

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



Physics for a 3D Driving Simulator

Torsten Sommer

Bachelor Thesis

Physics for a 3D Driving Simulator

Bachelor Thesis

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

Executed at Robotics and Automation Lab
Centre for Intelligent Information Processing Systems
University of Western Australia
Perth

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

Author: Torsten Sommer
Im Freihöfl 42
85057 Ingolstadt

Submitted on the 1st of March, 2008

Contents

List of Figures	vi
List of Tables	viii
List of Symbols	ix
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	3
1.3 Thesis Outline	4
2 Related Work	5
2.1 Literature Review	5
2.1.1 Thinking in C++ vol. 2	5
2.1.2 Design Patterns: Elements of Reusable Object-Oriented Software	5
2.2 Geometric Data Systems	6
2.2.1 Google Maps	6
2.2.2 OpenStreetMap	6
2.3 Simulators	6
2.3.1 RARS	7
2.3.2 TORCS	7
2.3.3 Racer	8
2.3.4 SubSim	9
3 Robot Simulation Framework	11
3.1 Used Libraries	11
3.2 Software Design Patterns	12
3.2.1 Builder Pattern	12
3.2.2 Singleton	13
3.3 Framework Architecture	13
3.4 OSM Manipulator	15
3.5 AutoSim Client	16
3.6 AutoSim Server	17
3.7 The User Program	18
3.8 Server Software Design	19
4 The Simulation Tree	22

4.1	Hierarchy	22
4.2	Serializable Tree Node	23
4.3	Data Serialization	24
4.4	Data Visualization	26
5	The Robot	27
5.1	Structure	27
5.2	Devices	28
5.3	Noise Models	29
5.4	Construction Process	29
5.5	Programming Interface	32
6	Physics Simulation	34
6.1	Rigid Body Dynamics	34
6.1.1	Definitions	35
6.1.2	Simulation Process	35
6.2	Collision Detection and Response	37
6.3	Terrain and Streets	38
6.4	Bodies and Links	39
6.5	Devices	39
7	Networking	41
7.1	Update Distribution	41
7.2	Remote Sensor Access	42
7.3	Remote Actuator Access	42
8	Conclusion and Future Work	43
A	Code Listings	44
A.1	The User Program API	45
B	Tutorials	47
B.1	The AutoSimServer kick start guide	48
B.2	Working with the AutoSimClient	49
B.3	How to write a User Program	51
B.3.1	Workings of the User Program	51
B.3.2	The User Program API	51
B.3.3	The Client User Program API	52
B.3.4	A Simple Example	52
B.4	Manipulate an OSM file in 6 steps	54
C	The Configuration Files	55
C.1	General Syntax	55
C.2	Customizing a World File	56
C.3	The Robot File	56

C.4	General info on Osm Files	59
C.5	The Map Setup File	60
C.6	The House File List	64
C.7	General Information on Model Files	65
	Bibliography	66

List of Figures

1.1	Vehicle for autonomous mobility and Computer Vision (VaMoRs), 1985, UniBW [17]	1
1.2	Stanley, 2005 Grand Challenge winner from Stanford University [1].	2
1.3	Boss, 2007 Urban Challenge from Carnegie Mellon University [14].	3
2.1	RARS Screenshot [11]	7
2.2	TORCS Screenshot [16]	8
2.3	Racer Screenshot [5]	9
2.4	The Subsim AUV Simulator [13]	10
3.1	Builder Design Pattern	13
3.2	AutoSim Framework Deployment Diagram	14
3.3	The OSM Manipulator	15
3.4	Graphical User Interface of the AutoSim Client	16
3.5	Graphical User Interface of the AutoSim Server	17
3.6	Graphical User Interface of a User Program using FLTK	18
3.7	Internal Structure and Threads of the AutoSim Server	19
4.1	The Simulation Tree	23
4.2	An example Robot description file	23
4.3	Serialization and De-Serialization	25
4.4	Tree View showing the Simulation Tree on the Client	26
5.1	The Robot's Wire Frame and Physics Box	27
5.2	The SimDevice Class and its Members	28
5.3	Generated Gaussian White Noise	30
5.4	Implementation of the Noise Algorithm	30
6.1	Terrain and Street Triangle Meshes in the Physics Engine	38
6.2	Physics Bodies [2]	39
6.3	Links between Physics Bodies [2]	40
6.4	PSD Sensor [2]	40
7.1	AutoSim Client Server Network	42
B.1	AutoSimClient	49
B.2	OsmManipulator	54

C.1 Triangle Fan 62

List of Tables

C.1	highways	59
C.2	landuse	60
C.3	house node	60

List of Symbols

RCS	Chair for Real-time Computer Systems
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
UML	Unified Modeling Language
DLL	Dynamic Link Library
IP	Internet Protocol
TCP	Transfer Control Protocol
PAL	Physics Abstraction Layer
STL	Standard Template Library
UniBw	University of the German Federal Armed Forces, Munich
<i>Robot</i>	<i>Italic font</i> ⇒ class, object or method
A	bold font and captial letter ⇒ matrix
ã	small letter with arrow ⇒ vector

Abstract

Despite the increase in interest by universities and companies in driverless vehicles over the past two decades, the cost and effort involved in their operation still remains at a very high level. However, due to an almost exponential growth of the calculation power of modern computers, and the availability of free high-quality simulation frameworks for both physics and graphics, it is now possible to simulate mobile robots, including their sensors, actuators and environment in real-time. This allows for fast, cost efficient and above all, risk-free research and development of cognitive automobiles.

The purpose of this thesis is to develop a networked simulation environment for autonomous mobile robots. After an overview of the different software components, the internal architecture of the server side is described followed by a closer look at the robots, the physics simulation and the networking aspect. In the appendix a short introduction to the particular programs and their configuration can be found.

Zusammenfassung

Während das Interesse sowohl von Universitäten als auch von Firmen an führerlosen Automobilen in den letzten zwei Jahrzehnten stetig gewachsen ist sind Kosten und Aufwand für den Betrieb eines solchen Fahrzeugs leider auf einem sehr hohen Niveau verblieben. Aber dank der nahezu exponentiell wachsenden Rechenleistung moderner Computer und der Verfügbarkeit von mittlerweile hochwertigen und frei verfügbaren Simulationsumgebungen für Physik und Grafik ist es möglich einen mobilen Roboter samt seiner Sensoren, Aktuatoren und Umgebung in Echtzeit zu simulieren. Dies ermöglicht eine schnelle, kostengünstige und nicht zuletzt risikofreie Forschung an und Entwicklung von cognitiven Automobilen.

Gegenstand dieser Arbeit ist die Entwicklung einer netzwerkfähigen Simulationsumgebung für autonome, mobile Roboter. Nach einer Übersicht über die einzelnen Softwarekomponenten wird der interne Aufbau der Serverseite dargestellt, gefolgt von genaueren Betrachtung der Roboter, der Physiksimulation und des Netzwerkaspects. Im Anhang findet sich eine kurze Einführung in die Bedienung und Konfiguration der einzelnen Programme.

1 Introduction

This introductory chapter tries 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

It all started with the great success of Ernst Dickmann and his UniBw team's 5-ton Mercedes-Benz van 1.1 in the mid 1980s. It was the first vehicle to drive on Bavarian streets - without a driver. Equipped with servo controlled steering, throttle and brakes the car was controlled by a computer, more than 10,000 times less powerful than the ones today, using computer vision to stay on track. In 1994 it was again a UniBw team presenting a robot able to drive the 1678km from Munich to Denmark and back at speeds up to $180\frac{km}{h}$ and automatically taking over other cars.



Figure 1.1: Vehicle for autonomous mobility and Computer Vision (VaMoRs), 1985, UniBW [17]

During the past two decades a lot of money has been spent in this field of research. The first financial boost was the US\$1 billion of funding from the European Commission for

the EUREKA Prometheus Project (1987-1995) aimed at the development of driverless cars. After a few minor milestones the second run of the Grand Challenge, held in 2005, was the next big event in mobile robotics to attract public attention. Sponsored by DARPA and worth US\$2 million in prize money (the largest in robotics history) its task was to complete an off-road course in the Mojave Desert, defined only by 2935 GPS points. Its winner, "Stanley" 1.2, used cameras in combination with GPS and LIDAR¹⁾ sensors to generate a 3D map of its environment, find its way through the course and finally finish the 11.78 km race after 6 hours and 54 minutes.



Figure 1.2: Stanley, 2005 Grand Challenge winner from Stanford University [1].

The 2007 Urban Challenge finally took the competition to the city where the competitors had to make their way through an obstacle-packed 96 km urban area course while obeying all traffic regulations and dealing with the traffic. The two million dollars this time went to the Tartan Racing Team based at the Carnegie Mellon University with their Chevy Tahoe named "Boss".

But how can the average researcher continue to take advantage of the recent achievements in the mobile robotics sector without spending enormous amounts of money and manpower on the development and operation of such a vehicle? The solution to this problem is simulation which is not only more efficient in terms of cost and time, but also allows for experiments that are not even possible in the real world. Cars can be damaged, buildings can be constructed and demolished, and even sensors, worth thousands in the real world, can be simulated for the price of a piece of CPU time.

¹⁾ LIDAR (Light Detection and Ranging) is an optical remote sensing technology that measures properties of scattered light to find range and/or other information of a distant target. [19]



Figure 1.3: Boss, 2007 Urban Challenge from Carnegie Mellon University [14].

But it is not only the price inflation of calculation power that fuels the development of simulators. It's also the availability of realtime physics and graphics engines allowing a realistic simulation and visualization of the robot's behavior and interaction including sensors and actuators. Inspired by the great success of the SubSim project, hosted by CIIPS, that aimed at simulating autonomous underwater vehicles (AUVs) it was our idea to do the same for driving robots - a simulator that provides a wide range of sensors and allows researchers to build their custom robots inside the simulation platform and to develop control programs processing the sensor data and setting the actuators. In addition, these programs can be adopted for real robots - maybe even in a competition.

1.2 Objectives

The purpose of this thesis is the development of a robot simulation environment that is entirely based on open source software. Its main objective is to provide the user with a framework that is capable of simulating multiple robots in real time using a physics engine. This framework is split up into 3 major components: A server program running physics and a client to visualize the scenery, both connected through an IP based network, as well as a set of tools and configuration files used to generate and manipulate the simulated world.

A whole simulated world can be generated out of geographic data from the Open Street Map project that supplies information about streets, buildings and land use. Also, real world height data can be used to model the terrain.

All robots in the simulation are controlled by small programs that are being separately

compiled and dynamically loaded during runtime. These control programs are provided with a wide range of virtual sensors and actuators to interact with their environment and therefore enable the user to easily develop and immediately test complex programs on inexpensive standard PCs.

To increase the ease of use all programs contained in the framework feature a graphical user interface that allows for simple access to the most common parameters. All details of the simulated world and especially the robots and their configuration can be customized by editing the corresponding configuration files based on the industry standard XML format.

1.3 Thesis Outline

Chapter 2 gives an overview over work related to programming and simulation of robots and provides a listing of free, state of the art driving and robot simulators followed by an introduction to the simulation framework, its architecture, used libraries and deployed design patterns and last but not least a description of its components. The next two chapters cover the internal structure and representation of the simulated scenario with a focus on the robots (5) forming the heart of the simulation. In 6 a short introduction to the inner workings of the underlying physics engine is given and chapter 7 deals with the networking aspect of the simulation. Finally chapter 8 concludes the paper by pointing out the achieved goals and showing some perspectives for future development.

2 Related Work

This chapter presents an assortment of projects related to AutoSim and gives a short overview over the literature that proved helpful for engineering the framework's structure and implementation.

2.1 Literature Review

2.1.1 Thinking in C++ vol. 2

Thinking in C++ vol. 2 [4] by Bruce Eckel and Chuck Allison covers the Standard C++ classes and other advanced topics, a.o. the analysis and design fundamentals, flow control, data types, casting, design of reusable C++ classes, abstract classes, templates and iterators.

The emphasis is on practical programming and therefore all covered topics are accompanied by dozens of helpful code examples.

Its detailed description of advanced software design patterns as well its introduction to the most common features of the STL and their application proved to be good value during the design and implementation process of the AutoSim framework.

2.1.2 Design Patterns: Elements of Reusable Object-Oriented Software

Design Patterns: Elements of Reusable Object-Oriented Software [?] 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 to get 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.

2.2 Geometric Data Systems

In order to create a realistic virtual world inside the simulation including streets, houses and terrain external information is required which can be obtained from a number of internet based geometric data systems. Two of them are described in this section.

