer's graphics engine is based on OpenGL and capable of displaying effects like smoke, skid marks, sparks, sun, flares and vertex-color lit tracks. The physics system is able to simulate a large range of different vehicles and these vehicles, the tracks and the vehicle AI can be self-created by the user. Customizing and modifying is well documented in the Racer project, which also provides a lot of tools and editors. The package is available for Windows, Linux and Mac OS X platforms.

Unfortunately Racer is not open source and although developers can obtain the source code from the author, they are not allowed to publish it in own projects. Furthermore

Figure 2.8: Racer Screenshot [25]

Racer aims mainly at arcade game fans and people testing race car physics rather than simulating a whole city environment including sensors and actuators. Also the program is not networked which limits the whole simulation and control programs to a single machine.

2.2.4 SubSim

SubSim [1] is a software framework for simulating autonomous underwater vehicles. Without the need for a physical hardware it allows researchers to develop robotic software, controlling an autonomous vehicle in a virtual three dimensional underwater environment. The environment and the simulation can be customized through XML files and the application's programming interface is compatible with *C* and *C++*. The user can extend the simulator through a *C++* plugin model and furthermore use the sensor outputs for interconnection with an *EyebotController* (hardware device that controls motors and servos [8]). Application design, controller tuning, mission simulation, and fault-tolerance can all be tested with this simulator.

The Subsim framework features a realistic underwater simulation by using a physics engine for calculating the actions in the scene and by having an adjustable noise system for the sensors. Almost all realistic sensors for submarines can be obtained of the simulator and additional debug information helps the researcher during development.
The downside to SubSim is that it is only capable of simulating a single robot on a single machine. Also it is not networked and the visualization features are rather poor due to the underlying graphics engine that only can load a simulation scenario out of a static 3D file.

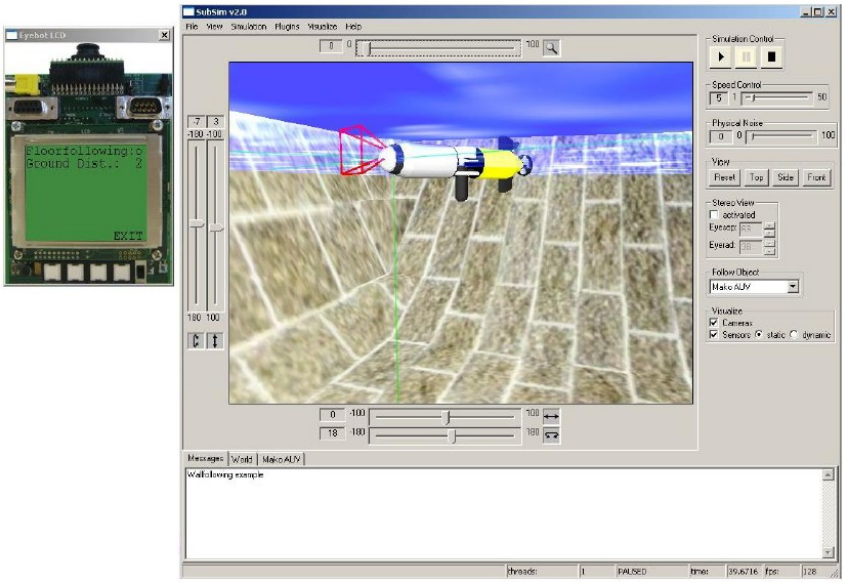


Figure 2.9: The Subsim AUV Simulator [1]

3 AutoSim Framework

Regarding the driving simulators section in the previous chapter, there are numerous of good driving simulators already in existence. Some of these driving simulators are really good for testing artificial intelligence used in computer games, but among them there is not a single one useful for simulating real world problems faced by autonomous driving vehicles. AutoSim has the intention to fill this gap and help people who are researching in the driverless vehicle field (1). But why not use one of the existing simulators and extend it with real world features? Because many of the existing simulators have still a software structure that is not up to date and especially not the way AutoSim is meant to be designed. The networking concept has been a major part of the AutoSim design from the beginning of development. The simulation and visualization tasks are completely decoupled and symbolize the server-client concept. The physics engine can be easily replaced due to the usage of a Physic Abstraction Layer [6] and the visualization also uses up to date open source graphics engine technology.

Most of the existing simulators and further 3D applications are written in the *C++* programming language. *C++* combines low-level and high-level language features to form a fast and extensible programming language that is easy to learn from a general or even better *C* programming background [10]. For this reason the language was also chosen for AutoSim as the framework wants to use all advantages of object oriented programming by still focusing on a fast and resource saving underlying code system. Furthermore most of the network, graphic or physic open source libraries have their priority in the *C++* version or even only exist in *C++*.

This chapter will describe the AutoSim framework in detail by showing a general presentation of its functionality and pointing out the most important features. First a brief overview and explanation of the used libraries is given, followed by a survey of the software design.

3.1 Used Libraries

Irrlicht: The Irrlicht Engine [2] is an open source high performance realtime 3D engine written and usable in *C++* and also available for .NET languages. It is completely cross-platform, using Direct3D, OpenGL and its own software renderer, and has all of the

state-of-the-art features which can be found in commercial 3d engines. Irrlicht is platform independent and can be currently used on Windows, Linux, MacOS and Solaris/SPARC. Compared with other open source engines it contains a big support of mesh and picture file types as well as a good documentation and community help.

Qt: Qt (pronounced *cute*) is a cross-platform application development framework, widely used for the development of GUI programs (in which case it is known as a Widget toolkit), and is also used for developing non-GUI programs such as console tools and servers. Qt is most notably used in KDE, the web browser Opera, Google Earth, Skype, Qtopia and OPIE. It is produced by the Norwegian company Trolltech.
Qt uses *C++* with several non-standard extensions implemented by an additional pre-processor that generates standard *C++* code before compilation. Qt can also be used in several other programming languages; bindings exist for Ada, *C#*, Java, Pascal, Perl, PHP (PHP-Qt), Ruby (RubyQt), and Python (PyQt). It runs on all major platforms, and has extensive internationalization support. Non-GUI features include SQL database access, XML parsing, thread management, network support and a unified cross-platform API for file handling.

RakNet: Raknet is a cross platform *C++* network library designed to allow programmers to add response time-critical network capabilities to their applications. It is mostly used for games, but is application independent. The major advantages of this package comparing to other network libraries that it is easy to use, well documented, open source and extremely fast which is absolutely essential for networked real time simulations. It also adds very little overhead to the packages that are being sent.

PAL: The Physics Abstraction Layer (PAL) provides a unified interface to a number of different physics engines. This enables the use of multiple physics engines within one application. It is not just a simple physics wrapper, but provides an extensible plug-in architecture for the physics system, as well as extended functionality for common simulation components. PAL also has an extensive set of common features such as simulating different devices or loading physics configurations from XML, COLLADA and Scythe files. PAL supports a large number of physics engines among others Bullet, JigLib, Newton, ODE, Tokamak, TrueAxis and OpenTissue and also features an extensive testing and benchmark suit for evaluating and visualizing dynamic simulation systems.

TinyXML: TinyXML [30] is a very small and simple open source XML parser for the *C++* language. It can be easily integrated into programs to parse an XML document and build a Document Object Model (DOM) from it. The DOM can then can be read, modified, and saved. It also allows the users to construct their own XML documents with *C++* objects and write these to the harddisk or another output stream. As the name already says, it is tiny and does not support Document Type Definition(DTD) or eXtensible Stylesheet Language(XSL) and in terms of encodings, it only handles files

using UTF-8 or an unspecified form of ASCII not entirely dissimilar from Latin-1.

3.2 Framework Architecture

The AutoSim framework is an open source 3D robot simulation software on the basis of platform independent open source software libraries 3.1 and is formed by four programs:

- **AutoSimServer**: Program that is running the simulation by creating sensor data through its included physic model and transfers it over the network.
- **AutoSimClient**: Network client that processes and visualizes the received data through the graphic engine.
- **UserProgram**: Instance controlling a robot by using functions of the UserProgramAPI and ClientUserProgramAPI.
- **OsmManipulator**: Tool changing the OpenStreetMap data by automatically adding houses along the streets.

At runtime the AutoSimServer and AutoSimClient are the programs forming the simulator and the UserProgram controls the robots. As the OsmManipulator is only changing the OSM data and preparing it for the creation process it does not have any functionality within the simulation part. Figure 3.1 gives a brief overview of the framework throughout a running simulation.

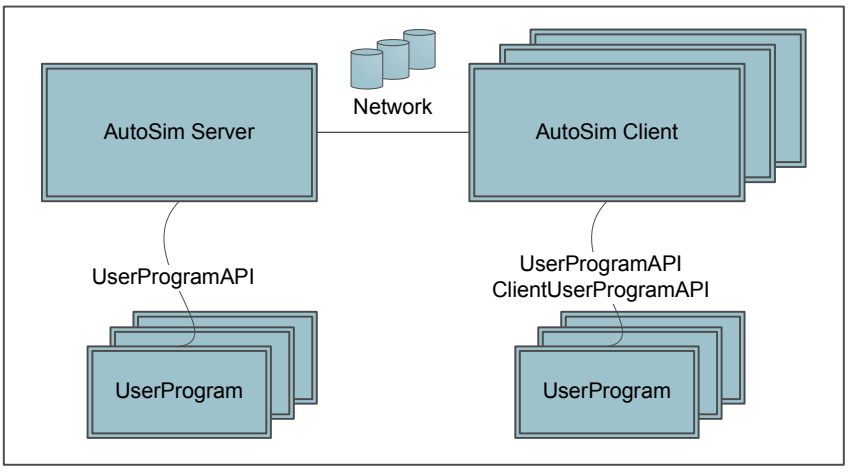


Figure 3.1: AutoSim during runtime

Multiple clients can connect to the server through the network interface and receive, process and visualize the broadcasted data. Instances of the UserProgram are loaded as external DLL programs on both the server and the client side and control the robots

through the UserProgramAPI respectively ClientUserProgramAPI. The two APIs differ in the way they send and receive data. As the server-side UserProgramAPI can directly ask for information from the data creating server the API on the client-side has to transfer the data over the network.

3.3 Program Description

The following four sections give a brief overview of the four AutoSim programs.

3.3.1 AutoSimServer

The AutoSimServer is responsible for simulating all interaction between the simulation objects in the underlying physics engine and preparing the data for the UserProgram and the network transfer to the AutoSimClient respectively. It provides a user friendly and easy to use graphical interface for the most common operations on the server such as configuration file selection and tuning of the integration step size for the physics engine. Figure A.1 shows the AutoSimServer after a simulation has been loaded.

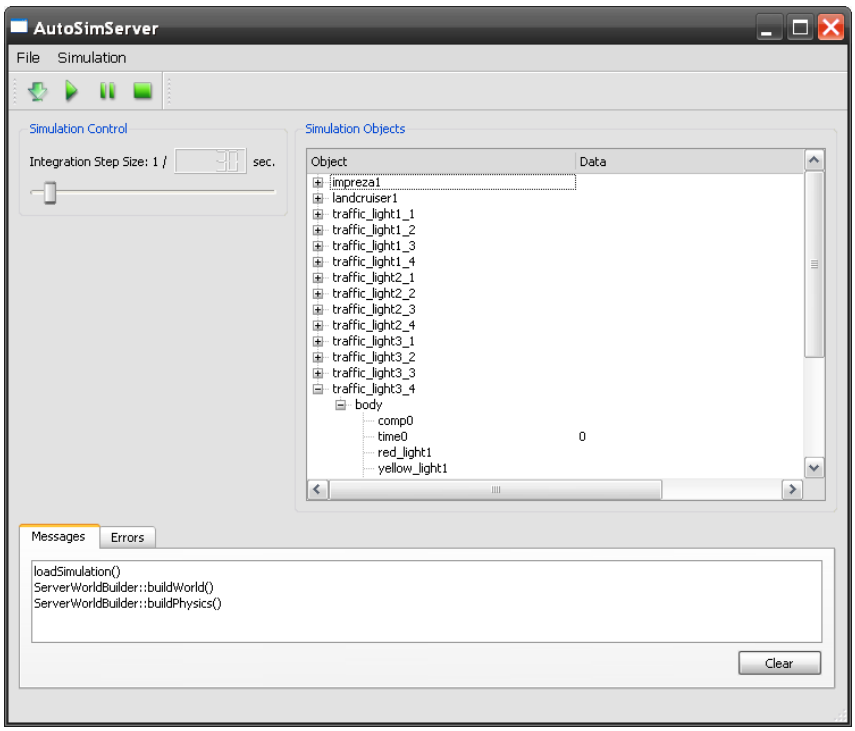


Figure 3.2: Graphical User Interface of the AutoSim Server

On top there are buttons to load, start, pause and stop the simulation process. To the right all loaded simulation objects such as the robots are displayed including all their

parts and devices which is especially interesting for debugging purposes since it shows all parts loaded from the XML files and often reveals problems caused by corrupted configuration files.

On the bottom of the window two boxes display all messages and warnings generated during the loading process and runtime of the simulation. It can easily be hidden by pulling on the bottom line of the Simulation Objects box to gain more space. The server also runs all common user programs that control the robots representing dummy traffic, pedestrians and traffic signals etc. Contrary to the client the server's UserPrograms are set in the world configuration file instead of the GUI.

3.3.2 AutoSimClient

AutoSimClient provides the user with an easy to use graphical interface to load, run and debug the client part of the networked simulation. Figure 3.3.2 shows the GUI of AutoSimClient created with QT.

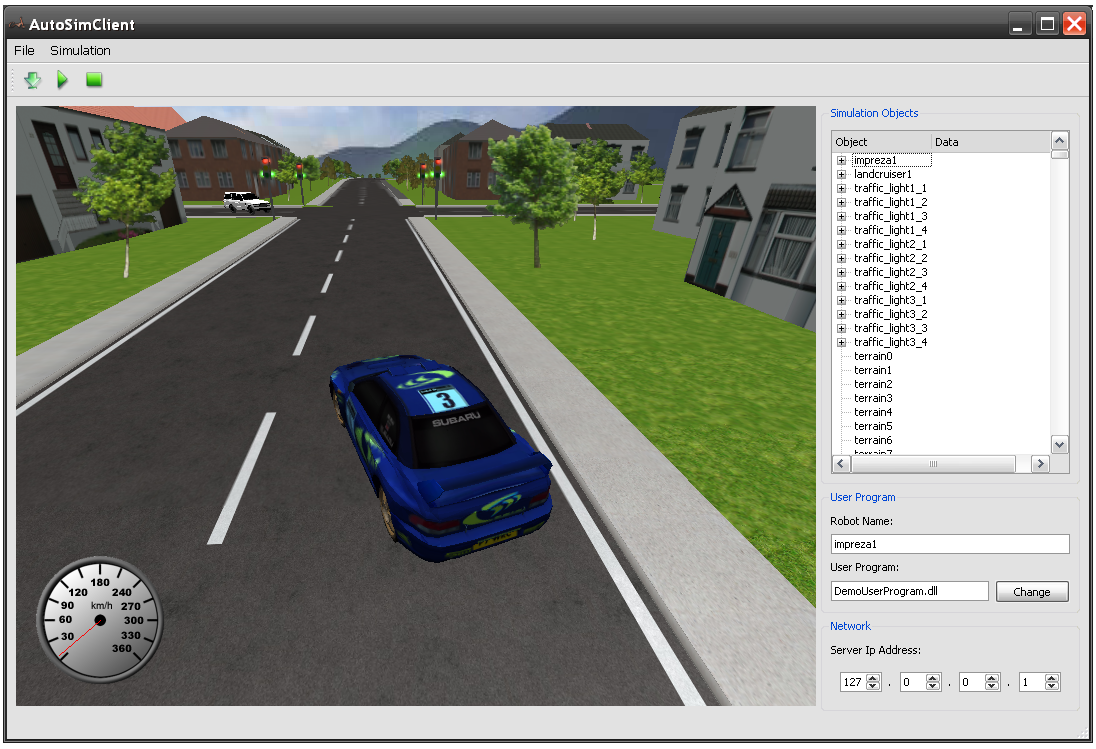


Figure 3.3: AutoSimClient Graphical User Interface

On the top of the GUI the user can access a menubar to select the world file or UserProgram the client should load and run. Supplementary UserProgram settings can be done in the input fields of the *UserProgram* box. The network box to the lowest on the

right contains an IP address setting for the network. By default the IP address is set to localhost to let the client try to connect to a server running on the same computer. A press on the load button to the top left loads the graphic world into the AutoSimClient program. In the *Simulation Objects* box of the interface appears a scrollable tree view of the simulation objects when loading is completed. This tree contains debug information of the objects like position and rotation and is updated during runtime. Pressing the play button next to the load button connects the network client to the server, starts the rendering process and executes the loaded UserProgram with the robot in the UserProgram's properties box. Now the graphics window shows the rendered graphic scene. The scene is always rendered when new data arrives from the server. Inversely that means the graphics window will not be updated if no data from the server has arrived.

