Evaluation of a Vision-based Driver Assistance System in Simulation

Centre for Intelligent Information Processing Systems (CIIPS)
University of Western Australia

Pål Simen Ruud
August 2007 – June 2008

Master of Engineering in Information and Communication Technology Thesis

Supervisor: Associate Professor Thomas Bräunl

Pål Simen Ruud
65B Burniston Street
SCARBOROUGH WA 6019
3rd of June 2008

The Dean
Faculty of Engineering Computing and Mathematics
The University of Western Australia
25 Stirling Highway
CRAWLEY WA 6009

Dear Sir

I submit to you this dissertation entitled ”Evaluation of a Vision-based Driver Assistance System in Simulation” in partial fulfilment of the requirement of the award of Master of Engineering in Information and Communication Technology.

Yours faithfully

....

Pål Simen Ruud

Abstract

Testing driver assistance systems on real vehicles in real environments are both costly and dangerous. By developing an automotive simulation system, new driver assistance systems for vehicles can be tested and improved in a safe and reliable way. My contributions in this project have been to assist in the development of the automotive simulation system AutoSim and further integrate the image processing framework ImprovCV in order to be used for autonomous driving of vehicles in AutoSim. For the development of AutoSim my main contributions have been in the area of content creation, 3D modelling and physics simulations. Finally, lane detection based on image processing in ImprovCV was used in order to demonstrate autonomous driving and a driver assistance system in AutoSim. Two different scenarios were created for the experiments and different controllers were used in order to find the most robust one. My implementation of the Fuzzy logic controller proved to be the most suitable one when looking at the overall performance for the two scenarios. Also as different physics engines are known to behave different, the experiments were conducted using two different ones, Bullet and Newton, in order to further verify the robustness of the autonomous driving. Differences in the physics engines were discovered and only the Fuzzy logic controller implementation completed all scenarios with both physics engines.

Acknowledgements

I would like to thank the University of Western Australia for allowing me the opportunity to do this interesting project in partial fulfilment of my Master of Engineering in Information and Communication Technology.

Furthermore, the project would not have been possible without the supervision from Associate Professor Thomas Bräunl and his guidance throughout the duration of the project. The project has really opened my eyes for many new technologies that were unknown to me before the start of the project and I am certain the new knowledge obtained will help me in a future work setting.

I would also like to thank Adrian Boeing for all the good ideas and help that he has given me. His experience, knowledge and desire to help have been greatly appreciated.

Finally, my thanks go to all the other people that have given me help and directions in order for me to solve problems and finally finish the project.

Table of Contents

1 Introduction....................11
1.1 Background....................11
1.2 Project Objectives....................12
1.3 Thesis Structure....................13
2 Literature Survey....................15
2.1 Driving Simulators....................15
2.2 Autonomous Vehicles....................18
2.2.1 DARPA Grand Challenge....................18
2.2.2 EUREKA Prometheus Project....................19
2.2.3 Space Robots....................20
2.3 Available Subsystems....................21
2.3.1 SubSim and EyeSim....................21
2.3.2 Delta3D....................22
2.3.3 The Open Racing Car Simulator (TORCS)....................23
2.4 Driver Assistance Systems....................24
3 AutoSim - The Automotive Simulation System....................27
3.1 Overview of AutoSim....................27
3.2 Irrlicht....................29
3.3 OpenStreetMap....................30
3.4 TinyXML....................31
3.5 Blender....................31
4 3D Modelling and Content....................33
4.1 Differences in Coordinate Systems....................33
4.2 Light Reflection....................34
4.3 The Preferred Model Formats Used in the Simulator....................35
4.3.1 OBJ and 3DS....................35
4.3.2 The AS3D (AutoSim 3D) format....................36
4.4 Traffic Lights....................36
4.5 Other Objects....................38
5 Physics Simulations....................39
5.1 Physics Engines....................39
5.1.1 Bullet Physics Library....................40
5.1.2 Newton Game Dynamics....................40
5.2 PAL....................40
5.3 Vehicle Physics....................41
5.3.1 Straight Line Physics....................41
5.3.2 Wheels....................44
5.3.3 Steering....................45
5.3.4 Springs....................47
5.3.5 Implementation of Vehicle Physics....................47
5.4 COLLADA....................48
5.5 Scythe Physics Editor....................49
5.5.1 Scythe Loader for PAL....................50
6 Image-based Driver Assistance Systems....................51
6.1 ImprovCV....................51

6.2 Lane Detection..53
6.2.1 The Lane Detection Algorithm..53
6.2.2 Finding the Outer Lane Markings..54
7 Control of a Simulated Vehicle..57
7.1 Control Methods...57
7.1.1 On-Off Controller...58
7.1.2 PID Controller 58
7.1.3 Fuzzy Logic...61
7.2 Communication Between the AutoSim Client and ImprovCV Using Shared Memory..66
8 Experiments and Results...69
8.1 The Test Scenarios...69
8.2 The Controllers for Steering the Vehicle..70
8.3 Limitations 71
8.4 Scenario 1 Results..72
8.5 Scenario 2 Results..74
8.6 Replacing the Physics Engine 76
8.7 Robustness of the System..78
9 Conclusion and Future Work...81
Abbreviations...85
References...87
A Appendix...93
A.1 Autonomous Driving..93
A.2 Preparing a Car Model for the Simulator 94
A.3 Creating a World..97