2.2.1 Google Maps

Google Maps is a free web mapping service application and technology provided by Google that powers many map-based services including the Google Maps website, Google Ride Finder and embedded maps on third-party websites via the Google Maps API. It offers street maps, a route planner and satellite images for numerous countries around the world. Unfortunately the data used in Google Maps is not free and cannot be used by other programs [18].

2.2.2 OpenStreetMap

OpenStreetMap is a collaborative project to create free editable maps. The maps are created using data from portable GPS devices and other free sources. The data is published under a license that allows it to be used in other open source projects. Like other mapping services OpenStreetMap offers rendered Map images to plan a route or just print a part of the map. Everybody can take part in the creation and modification of street data. The website offers free editors to make it easy to download and change the current data. If two users uploaded the same modifications of a map, these would be saved in the OpenStreetMap database and published on the server. A description of the data structure is available to use the data descriptions inside a project [10].

2.3 Simulators

The following four sections give an overview of the most important free racing and car simulation projects. A closer look is then taken at the intended audience of the program, the networking capabilities and the underlying physics system that is crucial for the use as a simulator.

2.3.1 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 project's website [11].



Figure 2.1: RARS Screenshot [11]

Even though RARS meanwhile has a 3D graphics system, it aims at visualizing a two dimensional race track system which is also reflected by the physics system that hasn't undergone a reconstruction towards a real 3D simulation. Also the physics engine is kept fairly simple and therefore allows only for very few sensor data to be obtained and processed from within the robot's control program - it aims at programmers that write AI programs for computer games rather than real world simulation software.

2.3.2 TORCS

TORCS, The Open Racing Car Simulator, is a racing simulator allowing users to drive races against computer controlled opponents and to develop their own AI controlled

robots. The concept of TORCS is derived from RARS [11], but allows the user to control one of the robots by an input device like a keyboard, a mouse or a steering wheel. It also features a powerful rendering system, capable of visualizing lighting, smoke, skid marks and glowing brake disks. The physics engine of the simulator includes a simple damage model, collisions and copious car properties. The gameplay allows different types of races from simple practice sessions to whole championships. 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 2.2: TORCS Screenshot [16]

Similar to RARS, TORCS is intended for driving on a race course, rather than in a city environment and therefore the physics engine is optimized for a race-like feeling rather than realism. Furthermore TORCS doesn't provide networking which means that challenges between robots have to be simulated on a single machine.

2.3.3 Racer

Racer is a car simulation project, using real car physics to get a realistic driving feeling. Considering that it is open source it features an impressive OpenGL based rendering system. Racer's graphics engine is capable of displaying smoke, skid marks, sparks, sun, flares and vertex-color lit tracks. It also attempts to do well at the physics section, trying to create life-like cars emphasizing car control. The package is available for Windows,

Linux and MacOS X platforms.



Figure 2.3: Racer Screenshot [5]

However it aims mainly at arcade game fans and people testing race car physics rather than simulating a city environment including sensors and actuators. Also the program is not networked which limits the whole simulation and control programs to a single machine. The biggest downside to the software is that it's still subject to copyright even though the source code can be obtained from the author's website.

2.3.4 SubSim

SubSim is an autonomous underwater vehicle (AUV) simulator allowing researchers to develop autonomous software without the need for a physical AUV. Application design, controller tuning, mission simulation, and fault-tolerance can all be tested with the simulator.

The primary audience is universities, schools and other educational institutions interested in simulating AUVs. The SubSim program was developed for the an international AUV competition, to be held in Perth, Western Australia, that, unfortunately, never took place.

The program features a three dimensional dynamic simulation of an AUV and the external environment using the PAL, a C compliant API and a debugging console window as well as customization of the simulation parameters, AUV and environment through XML files.

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 simulated scenario is loaded as a static 3D file and the visualization features are rather poor due to the underlying graphics engine.

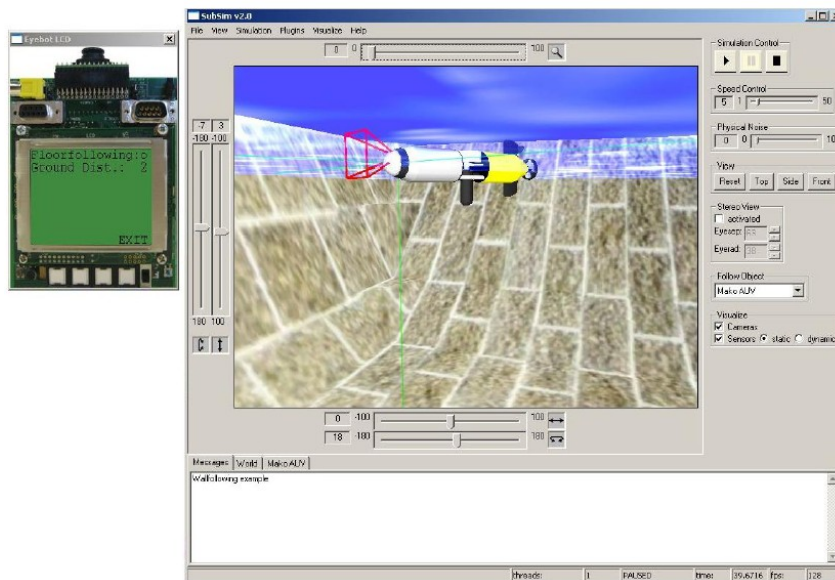


Figure 2.4: The Subsim AUV Simulator [13]

3 Robot Simulation Framework

This chapter gives an overview of the libraries and design patterns used in the AutoSim project followed by a description of the frameworks architecture, its applications and their interaction.

In contrast to most other simulation systems such as the ones described in the previous section, AutoSim does not focus on gaming but provides a full blown rigid body physics simulation, capable of simulating all kinds of interaction as well as sensors and actuators. It has also been taken into account from the very beginning of the design process that the framework needs a sophisticated representation of the simulated world that is versatile, easy to extend and whose information can be efficiently transferred to connected systems over a network.

Furthermore the rendering, simulation and physics system are loosely coupled to be able to change between different graphics and physics engines or even client programs. In addition all components are generally portable to "unix like" systems (like Linux) since all used libraries are available for these platforms too.

3.1 Used Libraries

TinyXML: TinyXML [15] 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 be read, modified and saved. It also allows the user to construct own XML documents with C++ objects and write these to the hard disk or another output stream. As the name implies, 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.

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 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 which additionally contributes to speed and bandwidth efficiency.

PAL: The Physics Abstraction Layer (PAL) provides a unified interface to a number of different physics engines allowing 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 benchmarking suite for evaluating and visualizing dynamic simulation systems.

3.2 Software Design Patterns

3.2.1 Builder Pattern

The goal of Builder (which is a creational pattern, like the Factories) is to separate the construction of an object from its "representation." This means that the construction process stays the same, but the resulting object has different possible representations. The main difference between Builder and Abstract Factory [4] is that a Builder creates the object step-by-step, so the fact that the creation process is spread out in time is important. In addition, the "Director" gets a stream of pieces that it passes to the Builder, and each piece is used to perform one of the steps in the building process. This design principle is of enormous importance for the load process of the simulation since it ensures that the data structures that are being used for serialization and de-serialization (see chapter 4) are exactly the same on all platforms.

Figure 3.1 shows the workings of the Builder design pattern. The simulation loading process is described in detail in chapter 5.4.

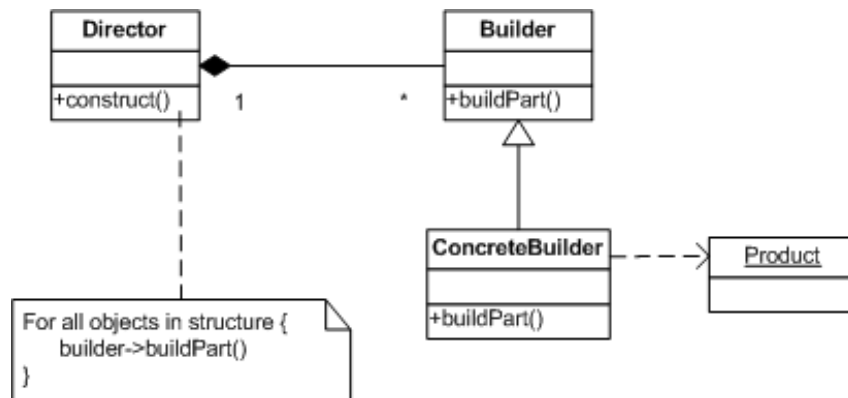


Figure 3.1: Builder Design Pattern

3.2.2 Singleton

The Singleton pattern is implemented by creating a class with a method that creates a new instance of the class if one does not exist. If an instance already exists, it simply returns a pointer to that object. To make sure that the object cannot be instantiated any other way, the constructor is made either private or protected. There is a distinction between a simple static instance of a class and a Singleton: although a Singleton can be implemented as a static instance, it can also be lazily¹⁾ constructed, requiring no memory or resources until needed. Another notable difference is that static member classes cannot implement an interface, unless that interface is simply a marker.

The Singleton's thread-safe implementation of the creation method ensures that different threads can access the exact same object which is created on the heap on the first invocation even if two threads are to execute the creation method at the same time when the Singleton does not yet exist. This is achieved by employing the concurrent processing capabilities of the compiler to make the construction a mutually exclusive operation.

In the case of AutoSim this design pattern is particularly useful to grant access to dynamically generated objects from within static methods and C functions such as API functions and callbacks from the GUI.

3.3 Framework Architecture

The AutoSim environment is a fully networked 3D robot simulation software that is entirely based on open source libraries and is split up into 4 components:

¹⁾ which means at the first time someone tries to access the object

- **AutoSim Server:** The server application runs the actual simulation hosted in the underlying physics engine and the server based user programs.
- **AutoSim Client:** The client visualizes the data received from the server and runs the client side user program.
- **User Program:** The user program is the instance that controls every robot. It uses an API to read and write data from and to the virtual devices of a simulated robot.
- **OSM Manipulator:** It is used to automatically add objects, like plants, signs and buildings, to the OpenStreetMap files using separate definition files.

The UML Deployment Diagram 3.2 shows the interaction between the server, client and user program components at runtime.

The clients connect to the server through a TCP/IP network from which they obtain updates on the simulation status including all positions and sensor data. The user programs connect to either the client or the server through an API. They're dynamically loaded as DLL's and run in a separate thread. The client-side User Program differs from that of the server-side in two ways: All actuator access is encapsulated into network packets and transferred to the server and secondly it can make use of the Client User Program API which provides extended functionality that requires the rendering system (which is not available on the server). The OSM Manipulator is not shown in the diagram since it is a tool used to modify the map files and isn't of any importance during the runtime of the simulation.

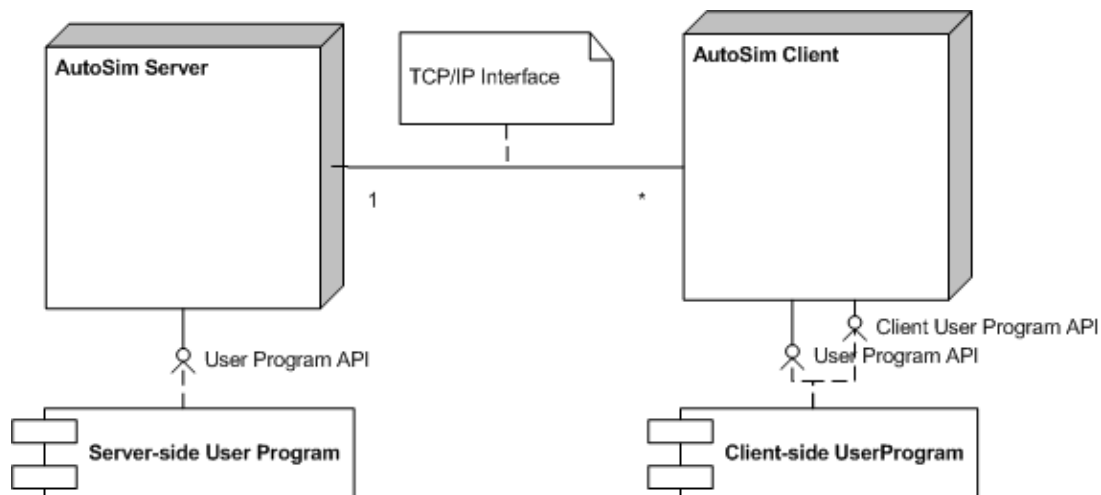


Figure 3.2: AutoSim Framework Deployment Diagram

In the next four sections all of the frameworks components are described in detail followed by a deeper insight on the server's internal software design.