3.3.3 The UserProgram

UserPrograms are representing the control instance of every Robot inside the simulation. They consist of a more or less complex *C/C++* programs which are being compiled as a DLL and dynamically loaded by either the server or the client. It can access all devices of the robot it belongs to through an API to read data from sensors, process them and control the actuators. Useful to for example obstacle avoiding programs might be that UserPrograms are not limited to control a single robot, but can access devices on every remote robot.

The UserProgram loaded by the AutoSimClient can be different to the UserProgram running on the AutoSimServer as it can access further client information through the ClientUserProgramAPI. The ClientUserProgramAPI expands the UserProgram possibilities by adding functions like getting a virtual camera image that are exclusively available on the client, because the server does not store any simulation data and the visualization process is only done on the client. UserPrograms only using the UserProgramAPI can be executed by either the AutoSimClient or the AutoSimServer.

Figure A.3 shows a FLTK based graphical user interface of a UserProgram capable of turning the rear and head lights on and obtaining images from various virtual cameras to display them in the upper part of the window.

3.3.4 OsmManipulator

Figure 3.5 shows the user interface of the OsmManipulator program. The program calculates positions for new houses in a line along the roads and adds these houses as new OSM nodes into the OSM file by consulting a house file list for model file paths and other

Figure 3.4: User Program

necessary information.

Inside the main window the user can set up the manipulator and launch the application by pressing the *Generate OSM File* button. The following XML files have to be specified:

- the original OSM file downloaded from the OSM server
- the target to store the new created file
- a map setup file where the sizes of all road types are stored
- a house file that includes the data of the houses (e.g. graphics model file path and type, house size)

Finally the distance of the house centers to the street and the distance between houses in a row of houses are changeable in two further input fields. After selecting all settings as well as input and output files, the generation process is started by pressing the *Generate OSM File* button and finishes with a message below the button.

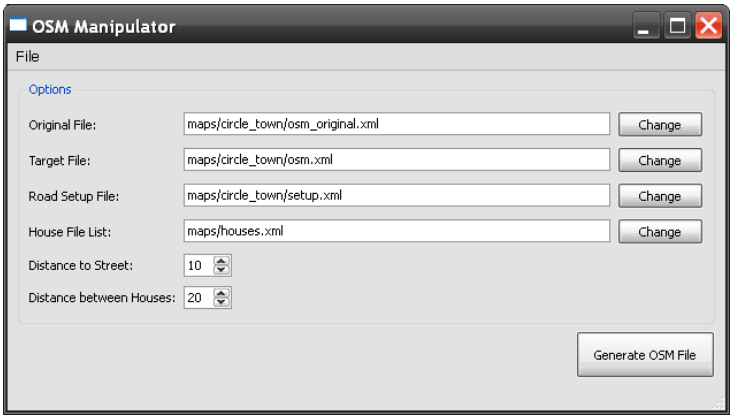


Figure 3.5: Main Window OsmManipulator

3.4 Client Software Design

Software design is a process of problem-solving and planning for a software solution [38]. After the purpose and specifications of software are determined, software developers will design a plan for a solution. A good software design prevents software from having errors before the actual implementation process has started.

As the thesis was concentrated on the client part of the networked simulator, this section will mainly focus on the design of the AutoSimClient. The software design of the OsmManipulator and the UserProgram is not important for understanding the simulator's functionality, while the AutoSimServer is explained in detail in Torsten Sommer's thesis *Physics for a 3D driving simulator* [28]. The UML diagram of Figure 3.6 gives an overview of the most important instances forming the AutoSimClient at runtime.

The main class of AutoSimClient is the controller named *ClientController*. As its name implies, it controls all the parts, can start and stop threads and finally shut down the program. If one of the client threads wants to talk to another thread, it has to do this by calling controller functions. Like some other client classes the *ClientController* uses the Singleton design pattern by inheriting the Singleton template class (section 2.1.1). The Singleton ensures the controller only exists once and is accessible from out of all other threads through a globally accessible pointer.

The first task for the controller after starting the AutoSimClient program is to load and build the simulation world. Loading the world by clicking on the load button in the GUI calls a controller function for creating instances of the *WorldLoader* and *ClientWorldBuilder* classes. Those create in turn all the particular builders necessary for constructing the world and this process is explained in detail in *World and Robot Creation* (chapter 4). Once the world exists the client can establish the network connection, visualize the

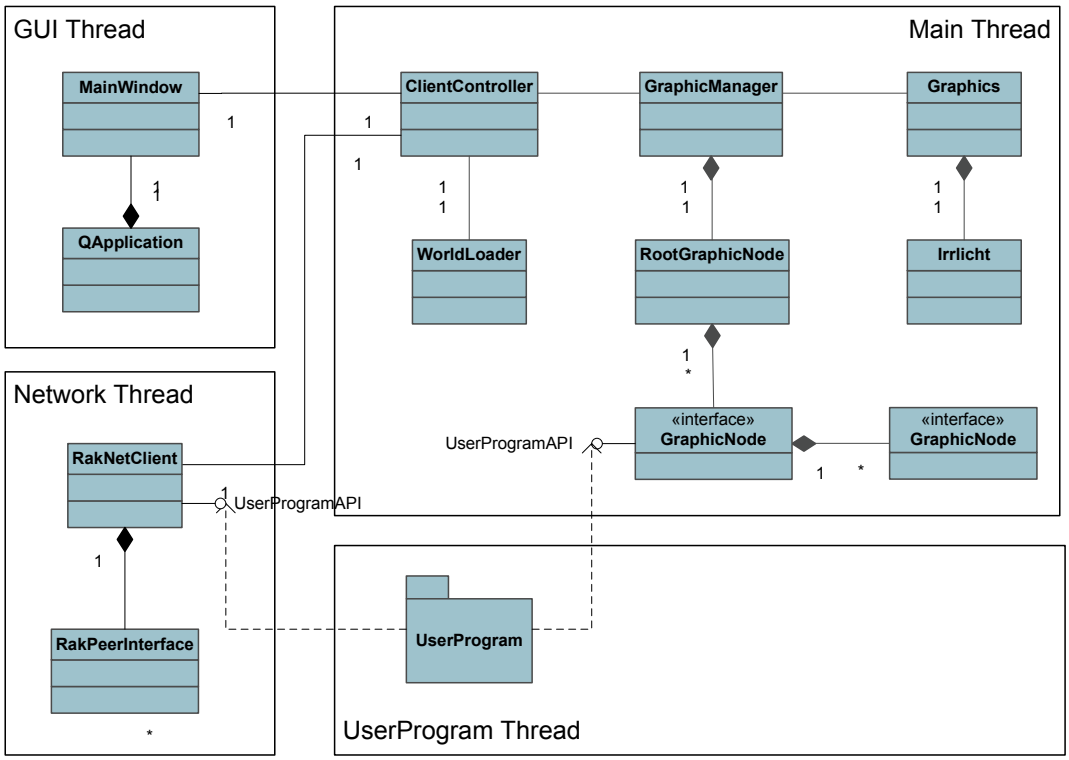


Figure 3.6: AutoSimClient Software Design

received data and execute the UserProgram. Pressing the run button in the GUI induces this sequence in the mentioned order. For specifying the current phase of loading and constructing, the *ClientController* possesses a member variable that indicates the status of the program. Before the controller proceeds with a new loading task it waits for the previous process to finish, signalized by a change of the status variable. If for example the network thread was started the controller has to wait until the network was correctly initialized by connecting to the server.

The graphic rendering process during runtime is mainly executed by the two *Singleton* derived classes *GraphicManager* and *Graphics*. While the *GraphicManager's* task is to manage the data tree and provide a pointer and functions to access it, the *Graphics* singleton performs the actual rendering process through its *Irrlicht Engine* [2] member. Therefore it starts the rendering every time new data arrives from the server, which is explained in detail by section 7.2.

The AutoSim simulation world consists of many different objects which share features like name, position and rotation in common. Apparently every kind of object is usually visualized in a different way and thus needs to execute particular operations for updating its graphical representation when the simulation data changes. Therefore the object representing classes are all deriving from an abstract *GraphicNode* class.

In *C++* a class is abstract and thus called an interface when it has at least one pure virtual function. A pure virtual function is declared but not implemented in the interface class and must be implemented by every deriving class, or the inherited class becomes abstract as well [10].

The abstract *GraphicNode* class represents an interface for all objects in the graphics world. Every particular graphic node must always implement the *update* function, which will be called by the graphic manager every time a new data package from the network has arrived. As the update function is declared in the interface it can be called from the graphic manager equally on every graphic node without knowing which operations are executed by the node.

The *GraphicManager* class has also functions to create all different kinds of graphic nodes. For this reason the user does not have to know anything about the underlying graphics engine, as the graphic manager takes over the task to create the engine's instances for the graphical representation. All created graphic nodes inherit a software tree structure that is also used by the server and provided by the *SerializableTreeNode* class [28]. Furthermore they obtain necessary serialize and deserialize functions for the network communication from that class. Besides the advantages for *World Creation* (section 4.1), the tree model structure of the object data also accelerates the update process on the nodes. To update the whole data tree the graphic manager simply calls a function on the tree's root node that updates all its children and these children perform an update on all of their children again. The root node is represented by the *RootGraphicNode* which derives from the *SerializableTree* and thus has extended functionality compared to a normal *SerializableTreeNode* [28]. Every graphic node has a boolean variable specifying if it is currently visible or not. If a robot for example is not visible in the graphic world because its position is too far away from the camera all robot connected devices like wheels and lights also do not have to be updated (section 7.3). Robot Creation (section 4.2) makes all *device* graphic nodes children of *part* graphic nodes and these again children of the parent *robot* graphic node. If now for example the robot's chassis is invisible all the children like wheels and lights will not be updated anymore. This method saves resources by not calling superfluous updates.

As already mentioned before, the graphic manager always calls an update on the complete graphic node tree when new data has arrived from the server and only when the update is completed a new render process is started. The rendering can be done on either the screen, a texture or on both of them. Render to a texture is provided to let the user do image processing on the simulation's graphic scene. Section 7.2 elaborates the rendering in chapter 7.

4 World and Robot Creation

The client software design section (3.4) of the previous chapter was mainly concerned with explaining the AutoSimClient's software design at runtime. Additional to the runtime actions this thesis took a big effort on the construction of the simulation world and its objects. Therefore this chapter gives an overview of the AutoSimClient's world creation concept and goes into further detail by describing the robot building process. As the terrain creation is a large topic it is handled separately in chapter 5 although it is part of the world creation.

4.1 World Creation

For being able of transferring the highest possible amount of object positions and other object information from the server to the client, AutoSim uses a networking concept that tries to avoid transmitting unnecessary information. However, the AutoSimClient needs to know the object belonging to the data package received from the server. Sending name information would solve this problem but increase the transmitted data a lot due to the large byte size of a string. AutoSim uses a different concept of synchronizing data. The previous chapter (3) introduced the graphic nodes and their inherited serializable tree concept. As its name implies the serializable tree representing class *SerializableTreeNode* is able to serialize its attributes and also deserialize the data again into a tree structure [28]. If the data tree structures on server and client are kept equal the *SerializableTreeNode* can directly inscribe the received data from the server into the client's tree. If a graphic node like for example a virtual camera does not need any data to receive from the server, its appearance in the client's graphic node tree does not effect the deserialization. The data arrays of every data transferring device are specified in an XML file, telling the server how much data to transfer for every device and helping the client to construct the data arrays for every graphic node derived class.

To meet the requirements of equally constructed data trees, the client and the server share their concept for construction and use the same files for reading out the information. The UML diagram of Figure 4.1 visualizes the software structure of the AutoSimClient's world creation, excluding classes not necessary for understanding the process.

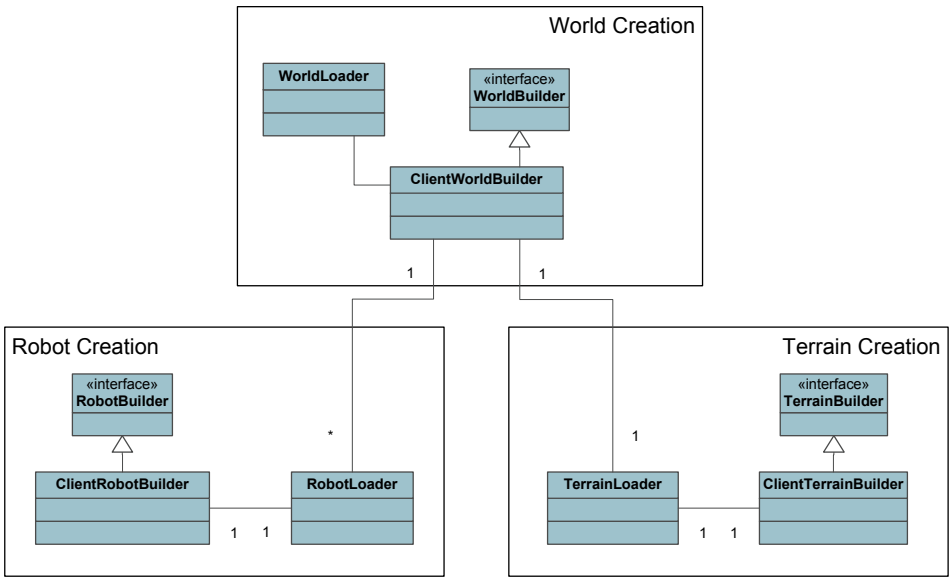


Figure 4.1: World Creation

The AutoSim framework uses the builder concept introduced in section 2.1.1 for world construction. Both the client and the server implement the virtual functions of the shared *Builder* interfaces in order to meet their particular construction requirements. Their shared *Loader* classes call the *Builder* functions without knowing which program they belong to. Sharing the *Loader* classes also ensure the data is loaded equally into both programs. The thesis will now enumerate the single steps of world loading in the AutoSimClient, as this is probably the best way for understanding the usage of the builder pattern in AutoSim.

1. create a new *WorldLoader* instance.
2. call *WorldLoader->loadWorld()* and pass a new *ClientWorldBuilder* object as argument.
3. function *loadWorld()* loads all settings of the world file into a data structure.
4. *loadWorld()* calls the *WorldBuilder* functions *buildWorld()*, *buildGraphics()*, *buildPhysics()*, *buildTerrain()* and *buildObjects()* of the *ClientWorldBuilder* implementation and passes therefore needed data to the builder.

The five *WorldLoader* functions mentioned in item 4 are declared as pure virtual within the *WorldBuilder* interface and must be implemented by the deriving *ClientWorldBuilder* and *ServerWorldBuilder* classes. Similar to that, the *TerrainBuilder* and *RobotBuilder* contain pure virtual functions to be implemented by their deriving server and client classes. In addition to interface declarations the builder classes may also contain build functions for operating tasks that do not distinguish between server and client. Applying height data to constructed road meshes is an example for this (section 5.3).

To complete the description of the client's world creation procedure the five main construction functions will be explained now:

- **buildWorld() and buildPhysics()**: the AutoSimClient does nothing since these functions are just needed on the AutoSimServer.
- **buildGraphics()**: initializes the *Graphics* instance with parameters for starting *Irrlicht*, creating a new texture as render target and setting the current mode for visualizing debug information.
- **buildTerrain()**: loads and builds the terrain by creating a new *TerrainLoader* with a new *ClientTerrainBuilder* as argument. The terrain construction is explained in chapter 5.
- **buildObjects()**: parses through all robots and creates a new *ClientRobotBuilder* for each of them to be loaded by the *RobotLoader* 4.2.

4.2 Robot Creation

For every robot listed in the world file AutoSimClient creates a new *ClientRobotBuilder* responsible for building exactly one robot, a new *RobotLoader* responsible for loading the robot's XML file and finally starts a new robot building process. As implied by the names of the classes the builder pattern is used again. Because this concept was already explained in world creation (section 4.1) and the robot's loading concept is similar to the one of the world, the next parts will mainly focus on the characteristic client tasks in the *ClientRobotBuilder*.

Every robot consists of parts and devices. Parts are always physic boxes with a size, a mass and a visualizing graphic model. A car has for example a chassis part and could carry a trailer as a second part. All devices have to be connected to one part of the robot, meaning a robot cannot exist without at least one part. As the parts and also the devices with a graphical representation are graphic nodes the *ClientRobotBuilder* has to construct these nodes and add them into the tree structure. *DummyGraphicNode*'s do not have any functions for visualization during runtime, but they can establish an easier understandable tree structure. One of them is constructed by every *ClientRobotBuilder*, as a general graphic node for the robot to keep all robot parts as children of this node in the tree. Figure 4.2 provides a graphical explanation of the robot tree.

All graphic nodes are created by the *GraphicManager*, as it simplifies the construction process by knowing how to construct every individual graphic node. Therefore it has functions for creating every type of graphic node and a pointer to a parent node specifies the part of the tree the node will be added. The robot builder creates a list of all the parts of a robot, as every device has to be added to its specified part. Therefore the parts

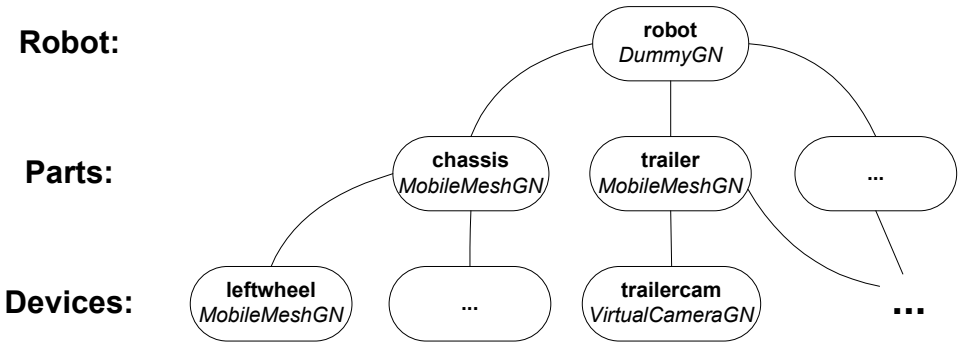


Figure 4.2: Robot Creation

of the robot are processed before the devices.

The following enumeration lists all graphic nodes the graphic manager can construct.

- **BillboardGraphicNode:** represents a static object that only consists of a texture which always looks towards the camera
- **DebugBoxGraphicNode:** visualizes a physic box in the graphic world; used for debugging
- **DummyGraphicNode:** *GraphicNode* used for keeping the tree structure; does not have any functionality
- **HeadLightDeviceGraphicNode:** displays a *HeadLightDevice*: the light cone of the head lights
- **LightDeviceGraphicNode:** displays a *LightDevice*: the light particles at the car
- **MeshGraphicNode:** visualizes any 3D graphic model
- **MobileMeshGraphicNode:** used for 3D models receiving position and rotation updates from the server
- **TerrainMeshGraphicNode:** displays a terrain tile
- **ThirdPersonCameraGraphicNode:** camera flying behind a graphic node; does not rotate with the node
- **VelocimeterGraphicNode:** introduces the possibility to display a speedometer with the current speed for every Velocimeter device.
- **VirtualCameraGraphicNode:** fixed camera attached to a graphic node; rotating with the node

The different types of graphic nodes mainly distinguish in the attributes they possess and the way they react on a data tree update. Some of the most interesting update methods are explained in chapter 7.

5 Terrain Modeling

Worlds of testing environments for autonomous driving vehicles should always be big enough to confront the vehicles with different unforeseen events. Rather than just going around a block of houses, a world consisting of a whole street web is built by the AutoSim simulator. Unfortunately the physics engine cannot simulate an unlimited world size and the polygons drawn by the graphics card are limited as well. While the limitation by the physics is still tolerable, as the loadable scene is highly sufficient for a testing environment, the amount of drawable polygons for the graphic world is far from being satisfactory. For that reason this chapter is primarily concerned with creating a terrain meeting the low polygon requirements.

This chapter starts by explaining the theory behind *Bilinear Interpolation* and *Shader* programming and continues with the AutoSimClient's implementation of these methods in order to lift meshes, create a graphic terrain and create static objects.

5.1 Bilinear Interpolation

In mathematics, bilinear interpolation is an extension of linear interpolation for interpolating functions of two variables on a regular grid. The key idea is to perform linear interpolation first in one direction, and then in the other direction [34]. If the 2D coordinates of a desired value are more accurate than the resolution of the grid bilinear interpolation has to be done between the 4 surrounding values. Figure 5.1 shows a point P with the coordinates (x, y) whose location is between the existing data points with coordinates $Q_{11} = (x_1, y_1)$, $Q_{12} = (x_1, y_2)$, $Q_{21} = (x_2, y_1)$ and $Q_{22} = (x_2, y_2)$. The regular grid is shown by the dashed lines.

The first step is to obtain the values of the points $R_1 = (x, y_1)$ and $R_2 = (x, y_2)$ by linear interpolation in x-direction:

$$f(R_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \tag{5.1}$$

$$f(R_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \tag{5.2}$$

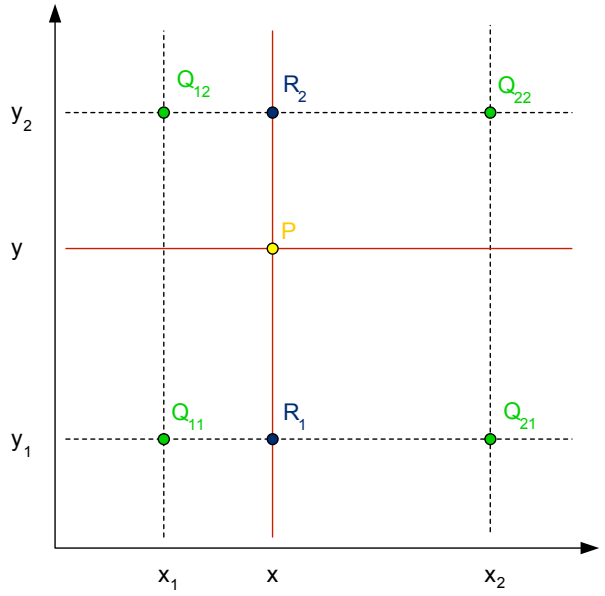


Figure 5.1: Bilinear Interpolation

$f(R_1)$ and $f(R_2)$ can now be used for performing linear interpolation in y-direction and thus obtain the value of P:

$$f(P) \approx \frac{y_2 - y}{y_2 - y_1} f(R_1) + \frac{y - y_1}{y_2 - y_1} f(R_2) \tag{5.3}$$

If distances between points are always 1 and (x_1, y_1) can be set in the origin $(0, 0)$, the equations above are reduced to:

$$f(P) \approx a_1 + a_2 x + a_3 y + a_4 xy \tag{5.4}$$

where

$a1 = Q_{11}$
$a2 = Q_{21} - Q_{11}$
$a3 = Q_{12} - Q_{11}$
$a4 = Q_{11} - Q_{21} - Q_{12} + Q_{22}$

and x and y are now distances relative to Q_{11}.

5.2 Shaders

CPUs usually have only one programmable processor. In contrast, GPUs have at least two programmable processors, the vertex processor and the fragment processor, plus other non-programmable hardware units. The processors and the non-programmable parts of the graphics hardware are linked through data flows. A common model of the GPU is illustrated by Figure 5.2 [21].

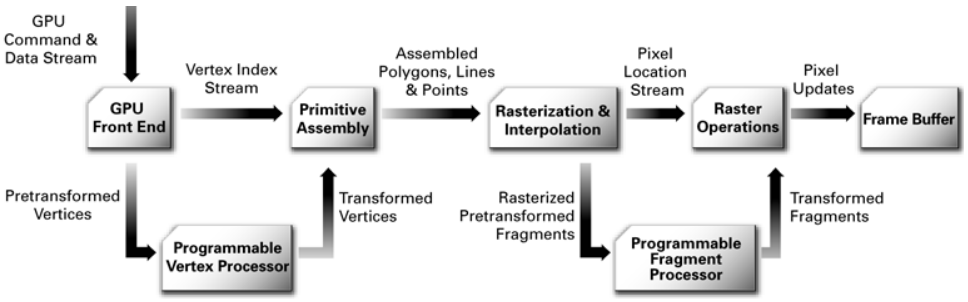


Figure 5.2: GPU data flow [21]

Shader languages allow us to write programs for both GPU processors. The vertex and fragment programs are called vertex and pixel shaders respectively. As their name implies, the vertex shaders are used for modifying the geometry data of the meshes, and pixel shaders perform operations on the pixels. The ongoing data flow on the graphics card requires a provident design of the GPU programs to prepare data for the following hardware units.

Today shaders can be written either in assembling language or in one of the several higher level languages. Unfortunately many shader languages are specified to either OpenGL or Direct3D and exist only for one of those two APIs. Therefore the graphic cards manufacturer *nVidia* established their shader language *Cg* to make it possible to write a program for both APIs. The *Cg* syntax refers to the currently most used *High Level Shading Language (HLSL)* by *Microsoft* in such a way that many HLSL programs are understandable to the *Cg* compiler without any change of the code. Nevertheless the shader programs are executed by one of the graphic APIs and have to meet the API's requirements for shader programming.

On the basis of the data flow model of the graphics card the programs are limited in the way they receive and process data. More precisely GPU processor programs in *HLSL* and *Cg* have two different ways of receiving input values:

- **Varying inputs** are used for data that is specified with each element of the stream of input data. For example, the varying inputs to a vertex program are the per-vertex values that are specified in vertex arrays. This can be model space positions, model space normal vectors or texture coordinates. Furthermore the number of

varying inputs is limited by the graphics card and because of that they have to use one of the predefined names (5.1) to specify their function and to keep the limitations of the variables.

- **Uniform inputs**, also named constants, are used for values that are specified separately from the main stream of input data, and do not change with each stream element. For example, the AutoSimClient vertex shader requires an array of height data to process a data update on every vertex of the mesh. The number of Uniform inputs is limited by the graphics card. Nowadays graphics cards have about 256 of the so called *VertexShaderConstants*. Constants have to be shader data types, but need not use predefined names like the varying inputs.

POSITION	**BLENDWEIGHT**
NORMAL	**TANGENT**
BINORMAL	**PSIZE**
BLENDINDICES	**TEXCOORD0 - TEXCOORD7**

Table 5.1: Example of predefined names for varying inputs

The *HLSL* and *Cg* shaders only know two data types, *float* and *int* and both are 32 Bit. To define a varying input the correct data type has to be combined with the chosen predefined type. For example a **POSITION** variable requires a 3 or 4 dimensional float vector. All the inputs have to be set as arguments of the shader's main function, as a shader is started by a main function call.

As the vertex shader is executed before the pixel shader it has to prepare the data for the pixel shader and also for the other following units. For example it has to transform its input **POSITION** by the graphic engine's world view matrix in order to get pixel coordinates. The data is passed to the GPU data flow by a structure of variables set as return value of the main function (Listing 5.1).

Listing 5.1: Output Structure

```
1 // Vertex shader output structure
2 struct VS_OUTPUT
3 {
4   float4 Position   : POSITION;   // vertex position
5   float4 Diffuse    : COLOR0;     // vertex diffuse color
6   float2 TexCoord   : TEXCOORD0;  // tex coords
7 };
```

Listing 5.2: Preparing data

```
1 VS_OUTPUT vertexMain(VS_INPUT IN)
2 {
3 VS_OUTPUT Output;
4
```

```
5 Output.Position = mul(IN.vPosition, mWorldViewProj);
6
7 return Output;
8 }
```

All CPU programs and programming languages support essentially the same set of basic capabilities. However, GPU programmability has not quite yet reached this same level of generality, as the vertex and fragment processors obtain more and more operation possibilities during the ongoing hardware development process. For example newer vertex shader versions since 3.0 also support texture lookups which was reserved for pixel shaders before. Regarding the functions a GPU program really needs, the chosen shader version should be the oldest possible to support a large range of graphic cards.

5.3 Applying Height Data to the World

The AutoSim framework is able to load real world height data into the simulated world. The height data is stored in a grayscale 8-bit image, named *HeightMap*. Independent from its pixel size, the *HeightMap* is always expanded to the size of the world. To 10 meter accurate height data should for example be stored in an image with a pixel size of a 10th of the world size.

Expanding height data to the world size is done via the bilinear interpolation (5.1) method. The *HeightMap* image and the world are interpreted as areas relative to their sizes. Meaning world coordinates are divided by the world size to get relative values from 0 to 1 and these are applied to the size of the *HeightMap* image. Bilinear interpolation can now calculate a height value for all world coordinates.

Some 3D Meshes, like for example roads, should lie exactly on the terrain surface and thus their meshes have to be transformed by using height data. This is done by iterating through all mesh vertices, take their x and z-coordinates as input points and add the interpolated height data value to their y-coordinate, which points upwards in the AutoSimClient coordinate system.

Objects like houses are lifted as a whole and not aligned along the terrain. For these kind of meshes only the y-coordinate of the model's position is lifted.

5.4 Graphics Terrain

During interactive visualizations, many types of geometric data compete for limited polygon budgets. Terrain remains one of the most challenging types because it is not naturally decomposed into parts whose complexity can be adjusted independently, and because the qualities required of a triangulation are view dependent.

Evaluating the existing methods for rendering large terrain areas of section 2.1.2 concludes in 3 good possibilities for constructing the AutoSimClient terrain. Regarding the simplicity of construction, the elegance of implementation, the time of publication and the promised rendering speed makes the *Geometric Clipmaps* more interesting than the *ROAM* and *Geometrical MipMapping* algorithms. The advantage of *Geometric Clipmaps* compared to ROAM is most of all the much simpler implementation and compared to *Geometrical MipMapping* the visualized area does not have to be divided. The client terrain rendering follows the idea of *Geometric Clipmaps* to have a mesh gliding over the data, but implements its own version by bringing the data update of the mesh to the graphics card and simplifying the algorithm by some not necessary features. The following section explains the way of constructing the mesh and the subsequent section *Terrain Vertex Shader* explains the operations processed during runtime on the graphics card.