3.4 OSM Manipulator

Figure 3.3 shows the user interface of the OSM Manipulator program.

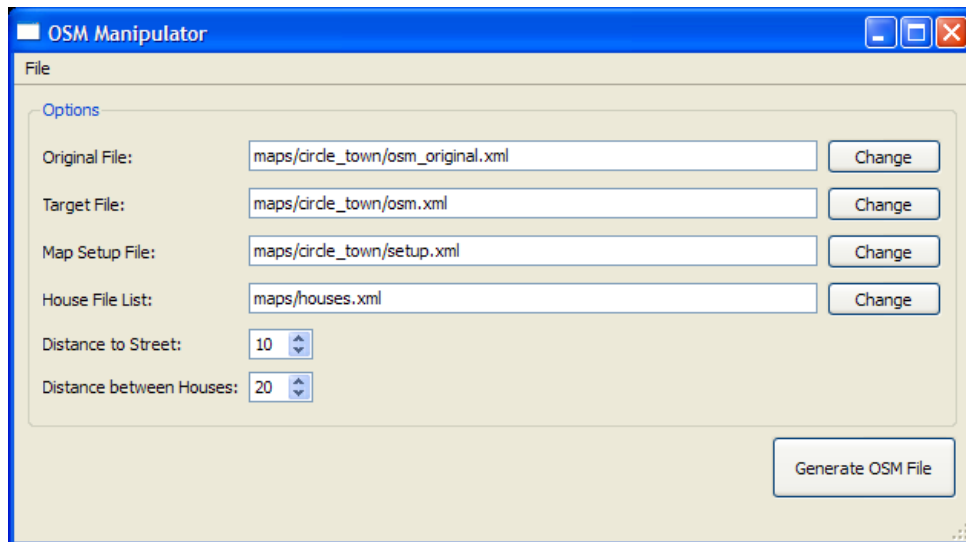


Figure 3.3: The OSM Manipulator

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 using separate configuration files containing the file paths to the models and other necessary information like distances between the houses and the street.

Inside the program 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 newly created file
- a map setup file containing the size information for all road types
- a house file that provides the data for the houses (a.o. size, the graphics model file path and type)

In addition two input fields allow the user to customize the distance of a house to the street as well as to its neighbor houses. After selecting all input and output files, the generation process is started by pressing the *Generate OSM File* button.

3.5 AutoSim Client

The AutoSim client provides the user with an easy to use graphical interface to load, run and debug the client simulation part. Figure 3.5 shows the GUI of the AutoSim client.

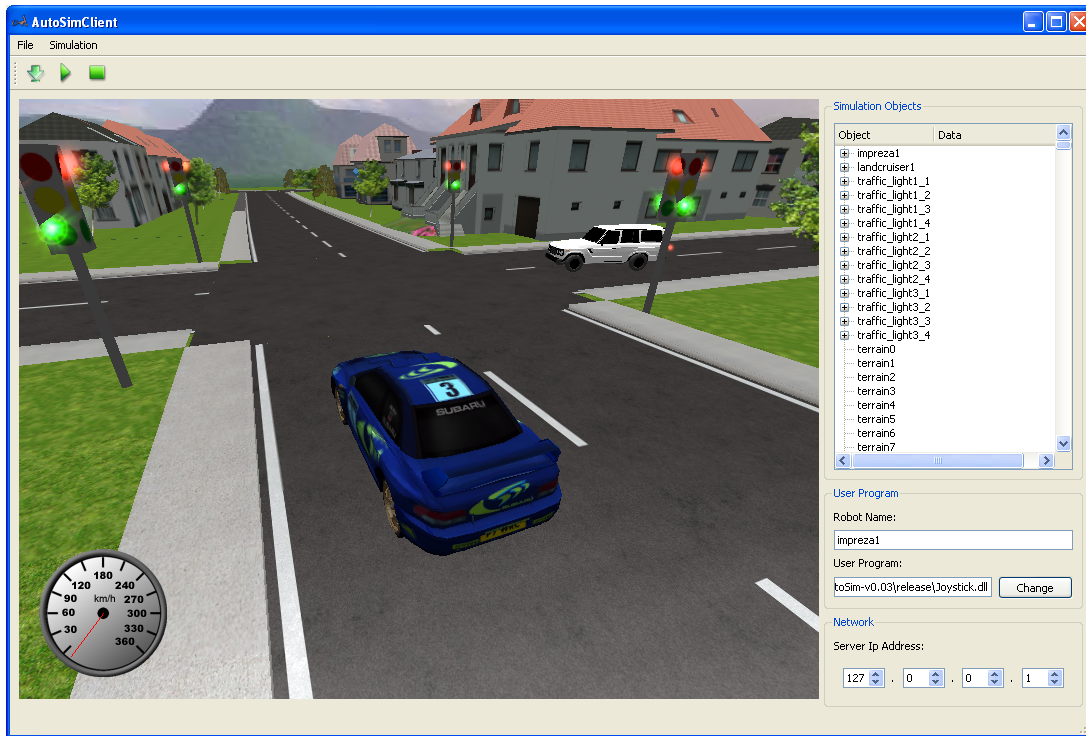


Figure 3.4: Graphical User Interface of the AutoSim Client

On the top of the GUI the user can access a menubar to select the world file and User-Program to load. 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 click on the load button to the top left loads the graphic world. In the Simulation Objects box of the interface appears a scrollable tree view of the simulation objects after the 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 that's being connected to the robot specified in the "Robot Name" text 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.

3.6 AutoSim Server

The AutoSim server is running the physics and simulating all interaction between the simulation objects and therefore represents the central instance of the framework at runtime. 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 of the step size for the physics engine. The following figure shows the AutoSim server after a simulation has been loaded.

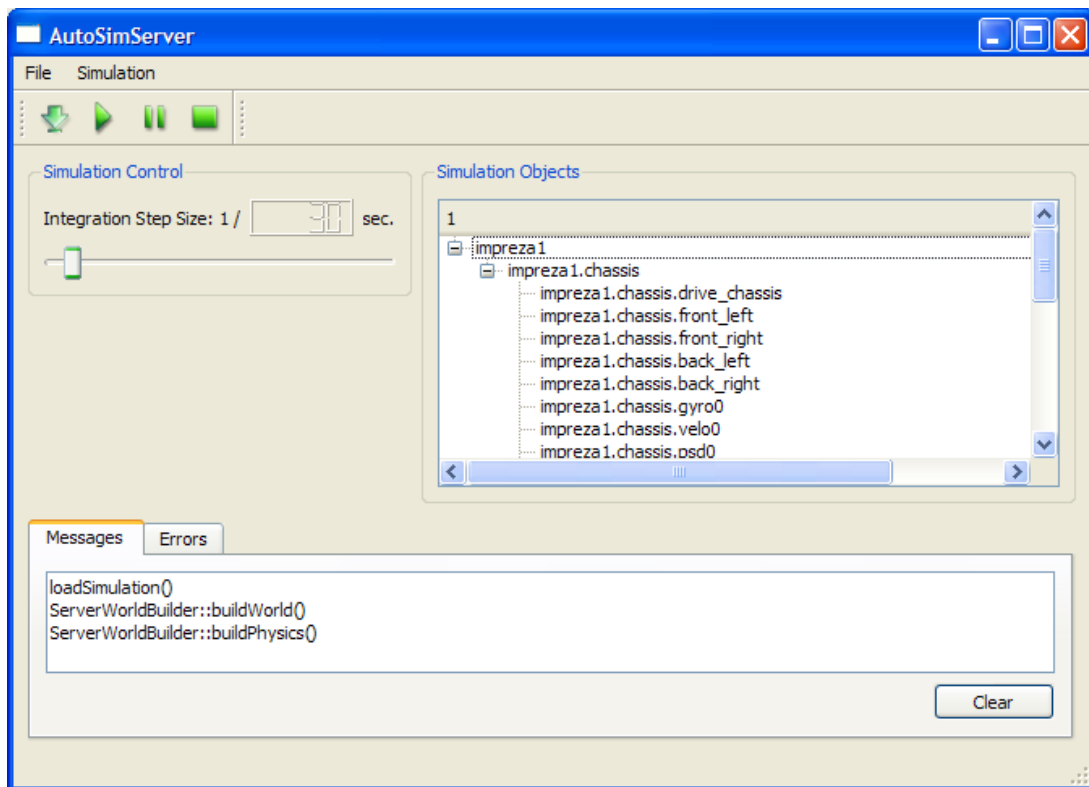


Figure 3.5: Graphical User Interface of the AutoSim Server

On top you find buttons to load, start, pause and stop the simulation process. To the left 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 reveal problems caused by corrupted configuration files. The structure and loading process of the simulation tree and in particular the robots are covered in detail in the following chapters.

On the bottom of the window two boxes display all messages and warnings generated during the loading 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.

3.7 The User Program

User Programs are the most interesting part of the framework to the user since they represent the control instance of every Robot inside the simulation. They consist of a more or less complicated C/C++ that is being compiled as a DLL and dynamically loaded by either the Server or Client.

It can access all devices of the robot it belongs to through the User Program API to read data from sensors, process them and control the actuators. However the User Program is not limited to control a single robot - it is possible to access devices on remote robots which is particularly interesting if e.g. positions of other robots are needed to facilitate obstacle avoidance.

The following figure shows a User Program that uses a graphical user interface based on FLTK to turn the rear and head lights on and obtain images from various virtual cameras and display them in the upper part of the window.

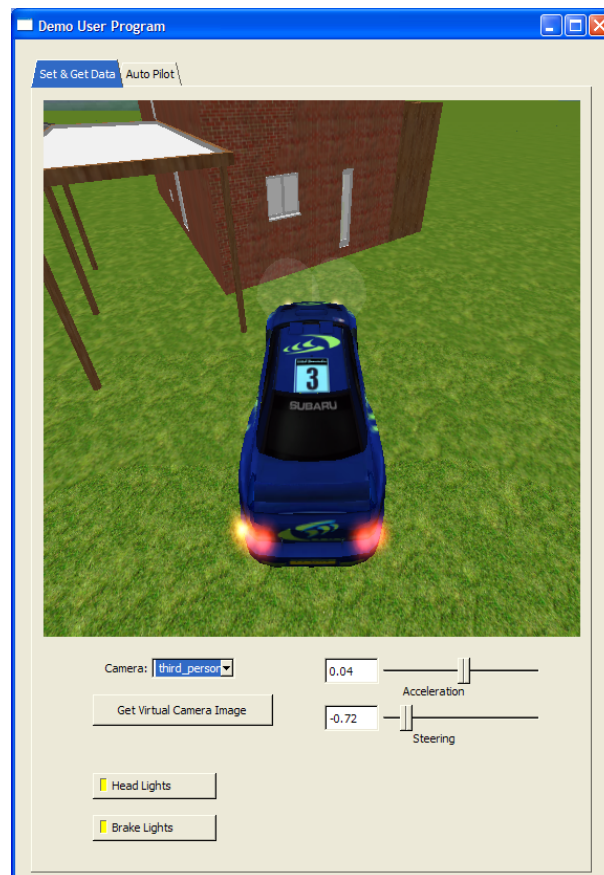


Figure 3.6: Graphical User Interface of a User Program using FLTK

There are two kinds of User Programs: One that uses only the functions provided by the User Program API and underlying operating system as well as additional C and C++ libraries. These programs can access all devices except cameras and can be run on both server and client.

The second type of user program can only be run on the client and uses the Client User Program API to access and process virtual camera images generated by the rendering system of the client program. Since these programs are compiled and loaded as dynamic libraries they're not subject to any limitations concerning complexity, multi threading or graphical user interface. They can easily be exchanged without forcing the user to recompile the whole framework or to change the configuration files.

3.8 Server Software Design

When designing AutoSim the main requirements such as the networking, portability and ease of use have been taken into account from the very beginning of the process. This has resulted in a piece of software whose architecture is small and simple but yet powerful, extensible and easy to understand.

The static structure diagram 3.7 below shows the basic workings of the different objects and the threading of the AutoSim Server and server-side user programs. The AutoSim Client has a similar architecture which is covered in detail in Johannes Brand's thesis "Graphics for a 3D Driving Simulation" [3].