5.4.1 Terrain Mesh

Geometric Clipmaps enhanced terrain requires the construction of a specific mesh used for gliding over the terrain data. To accelerate render speed this mesh should consist of different detailed parts, as the viewpoint will always be close to the mesh's center and never reaches the outer lying parts. In general there are two different ways of creating a mesh:

1. use a 3D modeling program and obtain an irregular mesh
2. program the mesh in order to be regular

The graphic terrain does not necessarily need a programmed mesh but there are advantages in building one. A regular mesh can for example consist of multiple tiles and the number of these tiles, and thus the range of vision, can be determined by the user later on. The *Irrlicht Engine's* mesh representation is also built on a limited number of *MeshBuffers* which simply contain a list of vertices, a list of indices and a material. As the engine cannot draw *MeshBuffers* with more than 65536 triangles they have to be divided into smaller parts in order to solve this limitation problem. Furthermore the level of detail of the individual tiles can be changed very easily in a regular model and different versions of details can be made available for various computer hardware configurations.

Figure 5.3 shows the mesh construction for one tile. The starting point is the tile rectangle with two outline defining vertices. This rectangle is divided into smaller rectangle parts in every construction step. A rectangle is represented by the AutoSim class *RectNode* and contains member functions for dividing itself and constructing new *RectNode* children. During every step the information of the parent rectangle is dropped in such a way that finally a list of small rectangles exists. The fan method for triangulation (7.1) is used in order to add new polygons to the *MeshBuffer* for every entry in the rectangle list.

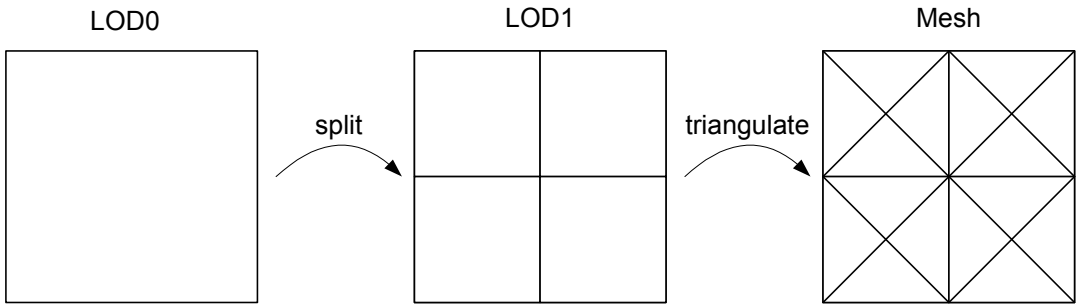


Figure 5.3: Tile Mesh Construction

The meshes of the tiles are now placed next to each other to form the terrain mesh. If tiles of different level of details are connected and the terrain is very hilly in the connecting area, gaps in the terrain, caused by the change of triangle amounts may occur at the so called *T-junctions* (Figure 5.4 and 5.5).

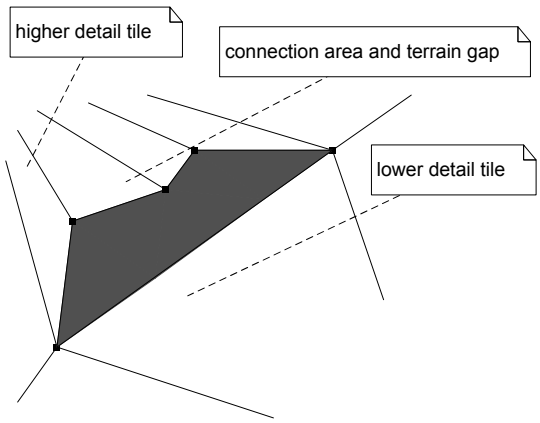


Figure 5.4: Terrain Gap

Figure 5.5: Gap in rendered Terrain

A solution for this problem is to triangulate these gaps. The mesh with the lower level of detail between two joining meshes adds triangles to achieve the same particularity in the connecting area as the linking tile. Therefore the tile first has to search for the boundary rectangles. These rectangles can split their edges through adding new vertices between the old edge points. Triangulation is done in the same way as before, because the mentioned triangle fan provides a method to construct the mesh independent from

the amount of vertices.

Figure 5.6 shows a higher detailed mesh surrounded by meshes with a reduced amount of triangles. The surrounding meshes added triangles at some of their edges to connect to the center mesh.

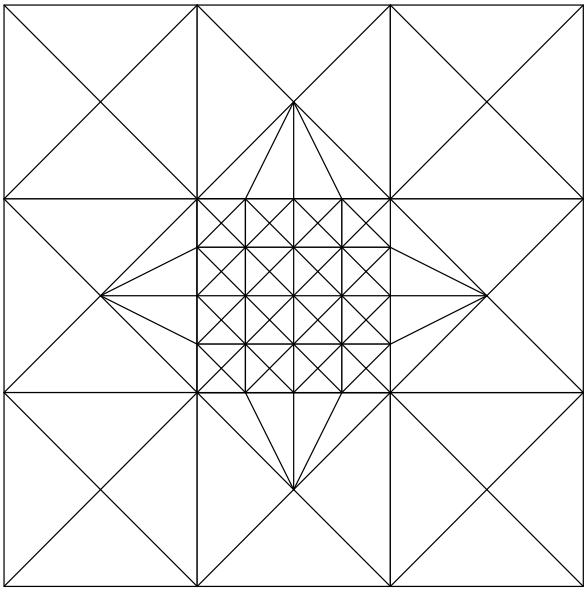


Figure 5.6: Terrain Mesh

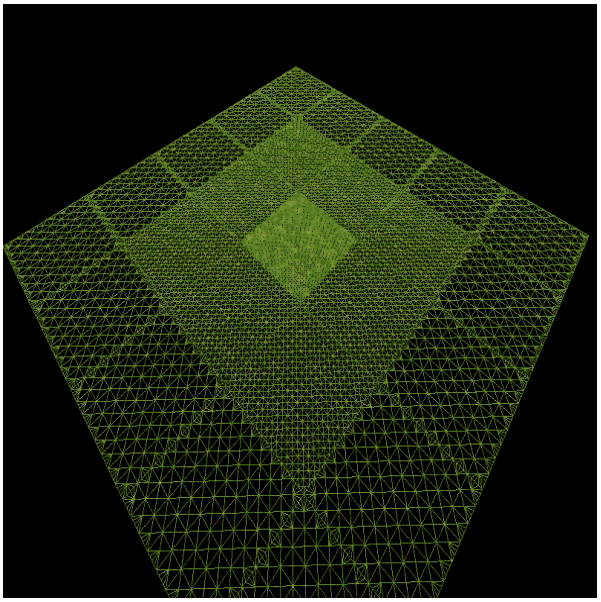


Figure 5.7: Wireframe Terrain

5.4.2 Terrain Vertex Shader

The updating process of height data on the graphic terrain's vertices can be accelerated by bringing it to the multiple parallel working vertex shader units on the GPU. This requires to take notice of the graphic card's data flow model during programming and to send the needed data for transforming the vertices to the vertex processors (section 5.2).

Before starting to program a decision had to be made about which shader language and what vertex shader version to use. The *Irrlicht Engine* currently supports the 2 languages *HLSL* and *GLSL (OpenGL Shading Language)* and unfortunately the otherwise preferred *Cg* language would demand a big change of the *Irrlicht Engine's* code. However, *Cg* support is planned for coming *Irrlicht* versions and in the light of this the terrain vertex shader was written in *HLSL* syntax to leave the door open for *Cg* in the future, as HLSL syntax is also understandable for the *Cg* compiler. The quite new vertex shader version 3.0 supports texture lookups by the vertex processor and would thus be useful for transferring data through textures to the GPU. As by the usage of this, older graphic hardware would not be supported by AutoSim anymore, vertex shader version 1.1 is

chosen involving a more difficult method of sending data.

Height data now has to be brought to the vertex shader through *VertexShaderConstants*, which are limited by the graphics card. Fortunately the transmitted data to the shader is usually also reduced by unprecise measured height data, as for many world positions height data not more accurate than one value per 10 meters can be obtained from geographical institutes. Nevertheless the size of a tile mesh is still limited, as the vertex shader has to process data for every position of the tile. For example height data as accurate as one value per 10 meter and the common *VertexShaderConstants* amount of 256 would limit the tile size to $130m * 130m$.

When the height map is loaded into the AutoSimClient program an array of height data for the vertex shader is created (section 5.3). Due to the usage of bilinear interpolation (5.1), height data can be obtained for every world position and a setting by the user has to specify how accurate the data really is. The data array for the vertex shader is created dependent on the world size and the distance between two data points. Before rendering a frame a part of the array is cut out and send to the shader. Sometimes it is not possible to cut out the current mesh's outline in the world from the vertex shader data array, because the tile may lie somewhere between data points. In the account of this the shader transmits an array expanded by the outlying data rows and information of the mesh's actual world position is used to determine the needed array part on the shader. The shader implements the bilinear interpolation method (5.1) and calculates the height for all vertices of the mesh out of the received height data.

According to the idea of *Geometric Clipmaps* the terrain mesh always has to follow the camera. The camera's world coordinates are sent as vertex shader constants to the GPU and after transforming them by the *Irrlicht Engine's* world view matrix every vertex is shifted by the camera's translation. Unfortunately in regions of lower detail this method can produce unwanted flipping of edges on the mesh, caused by a significant change of height data between two vertices. Due to the circumstance that a certain point in the world is sometimes closer to a vertex than in another time step, this point will move up and down as the terrain moves along the data. Figure 5.8 visualizes this effect in 2D. The position of the point on the top of the height data hill moves along the green arrow while the 2D lines (symbolizing the 2D version of the triangles) are continuously shifted from right to left.

The flipping edges effect can be avoided by waiting to a certain shifting point, dependent on the lowest detailed part of the mesh, instead of following the camera directly. Accidentally this leads to a popup effect of terrain at the borders of the mesh, which can be made nearly invisible by a big terrain mesh.

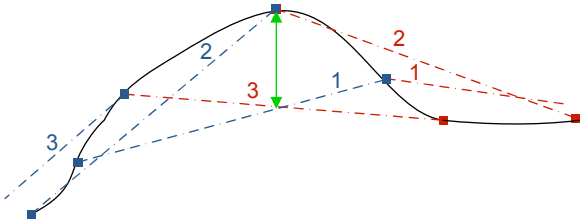


Figure 5.8: Flipping Edges

5.5 Creation of Static Objects

It often takes a long time to place houses or trees along a street, especially when the result cannot be immediately considered by the creator. For setting up a new testing environment, houses usually do not have to be exactly the same as in the real world, because the house look should not make any significant change for the robot controlling program. On account of this the OsmManipulator can accelerate the world creation process by automatically adding houses or trees along the streets. The manipulator constructs the points for the static objects by creating a new lane via the construction methods of the road generation process (chapter 7). The distances of the object's center to the street and to the next object are changeable through the OsmManipulator GUI.

All house and tree models to use are read from a *House File List* XML file. Before selecting a model the manipulator searches for areas inside the world's OSM file, which are defined as circular ways with a specific *OSM tag*. Therefore it has to calculate if the new house node is inside an area or not. The definition of the scalar product is the outgoing point for this calculation, as it can be used to calculate an angle α between two vectors:

$$\alpha = \arccos\left(\frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|}\right) \tag{5.5}$$

If two 2D vectors $\vec{a}$ and $\vec{b}$ are brought into 3D space, the sign of the 3rd coordinate of their cross product implies the direction of the angle between them in 2D space. Meaning by combining the angle calculation (equation 5.5) with a multiplication by the sign of the cross product, results in a directional angle between two vectors. Using this method makes it now possible to distinguish between inside an area or not. Lets assume the house node lies inside the area, meaning all nodes of the way lie in a circle around it. Vectors are constructed between every node and the house node and by going one direction around the circle, all angles between two vectors next to one another are added. Concerning the circle around the node, the angle should sum up to 360 degrees. On the other hand if a node lies outside an area, the added positive angles will be neutralized by negative angles that are added as the way goes down from the node's point of view. This method

is applied to all areas and by an angle of 0 degrees the manipulator knows the node is outside the area, whereas 360 degrees imply an area surrounding the node.

Having now detected the corresponding area of the static object node, the manipulator program can choose one of the area dependent model files to store it into the OSM file by creating a new OSM node at the object's world coordinates. Beside coordinates and model files these new entries include further information about the physics box and the rotation of the node (B.3). As the output file is again an OSM file containing streets and now also nodes for objects, it can be easily edited by one of the OSM programs from the *OpenStreetMap* website.

6 Road Construction

One stated objective of AutoSim is to recreate a street web of the real world inside the simulator. This could be done by creating the roads based on their real world illustrations by using a 3D modeling program and placing them in the correct positions inside the simulated world. However, creating large areas and even several cities would cause a lot of work of modeling and positioning the roads. The PC game *Midtown Madness* published by the *Microsoft Game Studios* was one of the first games to use a procedural method for building roads automatically out of road data. AutoSim follows this approach and constructs the roads and intersections in the simulator out of data obtained from the free street data service *OpenStreetMap* and doing so reduces the work of building road models whilst retaining the ideal of an exactly recreated world.

The section on *Splines* explains the theory of the mathematical spline functions used during road generation and the section on *RoadData* gives information about the data structure of the roads in AutoSim. Finally the section on *Road Generation* explains the entire construction process.

6.1 Splines

Curves in computer graphics are often created out of mathematical spline functions, because of the simplicity of their construction, their ease and accuracy of evaluation, and their capacity to approximate complex shapes [27]. A spline is a mathematical function defined piecewise by polynomials and, in general, every polynomial can be used to create a spline. However, functions like, for example, *Hermite* polynomials are making it easy to create smooth curves through a number of points. An interval of the *Hermite Spline* is defined between two control points, along a control variable t (equation 6.1):

$$p(t) = (2t^3 - 3t^2 + 1)p_i + (t^3 - 2t^2 + t)m_i + (-2t^3 + 3t^2)p_{i+1} + (t^3 - t^2)m_{i+1} \quad t \in [0;1] \quad (6.1)$$

The starting and ending points p_i and p_{i+1} are margining the hermite interval and the corresponding gradients m_i respectively m_{i+1} take influence on the shape of the curve. Figure 6.1 illustrates the influence of the four basis functions in the hermite interval defined along control variable t.

$$
\begin{aligned}
h_{00}(t) &= 2t^3 - 3t^2 + 1 \\
h_{10}(t) &= t^3 - 2t^2 + t \\
h_{01}(t) &= -2t^3 + 3t^2 \\
h_{11}(t) &= t^3 - t^2
\end{aligned}
$$

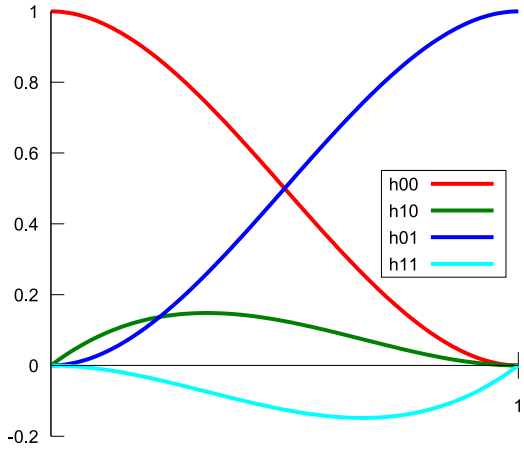


Figure 6.1: Hermite Basis Functions [36]

If t is 0 all basis functions except of $h_{00}(t)$ are 0, which means the hermite curve always starts out of starting point p_i. As at $t = 1$ only $h_{01}(t)$ is different from 0 the *Hermite Spline* also ensures the curve is ending at control point p_{i+1}. Defining *number of points* -1 intervals next to each other and each interval connecting two points constructs a spline curve as visualized by figure 6.2. As already mentioned the tangents at the control points determine the smoothness of the curve and in the light of this it's important how they are construction. *Kochanek Bartels* splines pursue the *Hermite* splines in adding formulas for constructing the tangents:

$$
\begin{aligned}
m_i &= \frac{(1-t)(1+b)(1+c)}{2}(p_i - p_{i-1}) + \frac{(1-t)(1-b)(1-c)}{2}(p_{i+1} - p_i) \\
m_{i+1} &= \frac{(1-t)(1+b)(1-c)}{2}(p_{i+1} - p_i) + \frac{(1-t)(1-b)(1+c)}{2}(p_{i+2} - p_{i+1}) \quad (6.2)
\end{aligned}
$$

where

$$
\begin{array}{lll}
\text{t} = \text{tension}; & t \in [-1;1] \\
\text{b} = \text{bias}; & b \in [-1;1] \\
\text{c} = \text{continuity}; & c \in [-1;1]
\end{array}
$$

Figure 6.3 shows the influence of the tension, the bias and the continuity parameter on an interval of the curve.