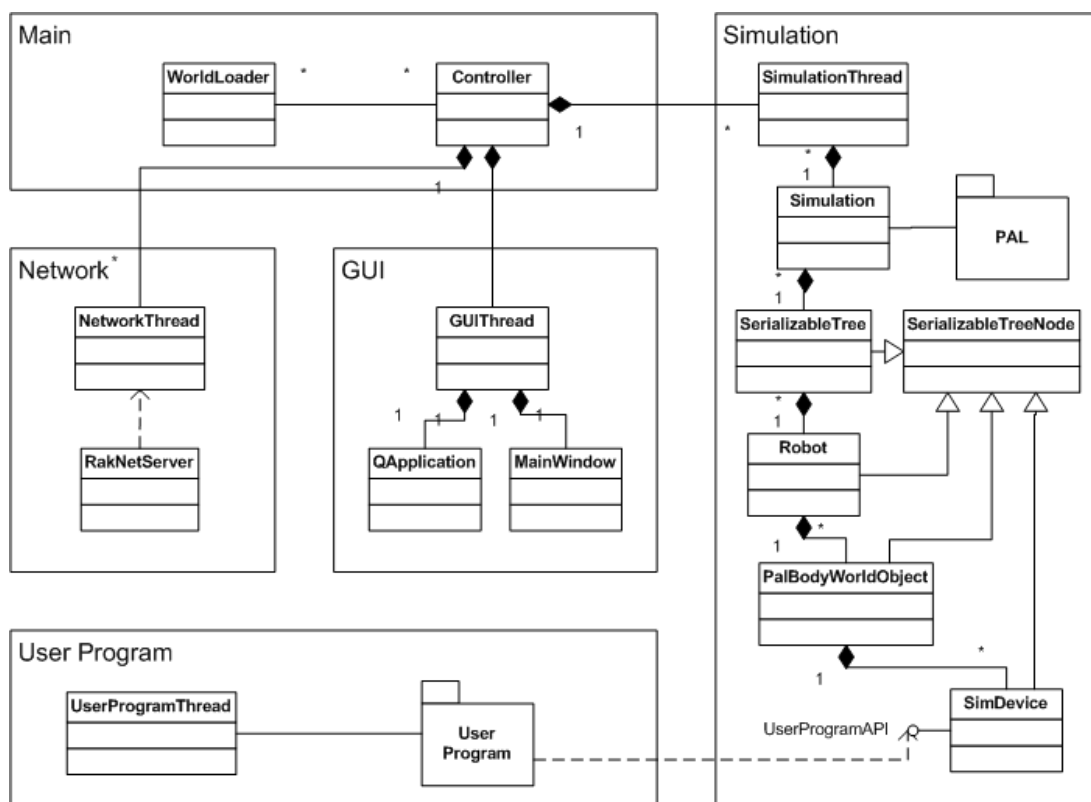


Figure 3.7: Internal Structure and Threads of the AutoSim Server

The *Controller* is the central instance of the server package policing all interactions between the different subsystems. It is implemented as a Singleton in order to be accessible across all threads and from within static function blocks like API calls and callbacks from the graphical user interface.

Apart from the internal API it provides to all other subsystems to influence the simulation status and its parameters, such as the integration step size, it is also in charge of the loading process since it possesses the *WorldLoader* object, which for its part generates the Simulation Tree and is described in chapter 5.4.

Even though the lifetime of the *Network* thread is controlled by the Controller instance the networking subsystem runs to a great extent autonomously. Once it is loaded and the simulation process has been started it delivers updates to the connected clients and accepts and processes packets from the remote user programs containing the actuator access information. For a closer look at the workings and implementation of the network go to chapter 7.

The *GUIThread* is the first instance to be loaded when the server application is executed since it provides the user interface which controls all operations. It hosts two objects: the *QApplication* that is part of the Qt environment and contains the main event loop, where all events from the window system and other sources are processed and dispatched. It also handles the GUI's initialization and finalization, and provides session management.

Secondly there is the *MainWindow* that represents the server applications' main window, including all user interface components such as buttons, sliders and switches as well as menus and file dialogues. It is automatically generated during the building process from a *.ui file that can be created using the *designer* shipping with the Qt Framework. These *.ui files conform to the XML standard and describe all parts of the user interface, its components and last but not least their alignment and interaction. E. g. a LCD²⁾ widget can be used to display the current value of a slider position without requiring any hand written C++ code and therefore simplifies the implementation process.

The *Simulation* is created during the loading process described in chapter 5.4 and reflects all objects that are simulated inside the physics engine and also runs the simulation loop that ensures time synchronization. All objects contained in the simulation are organized in a tree structure which is achieved by the fact that every object in the simulation tree inherits from the *SerializableTreeNode* class that provides the necessary infrastructure to organize the objects and a unified interface for the serialization. This tree includes the Robots, their parts (e.g. chassis and trailer) and the devices attached to the parts.

All *Robots* that are not intended to be controlled by a client, such as traffic and traffic

²⁾ simple, calculator like Liquid Crystal Display

lights, are controlled by a user program that runs directly on the server. This control program is dynamically loaded during the load process, runs in a separate thread, the *UserProgramThread*, and communicates with the simulation through the User Program API [5.5](#).

4 The Simulation Tree

In designing the data structure of AutoSim and therefore the kind of representation of all objects simulated, the following criteria had to be met. It needed to be flexible and extensible without limiting the depth of the structure; capable of reflecting the hierarchy of the simulated world, easy to access, and able to quickly, simply and efficiently serialize all data of the whole tree or any sub-tree.

The next sections give an overview of the hierarchy used in both the server and client in order to represent the different simulated objects, followed by a detailed description of the underlying mechanisms used to make up the tree, find and access nodes, serialize and de-serialize parts of the tree and last but not least visualize it in the graphical user interface.

4.1 Hierarchy

Structuring the simulated world in a tree structure offers a number of advantages over a flat or even uncorrelated representation. It is predestinated to be described by XML files which provide an industry standard configuration method to the user that can choose from a variety of programs in order to edit these files. Moreover it greatly simplifies the loading process since the structure can be taken over from the configuration files which represent a tree structure for their part. Lastly this structure provides a good foundation to display the data that is stored in and generated by its nodes in classified order.

The following figure [4.1](#) shows the basic structure of a simulation tree on the server. In contrast to the client, all static objects, i.e. all objects that don't actively interact in the physics and therefore don't have variable position or state data like houses, streets and the terrain are not represented. However due to the flexibility of the concept they could easily be integrated during the loading process should the need arise.

To the right an example configuration file [4.2](#) is shown that defines a robot consisting of a single box and a single sensor attached to it.

On top of the tree sits a *SerializableTree* node that forms a root node and access point to the tree by implementing two additional methods in addition to the ones inherited from the *SerializableTreeNode* which are being discussed in chapter 4.2:

Given a *SerializableTreeNode*'s unique name the *findNode* method returns a pointer to this node object and therefore provides a very easy and intuitive way to find and access nodes. The *getAbstractItemModel* method returns a pointer to a *QAbstractItemModel* that forms an interface for the visualization in the GUI 4.4.

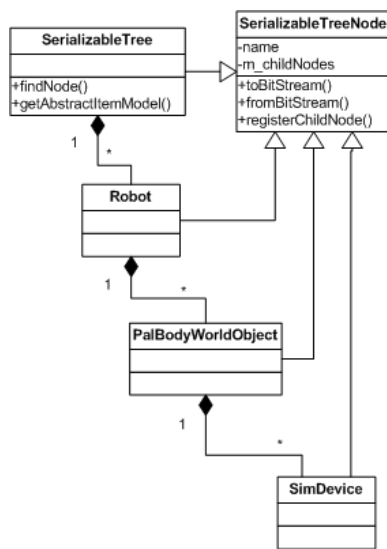


Figure 4.1: The Simulation Tree

```

1  <?xml version = "1.0"?>
2  <Robot name = "impreza">
3    <Parts>
4      <Box name="chassis">
5        <Position x="0.0" y="0.0" z="0.0" />
6        <Size x="2.0" y="1.3" z="4.8" />
7        <Mass value="1400.0" />
8        <CenterOfMass x="0.0" y="0.37" z="-0.76" />
9        <Model path="models/impreza/chassis" type="
10         3ds" />
11      </Box>
12    </Parts>
13    <Devices>
14      <PSDSensor name="psd0">
15        <Part name="chassis" />
16        <Direction x="0.0" y="0.0" z="1.0" />
17        <Position x="0.0" y="0.0" z="0.0" />
18        <Range value="10.0" />
19      </PSDSensor>
20    </Devices>
  </Robot>
  
```

Figure 4.2: An example Robot description file

Listing 4.2 shows the close relation between the simulation tree and its configuration files: Every node has its origin in an XML tag describing it - in this example it is a robot named "impreza" that consists of a single box, the "chassis" with only one PSD sensor device attached to that box. However the simulated robots may have an arbitrary number of bodies, links between those bodies as well as a vast number of devices - the complexity is only limited by the hardware that runs the physics engine.

4.2 Serializable Tree Node

The *SerializableTreeNode* class provides the whole functionality that is needed to become a member of a tree and is inherited by all classes on both server and client that form the respective simulation tree.

When a new *SerializableTreeNode* is registered with a node that is part of a tree by calling the *registerChildNode* method, the parent node stores a pointer to it in its internal

`m_childNodes` vector. After that it concatenates its own name, a "." and the child nodes original name and sets it as the new name value for the child node. Finally the pointer to the newly registered node is recursively passed on by calling *notifyChildNodeRegistered* until it reaches the root node which has an overloaded version of the method: it stores the nodes name and pointer to it in a map.

Whenever a part of the simulation needs to access a node, e.g. a sensor, it calls the root node's *findNode* function supplying it with the fully qualified name of the node, which is then looked up in that very map.

The name for the PSD sensor in this example would be "impreza.chassis.psd0" where "impreza" is the name of the robot, "chassis" the name of the part and "psd0" the name of the device.

Even though the use of strings as identifiers requires slightly more CPU power they have a number of advantages over pointers or numbers. Firstly they provide the device name in a human readable format which is great value when debugging. Also there is no need to initialize devices or request identifiers during runtime. All device names can be directly coded into the user program or even be read from external files using all the functionality provided by the *string* class. Another benefit is the point notation giving the user information on the location of the object inside the tree.

4.3 Data Serialization

One of the greatest achievements of the tree structure used for this simulation is the fact that its graph contains neither cycles ¹⁾ nor joins ²⁾. That means that the whole tree as well as any subtree ³⁾ can be serialized using a recursive algorithm.

This is achieved by the *toBitStream* and *fromBitStream* methods of the *SerializableTreeNode* class. Both methods are virtual and provide a default implementation that simply calls the *to* or *fromBitStream* methods of all child nodes.

If a node contains data members or in the case of the server generates status data e.g. its position, the node's implementation overloads these serialization methods in order to write that data into the provided bitstream before calling the corresponding functions on its children.

¹⁾ A join in a graph means there are two or more distinct paths to the same object [9]

²⁾ A cycle in a graph means there is a path from an object back to itself [9]

³⁾ A subtree is a portion of a tree data structure that can be viewed as a complete tree in itself. Any node in a tree T, together with all the nodes below it, comprise a subtree of T. [21]

Figure 4.3 visualizes the serialization and de-serialization process: It starts by the invocation of `toBitStream` of "Node" (step 1). After appending its data members (in this case only the name "Wheel" followed by the string terminator) to the bitstream it calls `toBitStream` for its child nodes "Child Node 1" (2) and "Child Node 2" (5). `ToBitStream` returns after the respective `toBitStream` methods of all children have been called. This process is recursively repeated until the "Grand Child Node" has added its information to the bitstream and returned.

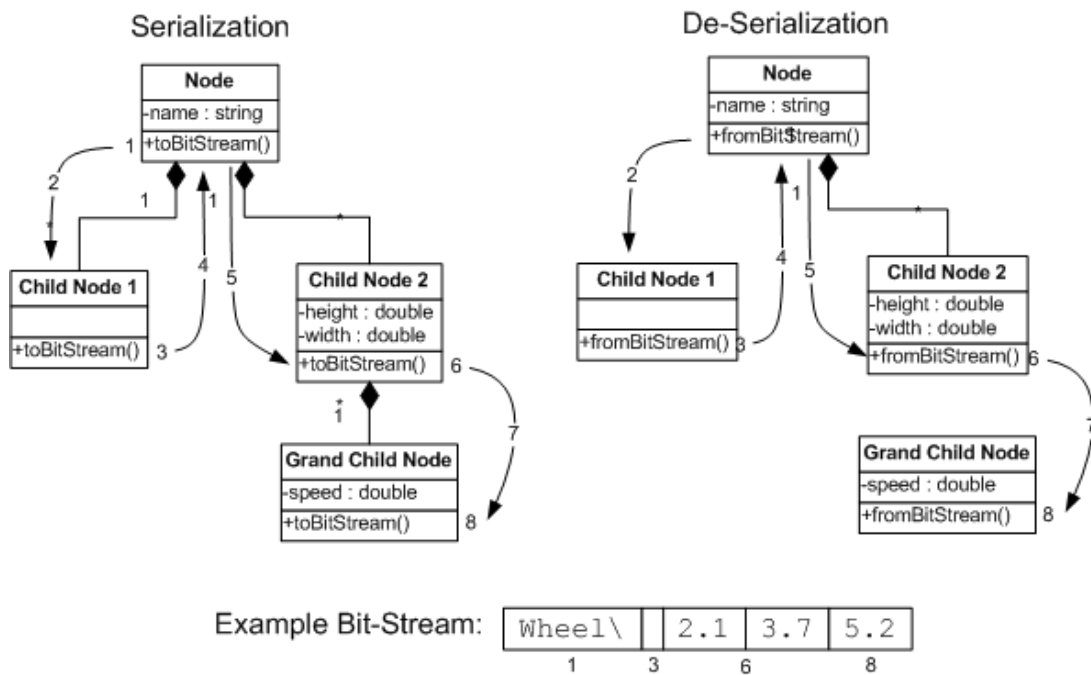


Figure 4.3: Serialization and De-Serialization

The de-serialization on the clients side happens in a similar way: the corresponding root node of the subtree that has been serialized on the server is provided with the bitstream containing the status data. After "cutting out" all bytes intended for this node the remaining bitstream is passed on to its child nodes.

The construction process of the simulation tree discussed in 5.4 ensures that the trees on the server and all clients match and therefore neither leftover nor missing data bytes in the stream can occur since every node reads exactly the data that has been inserted on the other side in the exact same order.

This serialized simulation status is particularly useful for transmission over a network or to be stored for monitoring purposes. Another feature of the discussed serialization method is the fact that the byte size of the bitstream⁴⁾ represents the theoretical optimum⁴⁾ since no meta information is being stored.

⁴⁾ assuming that the status data cannot be compressed

4.4 Data Visualization

The `SerializableTreeNode` provides two ways to visualize the data that is stored inside its data members. The first one is the recursive `print` function that can be invoked for an arbitrary subtree just like with the serialization. In its standard implementation it simply prints its name and calls the print function for its child nodes. Its behavior can be customized by overloading it in order to get more information. This way of visualization is meant for debugging and aims at command line or file output.

The Second way to display the three is to use the `getAbstractItemModel` method which returns a pointer to an `QAbstractItemModel` providing an interface for the `QTreeView` widget to visualize the data from the tree. This enables the user to access all position and sensor data at runtime in a hierarchical order which is especially useful when debugging user programs or to find syntax errors and misconfigurations in the world and robot description files.

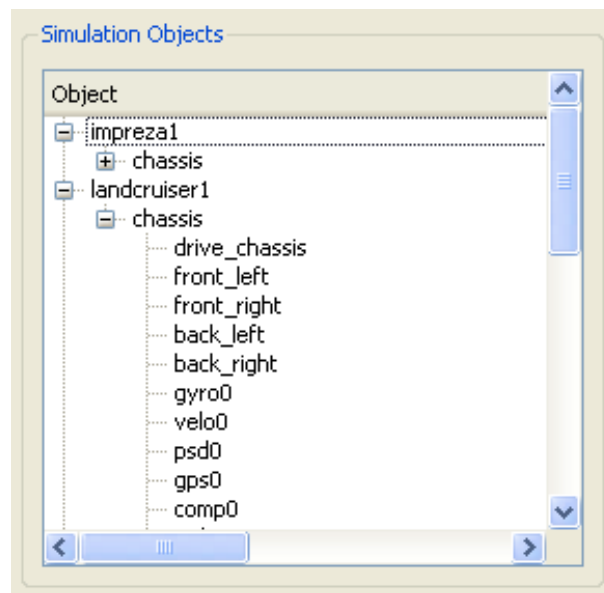


Figure 4.4: Tree View showing the Simulation Tree on the Client

Figure 4.4 shows an example tree on the client visualized in the `QTreeView` widget using the abstract item model provided by the `SimulationTree` object. This way of displaying the data in the simulation tree not only provided a well ordered structure but also turned out to be of great value when debugging and testing user programs.

5 The Robot

The Robot is the heart of the simulation since it provides the whole infrastructure that allows one to access sensors and control actuators from within the user programs and represents the bridge to the simulated world inside the physics engine. The following sections give a brief overview over the structure of a Robot, the available Devices and noise models, the Robot's construction process and finally its programming interface, providing a unified way to access all devices.

Figure 5.1 shows the "two-sided life" of the robot. On the client it is represented by a highly detailed graphics model whereas the box approximating the models shape that surrounds it reflects its greatly simplified physics model.

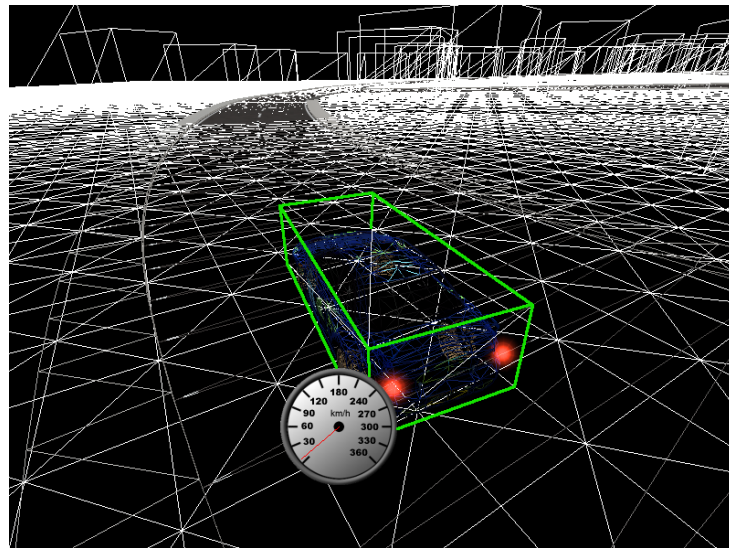


Figure 5.1: The Robot's Wire Frame and Physics Box

5.1 Structure

As shown in figure 4.1, the Robot class forms the root of the robot subtree. Its child nodes are the objects representing the various parts belonging to the robot such as boxes, spheres and compound bodies (imaged in 6.2,6.3). Again every part can have an arbitrary

number of devices attached to it.

This structure groups all the devices simulated by the physics engine and provides an uniform way to access them. However it is not limited to devices that have a match in the physics world. E.g. a radio device for communication between robots could easily be integrated should the need arise.

5.2 Devices

All interaction between the various user programs running on the client and server happens through the devices that form the bridge to the simulated physics world and other parts of the simulation. These devices provide sensor data such as velocities, distances and angles as well as meta information e.g. the simulation time. Another group of devices, the actuators, take input from the user program and influence the physics by generating forces and impulses. However a device can be a sensor and actuator at the same time.

As an example the *DriveActuator* device is used to set the steering angle, acceleration and brakes for a vehicle. Internally these parameters are passed on a complex vehicle model based on ray casts which is provided by the PAL. One can think of it as a body (e.g. a box) "flying" on these rays. All forces on the body as well as the position and orientation of the wheels are then calculated from the measured distances to the ground and the body's velocity and location.

The class diagram in figure 5.2 gives an overview of the internal structure of a *SimDevice* which is inherited by every concrete device. It provides default implementations for the access methods and a noise generator that is covered in section 5.3.

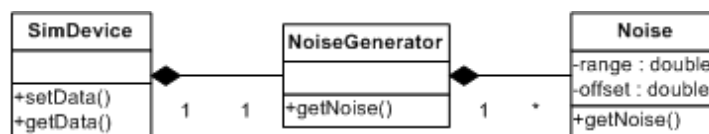


Figure 5.2: The SimDevice Class and its Members

The following enumeration lists the devices that are currently available in AutoSim and gives a short description of their function.

Actuators:

- **DriveActuator:** Controls the wheels of a robot
- **WheelDevice:** Represents the wheel with the attached suspension and brake

- **Propeller:** Simulates a propeller and a corresponding DC motor

Sensors:

- **Camera:** A virtual camera for the robot (available only on the client)
- **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 (latitude and longitude), velocity, time and a checksum
- **CompassSensor:** Measures the angle between the virtual north axis and the sensor axis
- **InclinometerSensor:** Measures the difference between the current and an initial orientation in relation to a given axis
- **TimeDevice:** Returns the current simulation time
- **LightDevice:** Simple light
- **HeadLightDevice:** Light with a cone

5.3 Noise Models

For added realism every *SimDevice* can have an arbitrary number of noise sources. These noise sources are held in a vector inside the *NoiseGenerator* that is part of any *SimDevice*. When a sensor signal is generated or an actuator is being set the corresponding function applies the noise from the *NoiseGenerator* to the data. This noise is generated by adding all values from the *Noise* objects attached to the *NoiseGenerator*.

Figure 5.3 shows an example of generated gaussian white noise and listing 5.4 lists the algorithm based on the central limit theorem [6] used to generate it. Every noise type has an offset and a range represented by two double values to adjust its impact on the signal it's applied to.

5.4 Construction Process

It might be surprising but roughly 30% of the code is only used during the loading process of the simulation. This involves loading and parsing of the configuration and definition files and other resources like height maps, textures, street maps and 3D models, the processing of the loaded information and finally the creation of all objects forming the

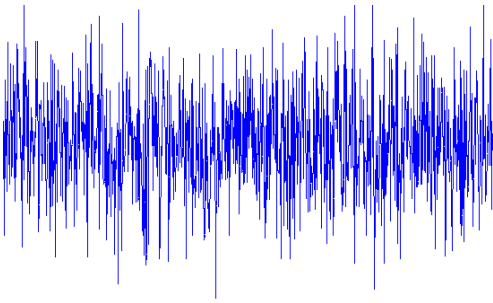


Figure 5.3: Generated Gaussian White Noise

```

1  virtual double getNoise() {
2      double N = 20;
3      double x = 0;
4
5      for (int i=0;i<N;i++)
6          x += udrand();
7
8      x -= N/2;
9      x *= sqrt(12/N);
10     x /= N;
11     return x + m_offset;
12 }

```

Figure 5.4: Implementation of the Noise Algorithm

actual simulation tree that is used during runtime. Even though this section describes only the loading process of the Robot it is paradigmatic for all parts of the simulation on both server and client.

In a first step the description file is being parsed into a custom object model where every "object" ¹⁾ is represented by a *BuildData* object containing all data from the the XML "Object" such as its name and various configuration values. The constructor of the *BuildData* class takes a handle to a XML "object" as its argument and stores all the information (the "variables") from the XML "object" in its internal maps which map a string representing the "variable's" name to either another string or a double representing the "variable's" value. "Variables" may also contain a list of strings or doubles.

The big advantage of parsing the data from a XML file into a number of generic data containers is that these containers can be used as arguments for all build functions (see 3.2.1), providing an unified interface to access their data members no matter what object they describe.

Secondly it makes the actual construction process independent from the underlying configuration data. E. g. all information needed could be stored in plain ASCII files or a data base and be parsed providing additional constructors.

Listing 5.1 shows the data members, parsing and data access functions of the *BuildData* class.

Listing 5.1: The BuildData Class

```

1  class BuildData
2  {
3  public:
4      MAP <STRING, STRING> m_StringMap;
5      MAP <STRING, double> m_DoubleMap;

```

¹⁾ last XML tag in the hierarchy containing child elements

```

6  MAP <STRING, MAP<STRING, double> *> m_MultDoubleMap;
7  MAP <STRING, MAP<STRING, STRING> *> m_MultStringMap;
8
9  BuildData(TiXmlHandle *h);
10 virtual void toXML();
11 virtual STRING& getType();
12 virtual STRING& getName();
13 virtual double getDouble(STRING tagName);
14 virtual double getDouble(STRING tagName, STRING attribName);
15 virtual STRING& getString(STRING tagName);
16 virtual STRING& getString(STRING tagName, STRING attribName);
17 bool containsDouble(STRING tagName);
18 bool containsDoubleArray(STRING tagName);
19
20 protected:
21  STRING m_type;
22  STRING m_name;
23
24  virtual void fromXML(TiXmlHandle *h);
25  int getAttributeCount(TiXmlElement *e);
26 };

```

After all configuration data has been parsed by the *RobotLoader* representing the *Director* 3.2.1 into vectors of *BuildData* objects these objects are then passed on to the specific "build" functions of its *Builders* - in this case the *RobotBuilders*. This construction principle ensures that objects are built in the exact same order on both the server and clients since the configuration data is shared among them and all *Builders* implement the same interface that is used by the *Director*. Therefore the resulting tree structure must be the same on all platforms too.