Continuity regulates the transitions between the intervals and only a fixed set to 0 of this value drives the transition to the usually desired tangent-continuous (C^1). The bias parameter b moves the curve a bit to either the starting or the ending point of an interval and tension t acts upon the sharpness of the curve in the interval, because it has affect

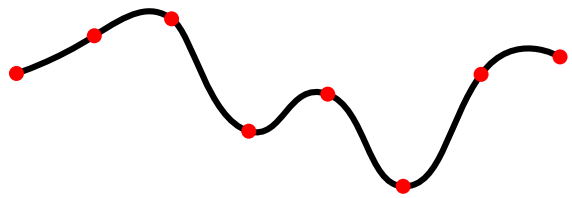


Figure 6.2: Spline [33]

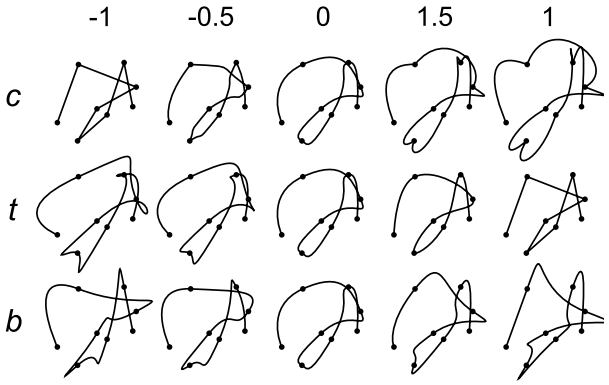


Figure 6.3: Kochanek Bartels Parameters [37]

on the length of the tangent vector.

When splines are used in computer graphics their length is often required to, for example, calculate the current position on the complete curve. The length of an interval can be obtained by integrating over the curve along t from 0 to 1 and the different intervals can be summed up afterwards to result in the complete length of the spline.

$$\text{Interval length} = \int_C ds = \int_a^b \sqrt{\dot{x}(t)^2 + \dot{y}(t)^2}dt \tag{6.3}$$

where $\dot{x}(t)$ and $\dot{y}(t)$ are the x and y values of the derived spline equation $\dot{p}(t)$.

6.2 Road Data

The data used for road generation can be downloaded as XML data files from the OSM server and contain a structure made of *node* and *way* XML sections. Nodes represent points in a 2D world and ways contain a list of nodes. Furthermore both XML sections have *OSM tags* specifying their usage. Figure 6.5 gives an overview of an already rendered node and way system of the *Perth* suburb *Crawley*, containing area surroundings that are also defined as ways.

AutoSim takes over the OSM file structure and adds a new type named *segment*. A segment is a part of a way and has two nodes defining the segment's starting and ending point. For this reason a way in the simulator now contains a list of segments instead of a list of nodes as before.

The class *OsmParser* parses all nodes and ways out of the OSM file into the simulator's

data structure. As the usage of those can be different the parser searches for ways and nodes representing streets respective static objects to store them in arrays and thus prepare them for the road generation. Due to the fact the *OsmParser* class is the only class interacting with the OSM file, a change of the OSM data structure would eventually only effect a change of the *OsmParser* class.

As roads and intersections are generated by the same classes on the AutoSimServer and the AutoSimClient, the constructed meshes are stored in order to be used for both the physics engine and the graphics engine. Therefore a hierarchical structure called *RoadData* was constructed, where the lowest parts contain the actual mesh represented by vertex and index lists. The intersections only need a further 3D vector for their position, whereas the roads are divided into segments storing multiple vertex and index lists for the different parts of the street. Listing 6.4 shows the structure for an *Intersection* and a *Road*. The roads contain *RoadSegments* and those contain again a position and lists for all vertices and indices of the street, the curbs and the pavements.

```
1  class Road
2  {
3  public:
4    Vertex m_Position;
5    VECTOR<RoadSegment*> roadSegment;
6  };
7
8  class Intersection
9  {
10 public:
11   Vertex m_Position;
12   VECTOR<Vertex> intersectionVertices;
13   VECTOR<int> intersectionIndices;
14 };
```

Figure 6.4: Road Data

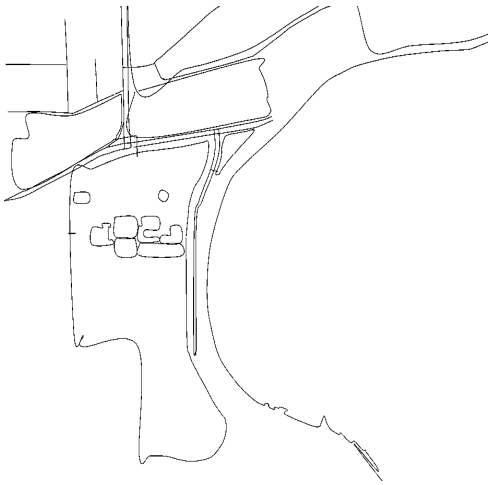


Figure 6.5: Streets in Perth

Finally server and client use the lists of vertices and indices of the road data to bring them into their mesh representation of the physics and graphics engine respectively.

6.3 Road Generation

Section 6.2 explained the *OsmParser* class parsing all OSM road data files into a *node*, *way* and *segment* structure and creating a separate list for the ways detected as roads. The *node*, *way* and *segment* classes keep all information of their XML section matches and additionally provide useful calculations for road generation. However, the main class for road generation is the *RoadFactory* and a call to its function *generateRoads()* is starting the construction process, which includes storing all the created meshes into a

road data structure. For simplifying the calculations the common righthanded x, y, z - coordinate system, with z pointing upwards, is used and after construction the vertices are transformed back into the lefthanded coordinate system of physics and graphics engine.

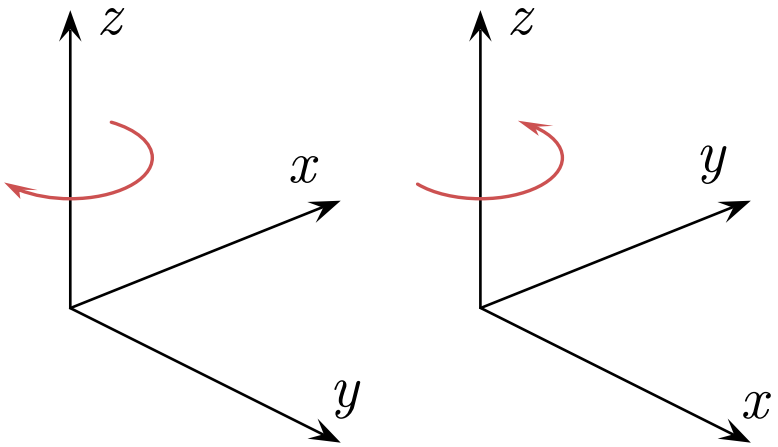


Figure 6.6: left handed and right handed coordinate system [35]

Iterating through the detected roads starts a new generation process for every road by first of all creating a new 2D Kochanek-Bartels spline class instance (6.1). All coordinates of the road belonging nodes are committed into the spline class. The spline is created by setting the bias, the tension and the continuity parameter always to 0, which was identified as the best configuration for creating a smooth and continuous curve.

The roads are divided into segments and every segment creation starts by checking both connected nodes whether being an intersection or not. Eventually an intersection creation is executed if the node is detected as an intersection by having more than two connecting segments.

The segment's construction continues by shifting through the segment's corresponding spline interval in a predefined step size while calculating vertices for the mesh on every position. The spline is taken as the center of the road and new vertices are constructed by moving from the current spline position along its normal until the desired distance to the road center is reached. Figure 6.7 shows this proceeding inside the calculation view and figure 6.8 visualizes the corresponding road mesh created by AutoSim.

Apparently the vertices for the pavements and curbs have to be lifted by the desired curb height value in order to obtain a proper road mesh for the segment. Every segment generation step integrates over its current step size interval and adds the value to a sum, representing the stridden length of the spline 6.1. The spline length is needed to calculate the texture coordinates for the vertex, as the street texture must always be repeated after

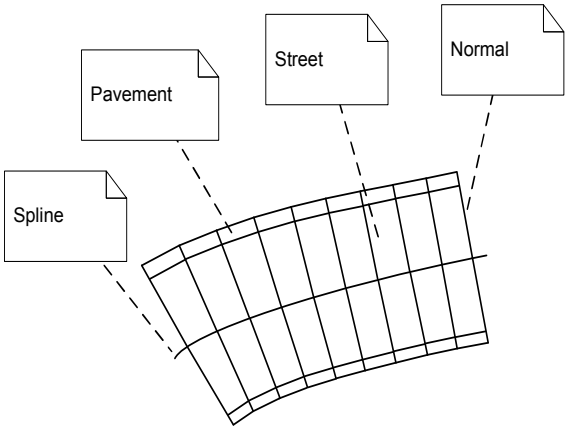


Figure 6.7: Road Construction

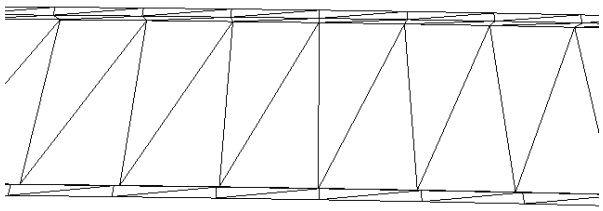


Figure 6.8: Constructed Road Part

reaching its boundaries. Therefore one texture coordinate always stays at a fixed value 0 respectively 1 and the second texture coordinate moves with the spline position. For example the fixed values for the lane texture are the street boundaries while the texture is repeated at its end through a moving second coordinate.

If multiple segments strike each other at an intersection node, the constructed road segments cannot end all together at the same node, because their road meshes would overlap. To avoid the overlap the *node* class has a function for calculating new connection points for the road segments. The number of points are dependent on the number of roads connected to the segment and the location of the points changes with the width of the streets. A T-junction would for example require the calculation of three points for its three connecting roads. Apparently the road segments must now be modified in order to end in the connection points of the intersection. First of all the calculated ending vertices of the road segment are shifted into these connection points and to provide a smooth transition of the road into the intersection, several other vertices should be influenced as well. Therefore the offset between the previous constructed ending vertices and the new calculated ending points is taken as input of a fading function that calculates the offsets' influence among the points of the segment. Figure 6.9 illustrates the method of fading the road into its new ending points and figure 6.10 presents a finished road generation junction.

As the offset must have its entire influence on the intersection transition and should completely disappear at the segment's connection to the next segment, the function must either decrease from 1 to 0 or increase from 0 to 1 inside an input interval from 0 to 1. The easiest, but nevertheless for this problem, most suitable functions are shown in equations 6.4 and 6.5.

Whereas the x^2 function can be used for adding increasing offsets to effect road fading at the segment's end, function $(x-1)^2$ is excellent for applying decreasing offsets to the

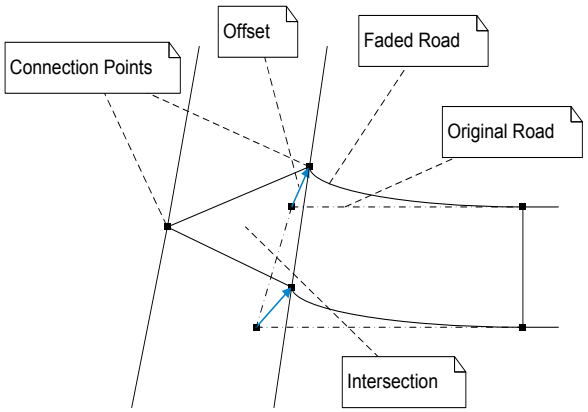


Figure 6.9: Offset Fading

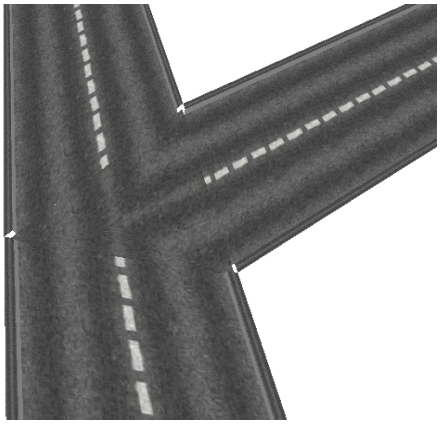


Figure 6.10: T-junction after Road Generation

$$y(t) = (x-1)^2 \qquad (6.4)$$

$$y(t) = x^2 \qquad (6.5)$$

segment's beginning.

The offsets for fading are calculated before the actual road segment construction process has started and every time a new vertex is created the increasing and decreasing offsets are added to the vertex's coordinates. After constructing the vertex it is transformed into the lefthanded coordinate system of graphics and physics engine and its position is calculated relative to the position of the road segment. Finally the indices for the mesh are created by the strip triangulation method explained in *Rendering Methods* (chapter 7) and all created road segment meshes are stored into the road data structure.

Creating the intersection meshes is simply done by taking the intersection node as center vertex and all segment connecting nodes as further vertices for applying the *Triangle Fan* method (7.1). Further nodes are added in between every two connection nodes to ensure a seamless texture transition inside the intersections. Thus the number of triangles a mesh consists of is now always of an even value and so the texture coordinates can be calculated in a way to always make a seamless texture transition from one triangle to the next one. Therefore the texture coordinates at the border of two connecting triangles are always the same.

7 Rendering Methods

This chapter starts by explaining the theory behind *Triangulation* of meshes and continues by giving an overview of the AutoSimClient methods for *Rendering the Scene* and applying *GeoMipMap* to the graphic nodes.

7.1 Triangulation

A model in a 3D graphics world is usually build out of triangles and its mesh is a collection of vertices and indices that form a 3D entity. Often, 3D models are created as irregular meshes in a 3D modeling program, meaning they consist of unordered lists of vertices and indices. Nevertheless it is sometimes necessary to program procedural meshes inside a software and therefore it is recommended to use one of the already existing triangulation methods. Furthermore graphic APIs often provide faster rendering and lower memory use if vertex and index lists are passed in a certain order.

Two common methods of triangulating meshes are the triangle strip and the triangle fan [19]. A strip is usually used if a series of connected triangles is desired, like for streets or house walls. For example the strip in figure 7.2 is visualized by using vertices v_0, v_1, and v_2 to draw the first triangle, v_1, v_3, and v_2 to draw the second triangle, v_2, v_3, and v_4 to draw the third, v_3, v_5, and v_4 to draw the fourth. It has to be noted that the vertices of the second and fourth triangles are out of order. This is required to make sure that all the triangles are drawn in a clockwise orientation as a convention in computer graphics constitutes that triangles are only visualized if their indication is clockwise from the spectator's perspective.

Constructing triangle strip meshes is made easier if the vertices are in a certain order. Most common and also useful is the order used in figure 7.2, where the numbers of the strip's margin vertices are always v_i and v_{i+1}. The *for loop* of listing 7.1 constructs the indices for the triangles out of such an ordered vertex list.

A triangle fan is similar to a triangle strip, except that all the triangles share one vertex. This is shown in the illustration 7.4. The system uses vertices v_1, v_2, and v_0 to draw the first triangle, v_2, v_3, and v_0 to draw the second triangle, v_3, v_4, and v_0 to draw the third triangle and so on. Triangle fans are useful if meshes have to be constructed for

```
1  for(int i=0;i<numberofvertices-3;i+=2)
2  {
3    // indices triangle 1
4    indices.push_back(i);
5    indices.push_back(i+1);
6    indices.push_back(i+2);
7
8      // indices triangle 2
9    indices.push_back(i+2);
10   indices.push_back(i+1);
11   indices.push_back(i+3);
12 }
```

Figure 7.1: Strip Index Calculation

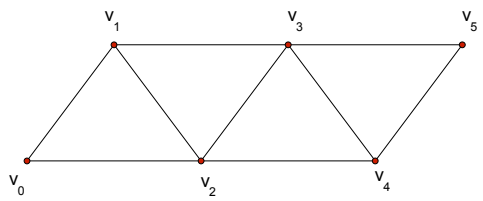


Figure 7.2: Triangle Strip

surfaces, where only points are available for the surface's outline. Listing 7.3 shows a method of constructing indices for a triangle fan by assuming the main vertex is stored at the beginning of the vertex list. After the loop another triangle is added to close the fan to a circle, meaning now the mesh consists of a real center vertex surrounding by the vertices of the surface's outline.

```
1  for(int i=2;i<numberofvertices;i++)
2  {
3    // indices triangle 1
4    indices.push_back(i);
5    indices.push_back(0);
6    indices.push_back(i-1);
7  }
8  // indices closing triangle
9  indices.push_back(1);
10 indices.push_back(0);
11 indices.push_back(numberofvertices-1);
```

Figure 7.3: Fan Index Calculation

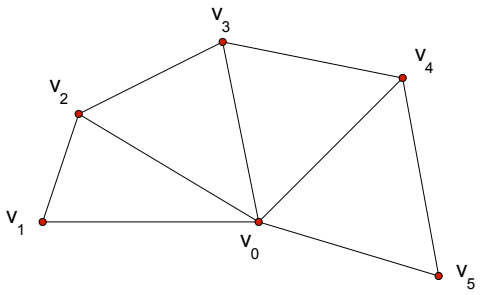


Figure 7.4: Triangle Fan

7.2 Rendering the Scene

The AutoSimClient program provides the user with two different targets for rendering the graphics scene. The scene can be either printed straight onto the screen or transferred to a special render target texture, whereof it can be obtained through the *ClientUserProgramAPI*. Therefore the user has to call the corresponding function with a world camera as argument.

Every robot in the graphics world can have two different types of connected cameras: A *VirtualCamera* is fixed on the robot by moving and rotating with it and a *ThirdPerson-*

Camera is regarding the robot from a certain outside point of view. The third person camera does not rotate and always keeps a specified height relative to the robot.

Before rendering to the screen the AutoSimClient checks if a new data package has arrived from the server since the last frame was rendered. If so the client starts rendering and otherwise the graphics go into a sleep mode. Nevertheless a request for a new camera image is processed immediately.

7.3 GeoMipMap

Visualization of huge graphic worlds is always connected to a reduction of world details in the distance. If graphic objects are far away from the camera position a human's eyes can hardly see any difference between detailed versions of graphic models and models with a lower number of polygons. Of course, the graphic models have to be reduced in a proper way. Methods like terrain following the camera or polygon reducing runtime algorithms are very difficult to apply on irregular meshes constructed by 3D modeling programs. An alternate method to reduce the polygons of a mesh during runtime is to load versions of different detail-levels of the meshes and replace them if they are at a certain distance. This method is called *GeoMipMap* and is used by the AutoSimClient to reduce the number of polygons sent to the graphics card.

The *GraphicNode* derived class *MeshGraphicNode* implements the *GeoMipMap* method and executes it when an update on the *MeshGraphicNode* is called. Therefore it has a member pointer to the only camera existing in the 3D world and calculates the current distance to it. Dependent on the distance of the node to the camera the representing 3D mesh switches between higher and lower detail versions. If the node is too far away it is set to invisible.

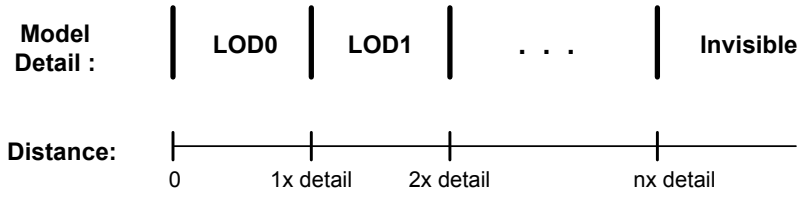


Figure 7.5: GeoMipMap

The number of detail levels for a *MeshGraphicNode* is defined by the number of models a user makes available for the program and a distance variable defines the size of one level of detail area.

8 Conclusion and Future Work

This thesis introduced a new open source 3D driving simulator framework called AutoSim which provides the user with a high level testing environment for autonomous driving vehicles. The framework is based on open source libraries and open source data services to accomplish the establishment of a software available for a wide audience by taking benefit of numerous existing free services. AutoSim is running the actual physics simulation and its graphical representation completely separate through a networking based concept of two disjoint working programs. Multiple additionally added user programs allow external control of objects and the simulation environment while the API makes it possible to observe and use simulation data in external running programs like for example image processing frameworks. The biggest effort of this thesis was put into constructing the graphical representation of the simulation world, which makes AutoSim now able to visualize an almost unlimited world size. Furthermore the framework is capable of reconstructing and editing real world street data, adding static objects like houses to the scene and obtaining camera images from every camera in the world. Finally the component-based software design makes AutoSim very flexible, modular and extensible by giving developers the possibility to easily add custom requirements by modifying the well documented disjoint software parts.

Currently AutoSim is only available for Microsoft Windows but as all used libraries are cross platform, a version for Linux and possibly Macintosh will follow. Therefore the compiler of the used shader programs will be changed to the cross-platform working *Cg* compiler as soon as the *Irrlicht Engine* supports *Cg* shading language. An alternative, but more effort taking approach is to rewrite the shaders into GLSL and use the OpenGL version of the *Irrlicht video driver* for visualizing the graphics scene.

The graphical representation of the virtual environment in AutoSim was designed in order to render a fast and large world. Therefore some of the best known today detail reducing methods have been implemented and they make it now possible to place a huge amount of graphic models into the simulation world. Future work will improve the graphical representation by adding shader or normal map materials to the 3D models for ameliorating their realism. As shader and normal maps are already provided by the graphic engine, the only effort has to be put into creating the textures and writing the shader programs. Additionally the lights in the world will be reworked by preparing the materials of the models for lighting and changing the dynamic light system of the scene.

The AutoSim framework already delivers many tutorials and examples of how to create a simulation world, observe simulation data and control the simulation robots, but future work on more extensive and higher sophisticated user programs will give the user a better entrance point for starting research in autonomous locomotion. Eventually preparing further testing environments, either by loading and rebuilding particular real world parts or by developing specially designed testing courses will be a future task and give the user a quicker start to test the control programs.

For the purpose of bringing real world road data into the simulator, the road meshes have been constructed by procedural road generation methods developed by the author and its visualization is now on a high level for constructing decent street webs. The models of roads and intersections and their lane markings will be improved to make them look even more realistic and will accelerate their already fast rendering speed in future versions. Road signs will be added for providing the image processing controlled robots more criteria for decision making during autonomous driving.

A Tutorials

A.1 The AutoSimServer kick start guide

This tutorial gives a brief introduction to the AVSE Server. It shows how to get started and explains the core features of the program. The following image shows the GUI.

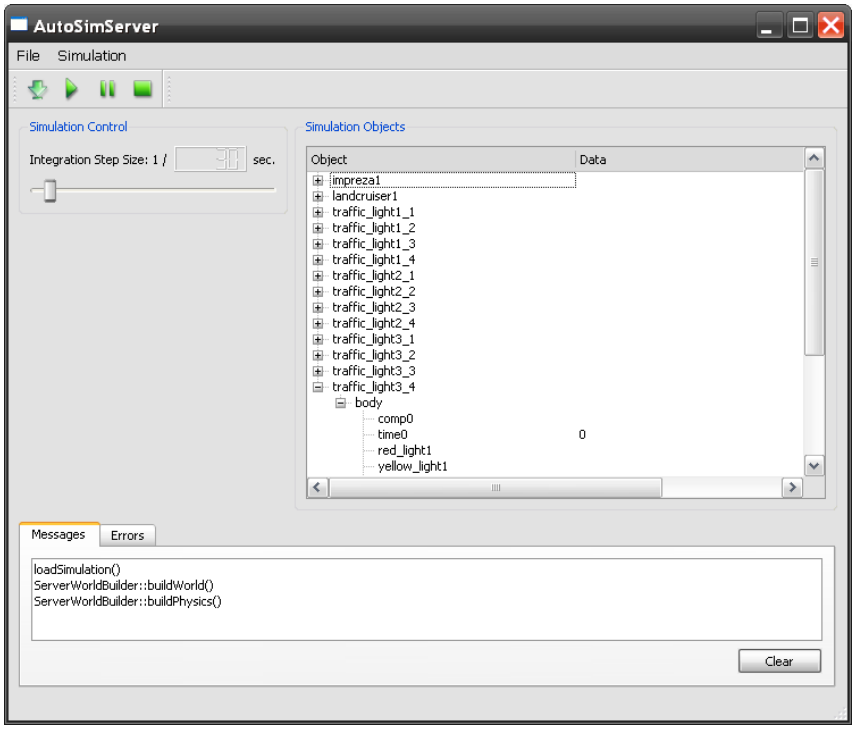


Figure A.1: Graphical User Interface of the AutoSim Server

To setup and start the Server for your Simulation do the following:

1. From the *File* menu choose *Select World File*. An *Open* dialogue will show up.
2. Navigate to the *worlds* folder pick a World file and click *Open*. This file must be the same on the server as well as on all clients in order to make the simulation work. If you do not select a World File a default World File will be loaded.
3. To *load* the Simulation you can either click on the *Load button* or select *Load* from

the *Simulation* menu. This takes aproximately 10 sec depending on the computer you use.

4. After the *Load* button poped up again the loading process is completed and the Clients can be *connected* to the Server. You will also see the all the *Robots* defined in the World File showing up in the *Simulation Objects* box.

5. To Start the Simulation simply hit the *Run Button* or click *Run* in the *Simulation* menu. This starts the actual *simulation process* and makes the server start sending updates to the clients through the network.

6. If you want to break the running simulation at any point just click the *Pause Button*. This will stop updating the physics. To proceed press the *Run Button* again or choose *Run* from the *Simulation Menu*.

7. The Slider inside the *Simulation Control* box can be used to adjust the *integration step size* for the physics engine. Note that the number displayed is the fraction of a second by which the physics will proceed in a single integration step. Pushing the slider to the right gives you more accuracy but slows the simulation down - which might not necessarily be bad.

A.2 Working with the AutoSimClient

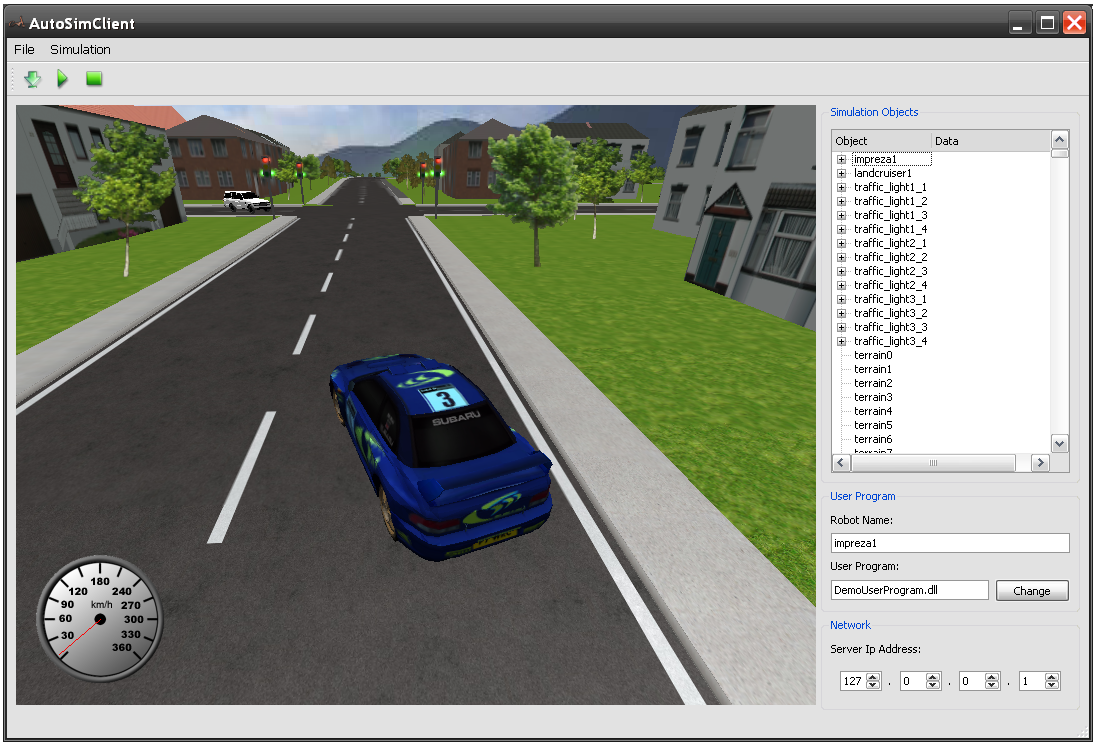


Figure A.2: AutoSimClient

To setup, start and use the AutoSimClient for your Simulation do the following:

1. From the *File* menu choose *Select World File*. An *Open* dialogue will show up.
2. Navigate to the *worlds* folder pick a World file and click *Open*. This file must be the same on the server as well as on all clients in order to make the simulation work. If you do not select a World File a default World File will be loaded.
3. To load the Simulation you can either click on the *Load button* or select *Load* from the *Simulation* menu. This takes approximately 10 sec depending on the computer you use.
4. After the *Load* button popped up again the loading process is completed and the client can be connected to the server. You will also see all the robots defined in the World File and additional objects showing up in the *Simulation Objects* box.
5. Set the IP address of the server you want to connect to in the *Network* box. The default value connects to the localhost meaning to a AutoSimServer running on the same pc as the AutoSimClient.
6. To set the robot you want to control by a UserProgram put your mouse cursor into the *Robot Name* input field of the *UserProgram* box and enter the name. The corresponding UserProgram can be set in the *UserProgram* input field below. Clicking on the *Change* button next to it opens a file dialogue that makes it easier to locate and select the UserProgram. This file dialogue can also be reached by navigating through *File* menu to *Select User Program*.
7. To Connect the client to the server, start the rendering process and execute the UserProgram simply hit the *Run Button* or click *Run* in the *Simulation* menu.
8. If you want to quit your AutoSimClient program you can do this in 3 ways: You can either navigate to *File* menu and choose *Quit*, hit the window's *Closing Button* or press *ESC* on your keyboard.

A.3 How to write a User Program

This chapter gives an introduction on the User Program API and how to use it. It also explains in short how to write compile and load a custom User Program into the simulation.

A.3.1 Workings of the User Program

The User Program is basically a C or C++ program that makes use of the API provided by the server and / or client to access sensors and actuators and is being compiled as a DLL.

Figure A.3: User Program

For convenience the main function is automatically being exported to provide an entry point to the calling thread and to make it appear more familiar to beginners since it looks like a "normal" C program.
Inside the main block the user program should enter a loop that controls the robot. This loop is executed in a separated thread. As soon as main returns the belonging thread will terminate as well.