These build functions are the actual workhorses in the loading process: They use the data from their *BuildData* argument to create the appropriate bodies, links and devices inside the physics, assemble the simulation object, attach the defined noise sources to it and finally register it with the simulation tree.

This process is illustrated by a short example based on the *Robot* definition in listing 4.2. The *RobotLoader* (Director) calls the *buildParts* method of the *ServerRobotBuilder* (the only Builder) passing to it a vector of *BuildData* as an argument which, in this example, only contains the description of a *PSDSensor* device.

For this PSD sensor firstly a new PSD sensor is created within the physics engine which is then initialized with the values from the *BuildData* object. In a second step the *PSDSensor* device is created by calling its constructor and supplying a pointer to the actual sensor in the physics as an argument. Finally the the newly created *PSDSensor* is registered as a child of the *Robot's* part (here: the "chassis") it belongs to and consequently becomes a member of the Simulation Tree.

The procedure described above applies generally to all parts of the simulated world's loading process such as the terrain and street generation which is covered in [3].

5.5 Programming Interface

Since the most "sensor packed" robot is useless without control it was essential to provide the user with a unified interface that is easy to use and understand, yet powerful enough to allow access to all different kinds of devices without any limitations concerning the data format they use.

This goal is achieved through the User Program API mainly consisting of two functions as shown in listing 5.2. A full listing including the definitions of all types is given in A.1.

Listing 5.2: setData and getData functions

```

1 SimDeviceError setData( SimDeviceName device , DeviceData *data , int dataSize );
2 SimDeviceError getData( SimDeviceName device , DeviceData *data , int dataSize );

```

These functions are exported as external C functions by both server and client and form the bridge between them and the user programs which import these functions to control the devices. Either function takes a fully qualified device name, a pointer to the data that is going to be set/read and the data's byte size as arguments.

This API also hides the differences in its implementation from the user program: On the server the device is looked-up using the functionality of the *SerializableTree* and then dynamically cast as a *SimDevice*. The pointer to the data and its size are then simply passed on to the device's setData/getData function to process it.

Since there are no actual devices on the client it has to go a different way when handling the API function calls: All sensor data are retrieved from generic data container nodes in the clients tree representing the devices and actuator data which are being sent to the server. These procedures are described in detail in the Networking chapter 7.

In addition to the functions provided by the user program API the client also provides access functions to the devices that require the rendering system such as the virtual cameras:

Listing 5.3: Virtual Camera Access Functions

```

1 int getImageHeight();
2 int getImageWidth();
3 VirtualCameraImage getImage( SimDeviceName camera );
4 void unlockImage();

```

The first two functions return the dimensions of the virtual camera image in pixels and *getImage* is used to render an image from the camera device specified by its argument. It returns a pointer to a height times width matrix of 32bit values representing the pixels red, green, blue and alpha value. After the operations on the image are finished *unlockImage* frees the memory occupied by it in order to render a new one.

6 Physics Simulation

Even though the PAL library used in this simulator and its underlying physics engines introduce an abstraction layer that widely separates the physics from the actual simulation, this chapter gives a short overview of the methods and models used to calculate the interactions between the simulated bodies followed by three chapters that describe the terrain, the bodies and links as well as the devices available in PAL and their inner life. Also, a closer look is taken at the problems arising with the use of simplified models and numerical calculation.

Physics engines can generally be split up into two major components: a collision detection system, discussed in chapter 6.2, and the actual physics simulation component responsible for solving the forces affecting the simulated objects.

6.1 Rigid Body Dynamics

Since it is still impossible to simulate all details of a physics world due to the enormous amount of equations involved, physics engines that are suitable for real time simulation rely on simplified models for their calculations. These engines take advantage of the fact that most bodies can be approximated by a rigid body which is an "idealization of a solid body of finite size in which deformation is neglected and the distance between any two given points remains constant in time regardless of external forces" [20].

The basic idea is to describe a body's state by its position and orientation and its linear and angular velocities. After the forces on the body and their point of force application have been figured out which is a non-trivial process, covered in section 6.2, the linear acceleration and the total torque at the center of mass can be determined. By numerically integrating the linear and angular acceleration (calculated from the total force and torque at the center of mass) one obtains the velocity and, after another integration step, the new position of the body.

The most important formulas and equations needed to perform these calculations for a 2D body are described in the following paragraphs.

6.1.1 Definitions

The vector to the center of mass \mathbf{R} is the linear combination of the vectors to all the points in the rigid body, \vec{r}_i , weighted by their masses m_i , divided by the total mass of the body.

For a rigid body with a mass density $\rho(r)$ the sum becomes an integral over the body's area A that can be calculated off-line:

$$\mathbf{R} = \frac{\sum_{i=1}^N m_i \vec{r}_i}{\sum_{i=1}^N m_i} \quad (6.1) \qquad \mathbf{R} = \frac{\iint_A \rho(\vec{r}) r dA}{\iint_A \rho(\vec{r}) dA} \quad (6.2)$$

The moment of inertia of a point mass rotating about a known axis is defined by

$$\mathbf{I} = mr^2 \quad (6.3)$$

where m is the mass and r is the (perpendicular) distance of the point mass to the axis of rotation.

The moment of inertia is additive. Thus, for a rigid body consisting of N point masses m_i with distances r_i to the rotation axis, the total moment of inertia equals the sum of the point-mass moments of inertia 6.4.

For a solid body described by a continuous mass density function $\rho(r)$, the moment of inertia \mathbf{I} about a known axis can again be calculated by integrating the square of the distance r from a point in the body to the rotation axis weighted by ρ :

$$\mathbf{I} = \sum_{i=1}^N m_i r_i^2 \quad (6.4) \qquad \mathbf{I} = \iint_A r^2 \rho(\vec{r}) dA \quad (6.5)$$

where A is the area occupied by the object and \vec{r} are the coordinates of a point inside the body.

6.1.2 Simulation Process

All simulated bodies need to be initialized with a position, orientation and linear / angular velocity before the first simulation step can be performed since these values provide the constants of integration for the two integration steps.

After all forces \vec{f}_i on the body have been figured out, the total force and the total torque at the center of mass

$$\mathbf{F} = \sum_{i=1}^N \vec{f}_i \quad (6.6)$$

$$\tau = \sum_{i=1}^N \vec{r} \times \vec{f}_i \quad (6.7)$$

can be calculated.

Given the total force and total torque the linear acceleration a can be determined by dividing \mathbf{F} by the total mass and the angular acceleration α by dividing τ by the moment of inertia \mathbf{I} respectively:

$$a = \frac{\mathbf{F}}{M} \quad (6.8) \quad \alpha = \frac{\tau}{\mathbf{I}} \quad (6.9)$$

Now it's a simple step to the new position and orientation. With the definition of acceleration and angular acceleration as the second derivative of the position and orientation respectively

$$a = \dot{v} = \ddot{r} \quad (6.10) \quad \alpha = \dot{\omega} = \ddot{\Omega} \quad (6.11)$$

the velocity and angular velocity is calculated by numerically integrating the acceleration and angular acceleration. Repeating this integration delivers the position and orientation. A very simple numerical integration method is the Euler method which multiplies the variable to be integrated with the integration step size and adds it to the previous value. This example shows the numerical integration of the acceleration where n is the time step and h the integration step size:

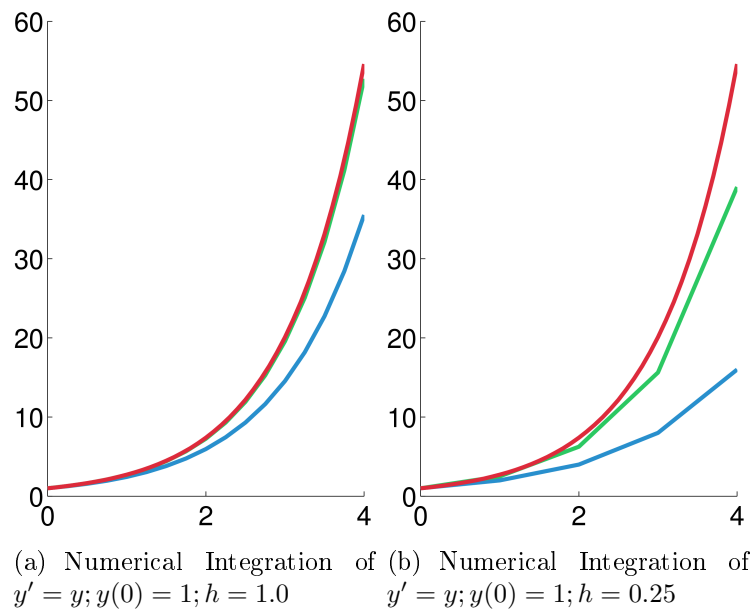
$$v_{n+1} = v_n + ha_n \quad (6.12)$$

Together with the algorithms calculating the collision response (see next section) the choice of the integration method is essential for the stability of the simulation. Figure 6.1.2 shows different results for the numerical integration of the ordinary differential equation $y' = y$ using the exact solution $y = e^t$ (red), Euler Method (blue) and the more sophisticated Midpoint Method (green):

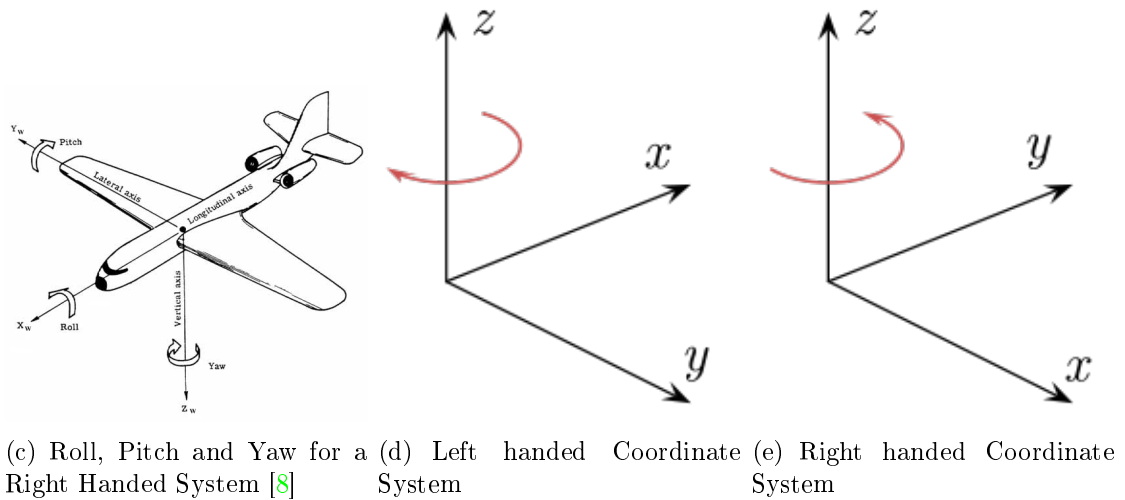
$$y_{n+1} = y_n + hf(t_n + \frac{h}{2}, y_n + \frac{h}{2}f(t_n, y_n)) \quad (6.13)$$

Even with a reduced step size it is obvious that the numerical integration introduces errors into the equations that cause instabilities and result in unrealistic behavior of the simulation.

The principles described above by the example of a two dimensional body basically carry across to 3D where R becomes a volume integral and I a tensor. The position of a



rigid body is then determined by 6 parameters: the position of its center of mass (coordinates in 3D space) and by its orientation (roll, pitch and yaw) as shown in figure 6.1(c).



6.2 Collision Detection and Response

As mentioned above the detection of collisions between objects poses one of the biggest challenges in a physics simulation since the methods used to resolve the collision are essential for the stability, consumed calculation power and realism of a simulation.

There are generally two ways to respond to a collision: The first is called the *a posteriori* method. It allows interpenetration of bodies and uses the gathered data such as its depth and the velocity and collision normal to calculate impulses that instantaneously change the velocity of the bodies. This method is comparatively easy to implement but causes stability problems since the actual collision is missed and "fixed" later on.

The second method predicts the trajectory of the bodies and the instants of collision are calculated with high precision which keeps the physical bodies from interpenetrating - it is called *a priori* method. This approach results in an increased fidelity and stability but also has to take a myriad of physical variables into account causing higher complexity.

6.3 Terrain and Streets

In the physics the terrain and the streets are triangle meshes represented by *palTerrain-Meshes*. The terrain mesh is generated once for the whole simulated world during the loading process using the height map that can be defined in the configuration file. Slightly elevated from the terrain are the meshes for the streets.

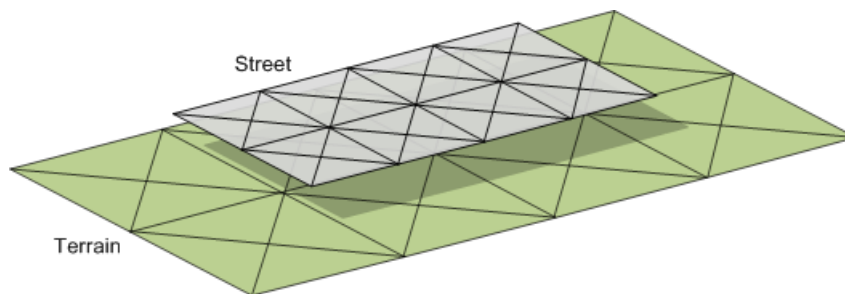


Figure 6.1: Terrain and Street Triangle Meshes in the Physics Engine

The data for the construction of the streets is provided by the OSM files. For every street segments defined in the OSM file a spline is generated. Along this spline the mesh for the street is created through triangulation taking the defined type, width and other parameters into account. The terrain and street generation process is covered in detail in [3]. Figure 6.1 shows a schematic terrain mesh with a street mesh floating above it.

This concept not only allows for a reuse of the triangulation algorithms written for the graphics system but also the definition of friction parameters different to the ones for the terrain.

6.4 Bodies and Links

The following section gives a brief overview of the various body shapes available in the PAL. These shapes form the "least common denominator" of the supported shapes of all supported physics engines that can be used with PAL. By default PAL provides a list of standard shapes:

- **palBox**: Represents a simple box (e.g.: cube, rectangular prism) at a given position, with a given width, height, depth and mass [6.2\(b\)](#)
- **palCapsule**: Represents a simple capped cylinder at a given position, with a given radius, length and mass
- **palSphere**: Represents a simple sphere at a given position, with a given radius and mass [6.2\(a\)](#)

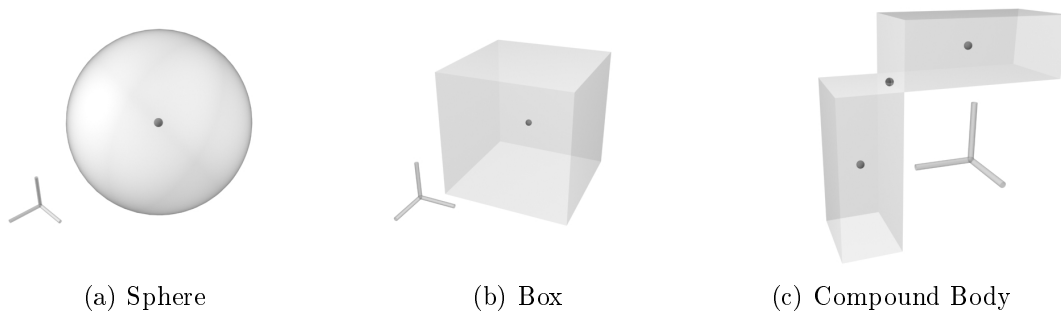


Figure 6.2: Physics Bodies [\[2\]](#)

In addition to the shapes mentioned above PAL provides a compound body class representing a body composed of multiple geometries. It combines a number of elementary geometry types to create a more complex body. For very complex objects it is advisable to use meshes instead of compound bodies which increase accuracy when performing collision detection. The downside to this method is that it also increases the demand for CPU power since collision detection has to be performed for every single triangle of the mesh.

6.5 Devices

On the physics level the devices provided by the PAL are used to extract data from the underlying physics engine. The PSD sensor, for example, performs a ray cast to determine the distance to the closest object from defined point in a given direction. Other devices extract data from or apply forces and impulses to the bodies they're attached to such as a

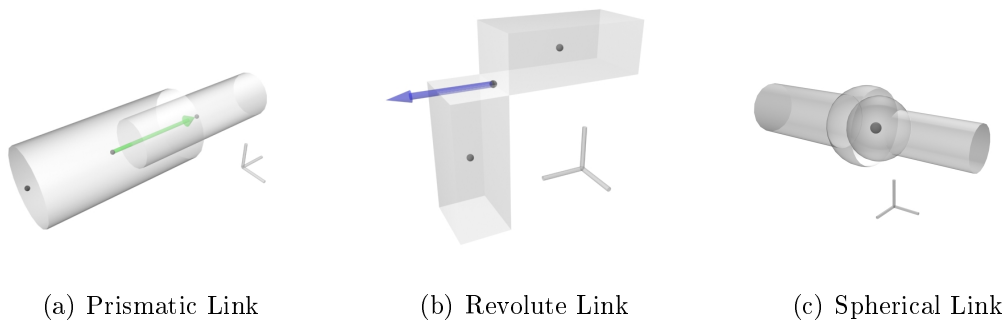


Figure 6.3: Links between Physics Bodies [2]

velocimeter that returns the current speed of a body in a certain direction or a propeller applying a force at its attachment point.

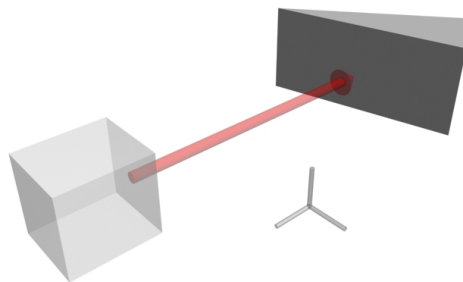


Figure 6.4: PSD Sensor [2]

The PAL's "vehicle" belongs to the third group of devices providing the simulation with physical models for complex structures such as vehicles whose complete model would be too CPU power consuming and/or cause stability problems during the integration process e.g. when integrating the differential equations of the suspension's springs. These devices employ simplified algorithms to calculate the forces and impulses on their belonging body as well as the (virtual) position of the parts they simulate in order to display them in a graphics system (e.g. the wheels of a car).

A common way to do this for a vehicle is to perform a ray cast at the position of every wheel pointing downwards and to use this distance data in combination with the orientation, linear and angular velocity of the body to calculate the resulting impact on it.

Even though this simplified model delivers acceptable results for "flat" terrains it's causing problems on "bumpy" ground since the actual size of the wheel is not taken into account which gives the simulated car a tendency to get stuck easily in small gaps or abrupt changes in the terrain's height e.g. at curbs.

7 Networking

As mentioned earlier one of the main goals for the driving simulator was to make it fully networked which requires techniques that are fast and flexible yet bandwidth and CPU power efficient enough to enable the server to concentrate on calculating the physics and therefore to allow a higher number of connected clients. This chapter gives an overview of the internal workings of the data distribution and the implementation of the actuator access for the client side user programs.

7.1 Update Distribution

After the server has loaded the physics world and the whole simulation tree has been constructed the network subsystem is loaded and waits for incoming connections on a predefined port. Given the port and IP address of an AutoSim server the clients can then connect to it after finishing their loading process. For both systems the used RakNet network library maintains a list of all connected systems and also provides queues for incoming and outgoing network packets. As soon as at least one client is connected the server can start to deliver updates.

For every time step that is run on the physics a new network packet is generated by calling the *toBitStream* function of the simulation tree's root node which serializes the current simulation status. This bitstream is then broadcast to all connected clients by using RakNet functions. Figure 7.1 shows a network of an AutoSim server, two local and one remote AutoSim client all connected through an IP network:

The benefit of using broadcasts is that a single packet can be used to update all clients which not only saves bandwidth but also minimizes the generation time since it can be reused thus the serialization as the most expensive part has to be called only once every time step.

On the clients this packet is identified by its first byte carrying an 8 bit identifier as an update packet and its actual data is passed on to the *fromBitStream* of the corresponding node in the client's simulation tree. However should a packet arrive after a more recent one has been received or the client is too busy to process all received packets only the most recent one is kept and all the rest are dropped.

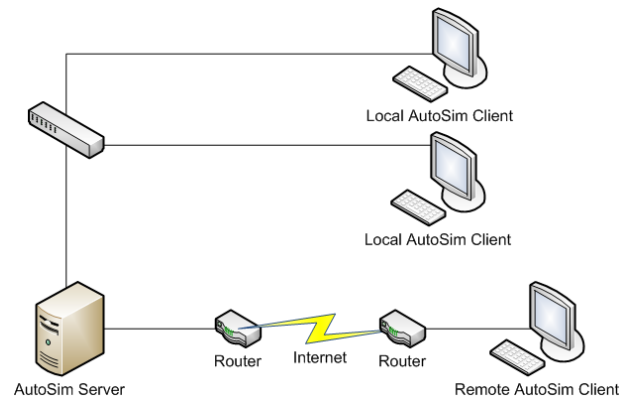


Figure 7.1: AutoSim Client Server Network

7.2 Remote Sensor Access

Since type information is not required for the sensor data all devices are represented by generic data containers on the client. They have the same *setData* and *getData* functions (see 5.2) as their counterparts on the server. The difference is however that the sensor data is generated by concatenating all data members (which are updated with every network packet received from the server). This block matches exactly the data format of the original device and therefore it doesn't make a difference for the User Program whether it runs on the client or server side when accessing sensors.

7.3 Remote Actuator Access

In order to enable the client side user programs to influence simulation the AutoSim client needs to provide a transparent way to send data back to the server that runs the physics engine which is achieved by a different implementation of the user program API's *setData* function: Whenever a user program tries to write to a device, a packet containing a packet type identifier, the fully qualified device name and the DeviceData (see A.1) is generated and sent to the server. For every received packet of that type the server looks up the corresponding actuator and calls its *setData* supplying it with the data from the network packet.

8 Conclusion and Future Work

AutoSim, a fully networked, real-time and open source robot simulation framework has been released. It provides the user with a virtual world generated out of real world height data and geographic information, such as streets and buildings, that can be obtained from free online sources such as Open Street Maps. Every robot in the simulation can be fitted out with a wide range of sensors and actuators and is controlled by a separate program, the user program, that is compiled as a DLL and loaded at runtime by either the server or the client. Every detail of both the simulated world and the the robots can be configured through industry standard xml files without requiring the use to change and/or recompile any code.

At the moment the AutoSim framework is only available for the Microsoft Windows platform but since all used libraries are available for other platforms too, such as Linux and MacOS X, a porting to them is planned. The Sony Playstation 3 platform is also a very interesting option since the Bullet physics engine (also developed by Sony) already runs on it and makes use of its extensive multiprocessing features provided by the Cell processor.

For the future it would be desirable to have a bigger number of demo programs as well as a full manual-like documentation that simplifies the access to the framework especially for unexperienced programmers. Another interesting feature would be the support of LASER and LIDAR scanners to support the development of autonomous robots as proposed in [12]. Also an improved version of the graphical user interface giving more feedback about the simulation is already under development.

A Code Listings

A.1 The User Program API

Listing A.1: The User Program API

```

1 namespace UserProgramAPI
2 {
3     /// Typedefs for types used by the User Program API
4     typedef STRING RobotName; ///< Name of the Robot
5     typedef STRING SimDeviceName; ///< Name of the device
6     typedef unsigned int SimDeviceError; ///< Device error value
7     typedef unsigned int SimDeviceStatus; ///< Status code for a device
8     typedef void DeviceData;
9
10    /// Enumeration for error codes
11    enum
12    {
13        NO_DEVICE_ERROR = 0,
14        UNKNOWN_ROBOT = 1,
15        UNKNOWN_DEVICE = 2,
16        INVALID_DEVICE = 3,
17        INVALID_OPERATION = 4,
18        DATA_SIZE_MISMATCH = 5
19    };
20
21    /// Enumeration for device state
22    enum
23    {
24        INVALID = 0,
25        READY = 1,
26        BLOCKED = 2,
27        ACTIVE = 4,
28        PAUSED = 8
29    };
30
31    /**
32    Sets the data for the specified device on the specified robot. */
33    USER_PROGRAM_API_DECLSPEC SimDeviceError setData( SimDeviceName device,
34        DeviceData *data, int dataSize );
35
36    /** Gets the data from the specified device on the specified robot. */
37    USER_PROGRAM_API_DECLSPEC SimDeviceError getData( SimDeviceName device,
38        DeviceData *data, int dataSize );
39 };

```

Listing A.2: The Client User Program API

```

1 namespace ClientUserProgramAPI
2 {
3     typedef unsigned int * VirtualCameraImage;
4
5     /** The height of the image generated by the virtual camera */
6     VIRTUAL_CAMERA_API_DECLSPEC int getImageHeight();
7

```

```
8  /** The width of the image generated by the virtual camera */
9  VIRTUAL_CAMERA_API_DECLSPEC int getImageWidth();
10
11 /** Retrieves the VirtualCameraImage from the specified camera device */
12 VIRTUAL_CAMERA_API_DECLSPEC VirtualCameraImage getImage( UserProgramAPI::
    SimDeviceName camera );
13
14 /** Releases the memory occupied by the VirtualCameraImage for reuse. */
15 VIRTUAL_CAMERA_API_DECLSPEC void unlockImage();
16 };
```

B Tutorials

B.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 plus extra information added in red.