A.3.2 The User Program API

The User Program API is available on both server and client. It provides in substance two functions that are used to access all devices on all robots:

Listing A.1: Parts section of a robot configuration file

```
1 namespace UserProgramAPI
2 {
3   SimDeviceError setData( SimDeviceName device, DeviceData *data, int dataSize
      );
4   SimDeviceError getData( SimDeviceName device, DeviceData *data, int dataSize
      );
```

```
5 };
```

Both functions take a SimDeviceName, a DeviceData pointer to the data that is going to be set / read and the byte size as their arguments. For convenience there are two macros defined in the include file that wrap around the setData and getData functions and save some typing and will be explained in the following example.

SimDevice: name specifies the device that is being accessed. It is a typedef for a std::string and has the following syntax: <RobotName>.<Part>.<Device>

DeviceData: typedef for void

dataSize: the byte size of the data

A.3.3 The Client User Program API

The User Program API is only available on the client and can be used to access the cameras of all robots.

Listing A.2: Parts section of a robot configuration file

```
1 namespace ClientUserProgramAPI
2 {
3   typedef unsigned int * VirtualCameraImage;
4   int getImageHeight();
5   int getImageWidth();
6   VirtualCameraImage getImage( UserProgramAPI::SimDeviceName camera );
7   void unlockImage();
8 };
```

The functions getImageHeight and getImageWidth return the size of the image that is beeing rendered from the virtual camera when getImage is called which returns a pointer to an array of the dimension 32bit * height * width. Every 32 bit value represents one pixel with the following color format: alpha, blue, green, red (8 bit each).
In order to obtain a new VirtualCameraImage call the unlockImage function which unlocks the internal mutex on the texture used by the rendering system.

A.3.4 A Simple Example

This example gives a short line by line introduction on how to use the API to obtain and write data from within a user program.

Listing A.3: Parts section of a robot configuration file

```
1 #include "UserProgramAPI.h" // include the user program API definitions
2 #include <windows.h> // for the Sleep() function
3
```

```
4  // for convenience: include the UserProgramAPI namespace
5  using namespace UserProgramAPI;
6
7  // Entry point for the user program.
8  // Do not change argument list or  return value!
9  int main(int argc, char *argv[])
10 {
11
12   // device names of an actuator and a sensor that are defined
13   // in the corresponding robot description file
14   SimDeviceName indicator = "chassis.indicator_light_back_left";
15   SimDeviceName inclinometer = "chassis.inclino0";
16
17     // the robot name that the user program belongs to is
18     // always the first argument string
19   RobotName robot = argv[0];
20
21   // variables to store the data from the devices
22   float intensity = 0.0f;
23   float angle = 0.0f;
24   float previousAngle = 0.0f;
25
26   // get the current angle from the inclinometer and make the
27   // left indicator light blink if the robot turns left
28   while(true)
29   {
30     GET_DATA(robot+"."+inclinometer, angle);
31
32     if ( (angle - previousAngle) < 0.0f || intensity == 1.0f )
33       intensity = 0.0f;
34     else
35       intensity = 1.0f;
36
37     previousAngle = angle;
38
39     SET_DATA(robot+"."+indicator, intensity);
40
41     Sleep(300);
42   }
43
44   return 0;
45 }
```

For more examples see the examples folder of the distribution: The Joystick example shows how to use data from a joystick device to control a robot The VirtualCamera retrieves images from a virtual camera and stores them on the The DemoUserProgram is the most complicated example: It uses its own GUI to obtain and display virtual camera images, control the robot and turn lights on and off.

A.4 Manipulate an OSM file in 6 steps

The OsmManipulator helps the user creating a world and is used before the simulation is started. During the actual simulation process the OsmManipulator does not have any functionality. This tutorial gives a quick start guide for using the OsmManipulator.

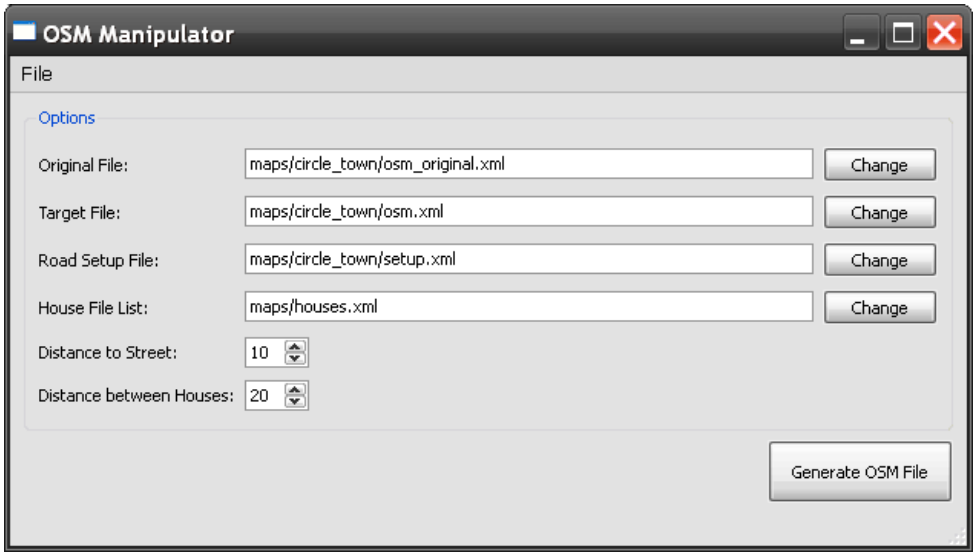


Figure A.4: OsmManipulator

1. Use your mouse to click on *Change* button next to the *Original File* input field. A file dialogue will show up. Navigate to the Osm file you want to manipulate and click *Open*. This file must be an Osm file of a version your OsmParser can handle with. For further information go to *General info on Osm Files* B.3.
2. Go to the *Target File* input field an either enter the path and file name or browse through the file dialogue by clicking on the *Change* button to specify where your created Osm File will be stored.
3. Select a *Map Setup File* in the file dialogue of the next input field. Information about how a *Map Setup File* should look like can be obtained in the *Map Setup File* description B.5.
4. Finally select a *House File List*. *House File List description* B.6 looks into these files.
5. Set the *Distance to Street* of the house centers and the *Distance between Houses* in a house lane in the two corresponding input fields.
6. Start and Create the new Osm file by clicking on the *Generate OSM File* button.

B The Configuration Files

B.1 General Syntax

All configuration files used in the simulation framework conform to the XML 1.0 standard. However, due to the way the data is being parsed into the internal generic object model, every configuration file must stick to a certain structure.
Here I will take the *robot* description file as an example:

Listing B.1: Examplary Configuration file structure

```
1  <?xml version="1.0"?>
2  <Robot name="impreza">
3
4    <Parts>
5      <Box name="chassis">
6        <Position x="0.0" y="0.0" z="0.0" />
7        <Size x="1.994999" y="1.265000" z="4.760004" />
8        <Mass value="1400.0" />
9        <CenterOfMass x="0.000000" y="0.376204" z="-0.764765" />
10       <Model path="models/impreza/chassis" type="3ds" />
11     </Box>
12   </Parts>
13
14   <Devices>
15     <Camera name="third_person">
16       <Part name="chassis" />
17       <Type name="thirdperson" />
18       <Position x="0.0" y="3.0" z="-4.0" />
19     </Camera>
20     <GyroscopeSensor name="gyro0">
21       <Part name="chassis" />
22       <Axis x="1.0" y="0.0" z="0.0" />
23       <Alpha value="0.05" />
24     </GyroscopeSensor>
25   </Devices>
26
27 </Robot>
```

Every file is surrounded by a tag (in this case the *Robot* tag) that can either contain *sections* or *BuildData* blocks. *Sections* are simply used to group *BuildData* blocks that belong together like the *devices* that belong to a robot in our example.

The *BuildData* blocks are the actual heart of any configuration file: They contain a list of parameters represented by their child tags that provide all information necessary to create a specific object inside the simulation e.g. a *Device* of a *Robot*. Every parameter tag on its part provides a list of either *double* **or** *string* attributes. In addition every *BuildData* block needs a *name* attribute.
If you want to customize or create your own configuration files it is generally a good approach to make a copy of an existing file or file structure and change it rather than writing a new one from scratch. You will find a detailed description of all tags and attributes inside the corresponding XML file.

B.2 Customizing a World File

The *World File* includes all information needed to set up a simulation on the server as well as on the client and should be located in the *worlds* folder. It is split up into four major sections:

- Graphics: Parameters for the visualization
- Physics: Parameters for the physics engine
- Terrain: Path to the folder containing the terrain information
- Objects: All objects that appear in the simulation

Please note that all parameters have to be exactly the same on the server and client systems except for the *Graphics* section. A good way to syncronize the world file is to keep it on the server and to share it with the clients (e. g. via Windows File Sharing).
All parameters are documented inside the XML file.

B.3 General info on OSM Files

The AutoSim framework uses street data from the open source website *OpenStreetMap* to construct roads for the simulated world. Actually OSM data can be downloaded in file version 0.5 and hence explanation of older versions is not provided any more. However, an *OsmParser* for OSM data 0.4 still exists. This section gives a short description about the file structure whereas more detailed information can be obtained from the *OpenStreetMap* website [23].

The Osm file structure is very simple and only consists of nodes and ways. A node always has a unique identification number and represents a point in the world whose position is given in GPS coordinates. A way also has a unique id but simply contains a list of nodes represented by their ids. The nodes are ordered to form a continues way as they are parsed by *OpenStreetMap* and AutoSim in the sequence they are listed. If two node

ids would be flipped, this would cause a complete change of the way!

Within the OSM files *Tags* are used to store information about the type of a node or way. Tags always consist of keys and values. A *key* declares the type of the Osm element whereas the *value* gives a more detailed expression for the type. Common keys for roads are for example *highways*. A *highway* can have values like *residential*, *motorway*, etc.

AutoSim tries to adopt the *highway* and *landuse tags* defined in the *Map Features* section [22] of the OpenStreetMap webpage. However, custom tags can be defined as well. Table B.1 shows the list of *highways* used by the demo world of AutoSim:

key	value
highway	motorway
highway	motorway_link
highway	trunk
highway	trunk_link
highway	primary
highway	primary_link
highway	secondary
highway	tertiary
highway	unclassified
highway	unsurfaced
highway	track
highway	residential
highway	service

Table B.1: Highways

Regarding the *Map Features* of the *OpenStreetMap* website a lot more existing highway *tags* can be obtained. To enable the simulator to load further street types they have to be added to the *Map Setup File* B.5.

Creating new *landuse* areas can be done by constructing a new closed way with a *landuse* tag in the OSM file. If houses or trees are added by the OsmManipulator their models are selected related to the *landuse* area they are surrounded by. By default, if no area is defined, the manipulator uses *residential* house models. The *landuse* areas to use must be declared within the *House File List* B.6 and an example of *landuse* tags is presented by table B.2:

Additionally to using predefined OSM tags the user can also define his own tags. The OsmManipulator for example constructs house nodes that do not exist in the official OSM documentation. The tags of the house nodes tell the simulator the necessary data for loading house models into the simulated world. An overview of the house node is given by table B.3:

key	value
landuse	residential
landuse	retail
landuse	commercial
landuse	industrial
landuse	forest

Table B.2: Landuse

key	value
filePath	defining the file path of the 3D model
fileType	model type
landuse	house area
name	house name
rotY	rotation angle for rotating the house around the up going Y-axis
sizeX	size of the house in x dimension
sizeY	size of the house in y dimension (up)
sizeZ	size of the house in z dimension

Table B.3: House Node

B.4 The Robot File

The robot files contain the description of all robots. It is split up into 2 major sections: the parts describing all physical body parts and links of the robot and the devices section containing the descriptions of all sensors and actuators attached to the robot.
A simulation may contain an arbitrary number of robots using the same description file and / or user program.

Listing B.2: Parts section of a robot configuration file

```
1  <Parts>
2    <Box name="chassis">
3      <Position x="0.0" y="0.0" z="0.0" />
4      <Size x="1.994999" y="1.265000" z="4.760004" />
5      <Mass value="1400.0" />
6      <CenterOfMass x="0.000000" y="0.376204" z="-0.764765" />
7      <Model path="models/myrobot/chassis" type="3ds" />
8    </Box>
9    <Box name="trailer">
10     <Position x="0.0" y="0.0" z="10.0" />
11     <Size x="1.994999" y="1.265000" z="4.760004" />
12     <Mass value="1400.0" />
13     <CenterOfMass x="0.000000" y="0.376204" z="-0.764765" />
14     <Model path="models/myrobot/trailer" type="3ds" />
15   </Box>
16   <SphericalLink name="towbar">
17     <Parent name="chassis" />
```

```
18      <Child name="trailer" />
19      <Position x="0.0" y="0.0" z="5.0" />
20    </SphericalLink>
21  </Parts>
```

The only parts that are currently available in the physics are *boxes* and *spherical links*. Please note that links must be defined after the body parts. In addition to the physics parameters such as position, size, mass and center of mass you have to specify the path and type of the graphics model that represents the body part, e. g. the chassis.

Listing B.3: Devices section of a robot configuration file

```
1  <Devices>
2    <DriveActuator name="drive_chassis">
3      <Part name="chassis" />
4      <MotorForce value="5000.0" />
5      <BrakeForce value="10.0" />
6    </DriveActuator>
7    <WheelDevice name="front_left">
8      <Part name="chassis" />
9      <DriveActuator name="drive_chassis" />
10     <Position x="-0.80" y="-0.45" z="1.35" />
11     <Radius value="0.36"/>
12     <Width value="0.245"/>
13     <SuspensionRestLength value="0.1"/>
14     <SuspensionKs value="200"/>
15     <SuspensionKd value="23"/>
16     <Powered value="true"/>
17     <Steering value="true"/>
18     <Brakes value="false"/>
19     <Model path="models/impreza/wheels/front_left" type="3ds"/>
20   </WheelDevice>
21   <Camera name="front_view">
22     <Part name="chassis" />
23     <Type name="fixed" />
24     <Position x="0.0" y="1.0" z="1.4" />
25     <Direction x="0.0" y="-0.25" z="1.0" />
26     <UpVector x="0.0" y="1.0" z="0.0" />
27   </Camera>
28   <GyroscopeSensor name="gyro0">
29     <Part name="chassis" />
30     <Axis x="1.0" y="0.0" z="0.0" />
31     <Alpha value="0.05" />
32     <WhiteGaussianNoise range="0.01" offset="0.0"/>
33   </GyroscopeSensor>
34 </Devices>
```

In order to be able to attach wheels to a driving robot every body part needs a DriveActuator device. You also have to specify the part every sensor / actuator belongs to.
Moreover every device can have a list of noise sources that have an offset and a range. All noises are generated individually and added to the original sensor value every time step.
The following devices are available:

Actuators:

- DriveActuator: Cotrols the wheels of a robot
- WheelDevice: Represents the wheel plus the belonging suspension and brake

Sensors:

- Camera: A virtual camera for the robot
- GyroscopeSensor: measures the angular acceleration of a body part
- VelocimeterSensor: measures the speed of a body part
- PSDSensor: Position sensitive device - measures distances to other physics bodies
- GPSSensor: Global Positioning System - returns a string containing the current coordinates, velocity, time and a checksum
- CompassSensor: a compass
- InclinometerSensor: measures the difference between the current and an initial orientation in relation to a give axis
- TimeDevice: returns the simulation time
- LightDevice: simple light
- HeadLightDevice: light with a cone

Noise:

- WhiteNoise: Adds white noise to the sensor value
- WhiteGaussianNoise: Adds white gaussian noise to the sensor value

For a more detailed description of all devices and parameters see the impreza.xml file.

B.5 The Map Setup File

The *Map Setup File* is a XML file storing settings for the terrain and road generation. Its name must be setup.xml and it has to be located in the *map* folder specified in the world file's *Terrain* section. To be used by the simulator the *Map Setup File* must fulfill structure and naming conventions explained by this section. The AutoSim framework provides the user a complete demo world to allow a simulator quick start and to give an example of world construction. The exemplary *Map Setup File* used for this documentation is available in the map folder of the demo world and includes many explaining comments. In the light of this some self explaining sections of the *Map Setup File* are just copied out of the actual XML code, whereas difficult parts is given a more detailed consideration here.