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 don't 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 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.

B.2 Working with the AutoSimClient



Figure B.1: 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 don't 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.

B.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.

B.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.

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.

B.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 B.1: Parts section of a robot configuration file

```

1 namespace UserProgramAPI
2 {
3     SimDeviceError setData( SimDeviceName device , DeviceData *data , int dataSize
4         );
5     SimDeviceError getData( SimDeviceName device , DeviceData *data , int dataSize
6         );
7 };

```

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

B.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 B.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 being 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.

B.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 B.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.

B.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 doesn't have any functionality. This tutorial gives a quick start guide for using the OsmManipulator.

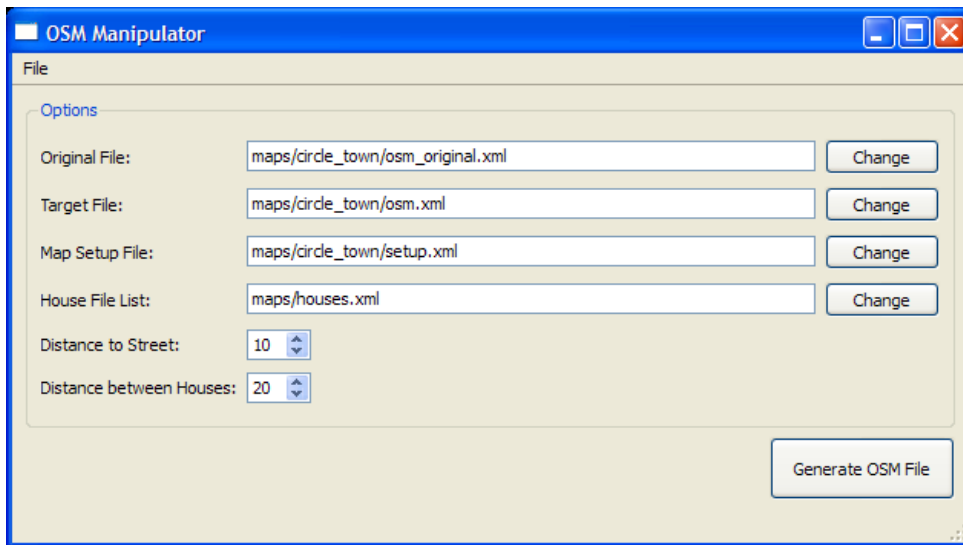


Figure B.2: OsmManipulator

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* [C.4](#).
2. Go to the *Target File* input field and 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 [C.5](#).
4. Finally select a *House File List*. *House File List* description [C.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.

C The Configuration Files

C.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'll take the *robot* description file as an example:

Listing C.1: Exemplary onfiguration 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'll find a detailed description of all tags and attributes inside the corresponding XML file.

C.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 synchronize 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.

C.3 The Robot File

location: /robots/

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 C.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 C.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: Controls 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
- InclinatorSensor: 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.

C.4 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 [10].

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 continuous 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 [?] of the OpenStreetMap webpage. However, custom tags can be defined as well. Table C.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 C.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* C.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* C.6 and an example of *landuse* tags is presented by table C.2:

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

Table C.2: *landuse*

Additionally to using predefined OSM tags the user can also define his own tags. The OsmManipulator for example constructs house nodes that don't 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 C.3:

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 C.3: house node

C.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 [C.4](#).

Listing C.4: OsmSetup

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <Setup name="circle_town">
3   <TerrainSetup>
4     <OsmSetup name="circle_town">
5       <!-- Lat and lon coordinates of the simulation world center-->
6       <NodeOffset lat ="-31.9728" lon="115.827" />
7     </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 C.5: RoadDimensions

```

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

```

Within the OSM file description [C.4](#) 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 C.6: HighwayDimensions

```

1 <HighwayDimensions name="circle_town">
2   <motorway width ="16.0" leftPavementWidth ="1.5" rightPavementWidth ="1.5"/>
3   <motorway_link width ="4.0" leftPavementWidth ="0.0" rightPavementWidth ="0.0
4     "/>

```

```

4 <trunk width ="12.0" leftPavementWidth ="1.5" rightPavementWidth ="1.5"/>
5 <trunk_link width ="4.0" leftPavementWidth ="0.0" rightPavementWidth ="0.0"/>
6 <primary width ="16.0" leftPavementWidth ="1.5" rightPavementWidth ="1.5"/>
7 <primary_link width ="4.0" leftPavementWidth ="0.0" rightPavementWidth ="0.0"
  />
8 <secondary width ="8.0" leftPavementWidth ="1.5" rightPavementWidth ="1.5"/>
9 <tertiary width ="6.0" leftPavementWidth ="1.5" rightPavementWidth ="1.5"/>
10 <unclassified width ="6.0" leftPavementWidth ="0.0" rightPavementWidth ="0.0"
  />
11 <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 shouldn't be a visible changeover caused by a seamed texture the intersections are triangulated by a triangle fan [C.1](#) in a way to provide a seamless transition.

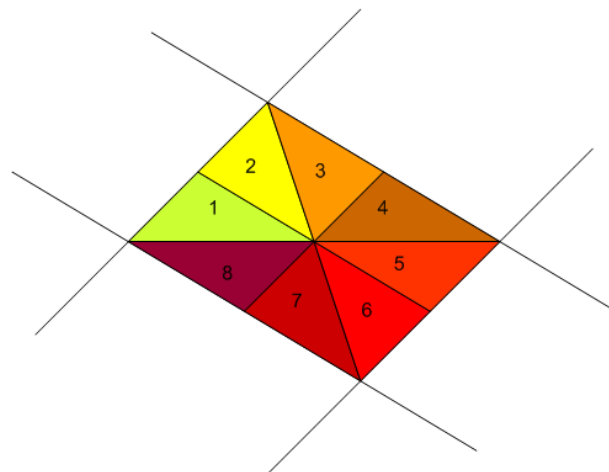


Figure C.1: Triangle Fan

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 C.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
13     be changed to a value bigger than one
14     if no texture for the complete terrain is used. The used texture should be
15     seamless then. -->
16 <TextureRepeat value ="300.0"/>
17 <!-- Folder path and file type of the skybox textures.
18     Inside the folder have to be six texture files of the declared file type:
19     1. left.*
20     2. front.*
21     3. right.*
22     4. back.*
23     5. top.*
24     6. bottom.* -->
25 <SkyBox path ="media/terrain/skyboxes/grass_and_hills" type="jpg"/>
26 </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 C.8: TerrainDimensions

```

1 <TerrainDimensions name="circle_town">
2 <!-- Size of the loaded terrain in meters. The terrain has to be at least big
3     enough to load all the street data of the OSM file and to contain all the
4     tiles the graphical representation of the terrain is made of (can be set
5     in the following values). A Terrain bigger than 3000m*3000m may effect
6     long loading times and slow physics. -->
7 <Size width ="1000.0" height="1000.0"/>
8 <!-- Size of one tile of the graphical terrain representation in meters.
9     tileSize has to be a multiple of HeightDataPerArea! -->
10 <TileSize value ="64.0"/>
11 <!-- Detail value of the graphic terrain's center tile. (maximum = 7) -->
12 <MaximumLOD value="6.0"/>
13 <!-- Number of levels of detail added to the detail level of the center tile.
14     Each additional level of detail will be one step lower. (e.g. the
15     simulator will add tiles of detail values 6,5 and 4 if MaximumLOD = 7 and
16     LevelsOfDetail = 3)-->
17 <LevelsOfDetail value="3.0"/>
18 <!-- Layers of one level of detail.-->
19 <LayersOfEachLOD value="1.0"/>
20 <!-- Number of meters to the next height data. (in meters per height data
21     value). TileSize has to be a multiple of HeightDataPerArea! -->
22 <HeightDataPerArea value="8.0" />
23 <!-- Specifying the size of the steps the graphic terrain follows the camera.
24     The value is related to the detail value of a tile. (maximum = 7). (e.g.
25     if the value is set to 7 the graphics terrain will always move in steps of
26     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 doesn't 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>

```

C.6 The House File List

The OsmManipulator automatically creates houses or trees along the streets and saves them into the OSM file C.4. The created nodes contain information like the path and type of the 3D model C.3. For adding these kinds of information they have to be taught 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 C.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 C.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 C.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>
```

C.7 General Information on Model Files

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] http://jcwinnie.biz/wordpress/imageSnag/Stanley_Image13.jpg. vi, 2
- [2] BOEING, ADRIAN: *Physics Abstraction Layer - Doxygen Documentation*. Technical Report, UWA, <http://www.adrianboeing.com/pal/current/pal/documentation/html/>, 2008. vi, 39, 40
- [3] BRAND, JOHANNES: *Graphics for a 3D Driving Simulator*. Technical Report, RCS, 2008. 19, 32, 38
- [4] BRUCE ECKEL, CHUCK ALLISON: *Thinking in C++, Vol. 2: Practical Programming, Second Edition*. 2003. 5, 12
- [5] CAR, RACER and RACING SIMULATOR: <http://www.racer.nl/>. vi, 9
- [6] DONADIO, MATT: <http://www.dspguru.com/howto/tech/wgn.htm>. 29
- [7] ERICH GAMMA, RICHARD HELM, RALPH JOHNSON JOHN VLISSIDE: *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. 5
- [8] MAHONEY, M. J.: *Pointing an Instrument on an Airborne Platform*. Technical Report, NASA, <http://mtp.jpl.nasa.gov/notes/pointing/pointing.html>, 2008. 37
- [9] MARSHALL CLINE, GREG LOMOW, MIKE GIROU: *C++ FAQs*. 1994. 24
- [10] OPENSTREETMAP: <http://www.openstreetmap.org/>. 6, 59
- [11] (RARS), ROBOT AUTO RACING SIMULATOR: <http://rars.sourceforge.net/>. vi, 7, 8
- [12] SEUNG-HUN KIM, CHI-WON ROH, SUNG-CHUL KANG MIN-YONG PARK: *Outdoor Navigation of a Mobile Robot Using Differential GPS and Curb Detection*. Intelligent Robotics Res. Center, Korea Inst. of Sci. Technol., Seoul, 2007. 43
- [13] SYSTEM, SUBSIM AN AUTONOMOUS SUBMARINE SIMULATION: <http://robotics.ee.uwa.edu.au/aw/subsim.html>. vi, 10
- [14] TARTAN RACING, CARNEGIE MELLON UNIVERSITY: <http://www.tartanracing.org/>. vi, 3
- [15] THOMASON, LEE: <http://www.grinninglizard.com/tinyxml/>. 11
- [16] TORCS, THE OPEN RACING CAR SIMULATOR: <http://torcs.sourceforge.net/>. vi, 8

- [17] UNIBW, THE COGNITIVE AUTONOMOUS VEHICLES OF: <http://www.unibw.de/lrt13/tas/medien/elrob2007-universitaetderbundeswehrmuenchen.pdf?searchterm=vamors>. vi, 1
- [18] WIKIPEDIA: http://en.wikipedia.org/wiki/Google_maps. 6
- [19] WIKIPEDIA: <http://en.wikipedia.org/wiki/LIDAR>. 2
- [20] WIKIPEDIA: http://en.wikipedia.org/wiki/Rigid_body. 34
- [21] WIKIPEDIA: [http://en.wikipedia.org/wiki/Tree_\(data_structure\)](http://en.wikipedia.org/wiki/Tree_(data_structure)). 24