To meet the requirements of the AutoSim framework general XML syntax ?? all sections of the setup file must be children of the *TerrainSetup* main section. The first section *OsmSetup* holds the GPS coordinates of the map center used for converting the openstreetmap nodes into the AutoSim world coordinates B.3.

Listing B.4: OsmSetup

```
<?xml version="1.0" encoding="utf-8"?>
<Setup name="circle_town">
  <TerrainSetup>
    <OsmSetup name="circle_town">
      <!-- Lat and lon coordinates of the simulation world center-->
      <NodeOffset lat ="-31.9728" lon="115.827" />
    </OsmSetup>
```

The *RoadDimensions* section allows the user to take influence on the road generation process. Explanation of the values is given by the comments above them.

Listing B.5: RoadDimensions

```
<RoadDimensions name="circle_town">
  <!-- Specifying how much (in meters) the roads are lifted above the terrain.
      If this value is set to low graphic problems may occur.-->
  <HeightAboveTerrain value="0.03" />
  <!-- Height of the curbs in distance to the road lanes in meters.-->
  <CurbHeight value ="0.15" />
  <!-- Number of levels of detail constructed for the roads. At least 1 level
      of detail is constructed. -->
  <LevelsOfDetail value="3.0"/>
</RoadDimensions>
```

Within the OSM file description B.3 many different types of highways can be defined. Constructing the highways the AutoSimServer and the AutoSimClient need to know the highway types they should load and the size they will be represented in the world. All the highways listed in the *HighwayDimensions* section are loaded into the world and constructed in the specified size.

Listing B.6: HighwayDimensions

```
<HighwayDimensions name="circle_town">
  <motorway width ="16.0" leftPavementWidth ="1.5" rightPavementWidth ="1.5"/>
  <motorway_link width ="4.0" leftPavementWidth ="0.0" rightPavementWidth ="0.0
      "/>
  <trunk width ="12.0" leftPavementWidth ="1.5" rightPavementWidth ="1.5"/>
  <trunk_link width ="4.0" leftPavementWidth ="0.0" rightPavementWidth ="0.0"/>
  <primary width ="16.0" leftPavementWidth ="1.5" rightPavementWidth ="1.5"/>
  <primary_link width ="4.0" leftPavementWidth ="0.0" rightPavementWidth ="0.0"
      />
  <secondary width ="8.0" leftPavementWidth ="1.5" rightPavementWidth ="1.5"/>
  <tertiary width ="6.0" leftPavementWidth ="1.5" rightPavementWidth ="1.5"/>
  <unclassified width ="6.0" leftPavementWidth ="0.0" rightPavementWidth ="0.0"
      />
  <unsurfaced width ="3.0" leftPavementWidth ="1.5" rightPavementWidth ="1.5"/>
```

```
12   <track width ="3.0" leftPavementWidth ="1.5" rightPavementWidth ="1.5"/>
13   <residential width ="8.0" leftPavementWidth ="1.5" rightPavementWidth ="1.5"/
         >
14   <service width ="3.0" leftPavementWidth ="1.5" rightPavementWidth ="1.5"/>
15 </HighwayDimensions>
```

The next two sections contain the filenames for the road and terrain textures as well as options concerning these textures. The intersection texture needs some more explanation: As within a junction should not be a visible changeover caused by a seamed texture the intersections are triangulated by a triangle fan B.1 in a way to provide a seamless transition.

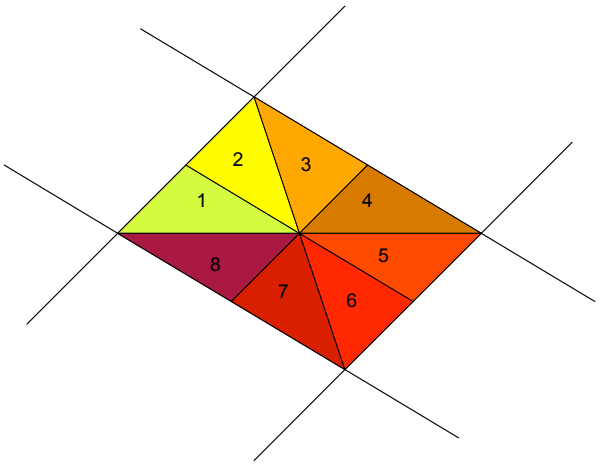


Figure B.1: Triangle Fan

Thus every triangle has to be split up once again because the number of triangles for every junction must be even. A T-crossing now consists of 6 triangles and a 4-Street crossing out of 8 triangles. Each of these triangles has the same texture on it. The part cut out of the given texture is a triangle with texture coordinates (0,0),(0,1),(1,1), where (1,1) is the center of the intersection.

Listing B.7: Textures

```
1  <RoadTextures name="circle_town">
2    <Road file ="media/roads/road.jpg"/>
3    <LeftPavement file ="media/roads/lane_withoutmarks.JPG"/>
4    <RightPavement file ="media/roads/lane_withoutmarks.JPG"/>
5    <LeftCurb file ="media/roads/terrain-heightmap_gray.bmp"/>
6    <RightCurb file ="media/roads/terrain-heightmap_gray.bmp"/>
7    <Intersection file ="media/roads/lane_withoutmarks.JPG"/>
8  </RoadTextures>
9  <TerrainTextures name="circle_town">
10   <!-- Texture file name for the terrain texture-->
11   <Texture file ="maps/circle_town/texture.jpg"/>
12   <!-- Specifying how often the texture is repeated on the terrain. Only has to
         be changed to a value bigger than one
```

```
13      if no texture for the complete terrain is used. The used texture should be
            seamless then. -->
14    <TextureRepeat value ="300.0"/>
15    <!-- Folder path and file type of the skybox textures.
16      Inside the folder have to be six texture files of the declared file type:
17        1. left.*
18        2. front.*
19        3. right.*
20        4. back.*
21        5. top.*
22        6. bottom.* -->
23    <SkyBox path ="media/terrain/skyboxes/grass_and_hills" type="jpg"/>
24  </TerrainTextures>
```

The last section in a *Map Setup File* changes the terrain construction process by setting dimensions and values for the detail level of the terrain mesh.

Listing B.8: TerrainDimensions

```
1   <TerrainDimensions name="circle_town">
2     <!-- Size of the loaded terrain in meters. The terrain has to be at least big
            enough to load all the street data of the OSM file and to contain all the
            tiles the graphical representation of the terrain is made of (can be set
          in the following values). A Terrain bigger than 3000m*3000m may effect
          long loading times and slow physics. -->
3     <Size width ="1000.0" height="1000.0"/>
4     <!-- Size of one tile of the graphical terrain representation in meters.
          TileSize has to be a multiple of HeightDataPerArea! -->
5     <TileSize value ="64.0"/>
6     <!-- Detail value of the graphic terrain's center tile. (maximum = 7) -->
7     <MaximumLOD value="6.0"/>
8     <!-- Number of levels of detail added to the detail level of the center tile.
            Each additional level of detail will be one step lower. (e.g. the
          simulator will add tiles of detail values 6,5 and 4 if MaximumLOD = 7 and
          LevelsOfDetail = 3)-->
9     <LevelsOfDetail value="3.0"/>
10    <!-- Layers of one level of detail.-->
11    <LayersOfEachLOD value="1.0"/>
12    <!-- Number of meters to the next height data. (in meters per height data
          value). TileSize has to be a multiple of HeightDataPerArea! -->
13    <HeightDataPerArea value="8.0" />
14    <!-- Specifying the size of the steps the graphic terrain follows the camera.
            The value is related to the detail value of a tile. (maximum = 7). (e.g.
          if the value is set to 7 the graphics terrain will always move in steps of
            TileSize/7). CameraStepsPerTile should usually be set to the detail level
            of the lowest detail tile. A value of -1.0 moves the terrain
          simultaneously to the camera and does not make any steps. Unfortunately
          hills will bump up and down with this setting. -->
15    <CameraStepsPerTile value="2.0" />
16    <!-- A heightmap picture contains values from 0 to 255. Those values are
          divided through the HeightDivisionCoefficient to be able to have a
17    different range of heights. (e.g. range of heights for
          HeightDivisionCoefficient value of 10 is from 0.0 to 25.5.-->
18    <HeightDivisionCoefficient value="10.0" />
```

```
19 </TerrainDimensions>
```

B.6 The House File List

The OsmManipulator automatically creates houses or trees along the streets and saves them into the OSM file B.3. The created nodes contain information like the path and type of the 3D model B.3. For adding these kinds of information they have to be teached to the OsmManipulator. This is done by a further XML file, the *House File List*. A new file is created because it is only used during AutoSim world creation. Once the simulator is running no parts are using the file anymore.

The structure of a *House File List* is quite easy. It simply starts with the common XML expression and a new *houses* section:

Listing B.9: File

```
1 <?xml version = "1.0"?>
2 <houses version = "0.01" generator="UWA">
```

After this entering the user can define the many *landuse* sections he wants to. The name of the sections represent the *landuse* they stand for. When the OsmManipulator has decided to create a new house node it searches for a surrounding area defined in the OSM file whose *landuse* tag is matching one of the sections in the *House File List*.

Every *landuse* section contains a list of houses of arbitrary length. Each house is a new section and consists of a name, a model file folder path B.7, a file type and a *BoxSize* in x, y and z-coordinates. The *BoxSize* specifies the representation of the house in the physic world. The y-Coordinate always points up and symbolizes the height. The following listing shows 2 *landuse* section examples *residential* and *forest*, representing that also trees or other static objects can be created in place of houses:

Listing B.10: House List

```
1  <residential>
2    <house name="oldschoolhouse">
3      <File path="models/buildings/oldschoolhouse" type="3ds"/>
4      <BoxSize x="5.0" y="10.0" z="5.0"/>
5    </house>
6    <house name="modernhouse">
7      <File path="models/buildings/residential/2" type="obj"/>
8      <BoxSize x="7.0" y="8.0" z="14.0"/>
9    </house>
10 </residential>
11 <forest>
12   <tree name="tree0">
13     <File path="models/nature/trees/tree0" type="3ds"/>
```

```
14      <BoxSize x="0.5" y="10.0" z="0.5"/>
15    </tree>
16  </forest>
```

B.7 General Model File Information

Storing 3D model files has to be done in a specified way to load them into AutoSim. The center for robot boxes and wheels has to be set in the exact center position of the meshes in order to achieve identical interpretation with the simulator. Static objects like houses must have the center point in the center of their underpart.

In the descriptions for loading houses and vehicles the file path is directing to a folder containing the model files. The structure of these folders must always be as follows: As multiple level of detail versions can be loaded for a model these have to be inserted in folders named *LOD0*, *LOD1*, ... *LODn*. The many of the folders are in here can be decided by the user, as the AutoSimClient always searches these folders. Only *LOD0* has to exist. Inside the *LOD* directory must be the model file named *model.**.

Bibliography

[1] Adrian Boeing, Thomas Bräunl: *SubSim: An autonomous underwater vehicle simulation package.* International Symposium on Autonomous Minirobots for Research and Edutainment, AMiRE 2005:pp. 33–38, Sep. 2005. v, 13, 14

[2] al, Nikolaus Gebhardt et.: *http://irrlicht.sourceforge.net/.* 15, 23

[3] Artificial Intelligence, Dalle Molle Institute for: *http://www.idsia.ch/ juergen/robotcars.html.* 2

[4] Blog, JCWinnie: *http://jcwinnie.biz/wordpress/imageSnag/Stanley_ Image13.jpg.* v, 3

[5] Blow, Jonathan: *Terrain Rendering at High Levels of Detail.* In *GDC*, 2000. 8

[6] Boeing, Adrian: *Evaluation of real-time physics simulation systems.* Technical Report, School of Electrical, Electronic and Computer Engineering, University of Western Australia, 2007. 4, 15

[7] Boer, Willem H. de: *Fast Terrain Rendering Using Geometrical MipMapping.* E-mersion Project, page 7, 2000. 8

[8] Bräunl, T.: *Embedded Robotics - Mobile Robot Design and Applications with Embedded Systems Second Edition.* Springer-Verlag, Heidelberg Berlin, 2006. 13

[9] DARPA: *http://www.darpa.mil/grandchallenge/.* 2

[10] Eckel, Bruce: *Thinking in C++ 2nd edition Volume 1: Introduction to Standard C++*, chapter 16, page 814. Prentice Hall, 1999. 15, 24

[11] Eckel, Bruce: *Thinking in C++ 2nd edition Volume 2: Practical Programming*, chapter 11, page 820. 1999. 6

[12] Erich Gamma, Richard Helm, Ralph Johnson John Vlisside: *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesly, 1994. 6

[13] Ögren, Andreas: *Continuous Level of Detail in Real-time Terrain Rendering.* Master's thesis, Sweden Umea University, 2000. 8

[14] Heckbert, Paul S. and Michael Garland: *Multiresolution modelling for fast rendering.* Proc. Graphics Interface '94, pages 43–50, 1994. 8

[15] Heidelberg, FH: *Aerospace Engineering The Cognitive Autonomous Vehicles of UniBwM.* Elrob, 2007. 2

[16] Huwaldt, J.: *http://homepage.mac.com/jhuwaldt/java/3DStuff/GeoMMap/GeoMMap.html.* 9

[17] Losasso, F. and H. Hoppe: *Geometry clipmaps: Terrain rendering using nested regular grids.* Siggraph, page 8, 2004. 8, 9

[18] M. Duchaineau, M. Wolinsky, D.Sigeti M. Miller C. Aldrich and M. Mineev-Weinstein: *ROAMing terrain: Real-time optimally adapting meshes.* IEEE Visualization, page 8, 1997. 8

[19] Microsoft: *Microsoft DirectX SDK (November 2007) Documentation*, 2007. 47

[20] NASA: *http://nssdc.gsfc.nasa.gov/planetary/mesur.html.* 1

[21] NVIDIA: *Cg User's Manual*, chapter 7, page 356. NVIDIA Corporation, 2005. v, 31

[22] OpenStreetMap: *http://wiki.openstreetmap.org/index.php/Map_Features.* 61

[23] OpenStreetMap: *http://www.openstreetmap.org/.* 10, 60

[24] Peden, Margie: *World report on road traffic injury prevention.* Technical Report, WHO, 2004. 3

[25] Racer: *http://www.racer.nl/.* v, 12, 13

[26] RARS: *http://rars.sourceforge.net/*, 1995. v, 11

[27] Richard H. Bartels, John C. Beatty, Brian A. Barsky: *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling.* Morgan Kaufmann, 1987. 40

[28] Sommer, Torsten: *Physics for a 3D Driving Simulator.* Technical Report, RCS, 2008. 5, 22, 24, 25

[29] Things, The Future Of: *http://64.202.120.86/upload/image/articles/2007/phoenix/mars-pathfinder.jpg.* v, 1

[30] Thomason, Lee: *http://www.grinninglizard.com/tinyxml/.* 16

[31] TORCS: *http://torcs.sourceforge.net/.* v, 12

[32] webpage, ROAM: *http://www.cognigraph.com/ROAM_homepage/.* 8

[33] Wikipedia: *http://de.wikipedia.org/wiki/Spline.* v, 42

[34] Wikipedia: *http://en.wikipedia.org/wiki/Bilinear_interpolation.* 29

[35] Wikipedia: *http://en.wikipedia.org/wiki/Cartesian_coordinate_system.* v, 44

[36] Wikipedia: *http://en.wikipedia.org/wiki/Cubic_Hermite_spline.* v, 41

[37] Wikipedia: *http://en.wikipedia.org/wiki/Kochanek-Bartels_spline.* v, 42

[38] Wikipedia: *http://en.wikipedia.org/wiki/Software_design.* 22