1 Introduction

1.1 Background

Testing new technologies like safety warning and driver assistance systems on real vehicles in real environments can both be expensive and dangerous. By developing an automotive simulation system new systems like those mentioned above can be tested and improved in a safe environment. As the traffic around the globe is getting denser and denser every year, new driver assistance systems are necessary in order to reduce accidents and the costs related to them [1]. It is estimated world-wide that 1.2 million people are killed in road accidents each year and as many as 50 million are injured [2], and the numbers are increasing every year. A simulator and driver assistance systems like the ones described in this thesis can be used to prevent future accidents. Training and education of new drivers [3] are other areas of use. Safety systems such as seat belts and air bags have made cars safer, however the current trend among car manufacturers are to implement systems that help people avoid crashing rather than help to survive them.

During the last two decades a number of systems have been implemented in cars world-wide. Some of the most successful and widely used are the Anti-lock Braking System (ABS) [4] and the Electronic Stability Program (EPS). These systems were first implemented in high-end cars, but have now become standard in basically all cars. The same trend can be seen nowadays in the high-end cars and some of the new systems in these cars will probably become standard in the future. However, many of the new systems developed these days are removing some control away from the driver of the vehicle and it is therefore absolutely crucial that the systems are well tested [5]. Failure in a driver assistance system can be disastrous.

Taking it even one step further is to design autonomous driving systems, vehicles that drive themselves. It might sound like a science fiction movie, but a number of tests and systems are already developed [6]. The DARPA Grand/Urban Challenge [7][8] is one competition where teams compete in developing autonomous cars for cities and outback. Similar projects in Europe are the European Land-Robot Trial (ELROB) and the EUREKA PROMETHEUS Project [9]. Although autonomous vehicles are still in their infancy, the new systems being developed are bringing us step by step closer to the ultimate goal; autonomous driving available to the public.

In order for the simulations to provide the user with correct information it is important that the simulation system has the accuracy needed. Depending on the simulation scenario this can be the accuracy of the physics models used for the vehicle, such as tire friction and behaviour of suspension [4]. Well developed physics engines will provide the foundation for the automotive simulator in order to replicate real behaviour.

1.2 Project Objectives

The project objective is to create an automotive simulation system for various types of vehicles. Developing an automotive simulation system is a major task and hence the development is expected to continue for years to come. My contributions in the development of the automotive simulation system have been in the area of content creation and physics simulations.

Furthermore, the next goal is to test a driver assistance system in the automotive simulation system. The driver assistance system is based on lane detection implemented in the image processing framework ImprovCV. ImprovCV and lane detection algorithm are already developed at the University of Western Australia for image processing specifically designed for vehicles. However modifications are necessary in order to achieve autonomous driving in the automotive simulation system. The robustness of the lane detection and the control of the simulated vehicle will be tested using different scenarios and controllers for the vehicle. Two different physics engines will also be used in order to verify the robustness of the lane detection algorithm and the controllers for the vehicle.

1.3 Thesis Structure

Chapter 2 will give a review of my literature survey which I conducted in the initial phase of the project. Topics covered in the literature survey are available simulation systems and driver assistance systems. An overview of the design and architecture of the developed automotive simulation system, AutoSim, is given in chapter 3 before the 3D modelling and content creation for AutoSim is discussed in chapter 4. In chapter 5 aspects of vehicle physics and general physics simulations which are relevant for the simulation system is described. The image processing in ImprovCV and the lane detection algorithm is described in chapter 6 together with the communication between the two programs ImprovCV and the AutoSim client. Chapter 7 discusses the theory behind the controllers that are implemented in order to demonstrate autonomous driving before the experiments and results are given in chapter 8. Finally, a conclusion is made in chapter 9 together with future work and improvements.

2 Literature Survey

As preparation for the project a literature survey was conducted in order to learn about current automotive simulations systems and their uses. The process of deciding which subsystems the simulator would be based on was also an important part of this phase in the project. A summary of available simulators, technologies and driver assistance systems are described in this chapter.

2.1 Driving Simulators

An automotive simulation system is not a new invention, but many of the available systems developed are proprietary and owned by car manufacturers and research institutes. The driving simulator at the Technische Universität München (TUM), the DaimlerChrysler systems and The National Advanced Driving Simulators (NADS) at the University of Iowa [10] are three examples.

The TUM driving simulator's movement of the vehicle cabin is based on the Stewart platform [11] with a dome surrounding the cabin. The inside of the dome displays the graphics of the simulator [12]. The cabin itself is detachable in order to use different vehicles, this is also the case for the actual simulation system (the software) which can be modified to a specific vehicle. To enhance the feeling of driving, real vehicles can be fitted to the platform. The Stewart platform gives the vehicle six degrees of freedom to simulate real movement. Figure 2.1 shows the simulator set up.

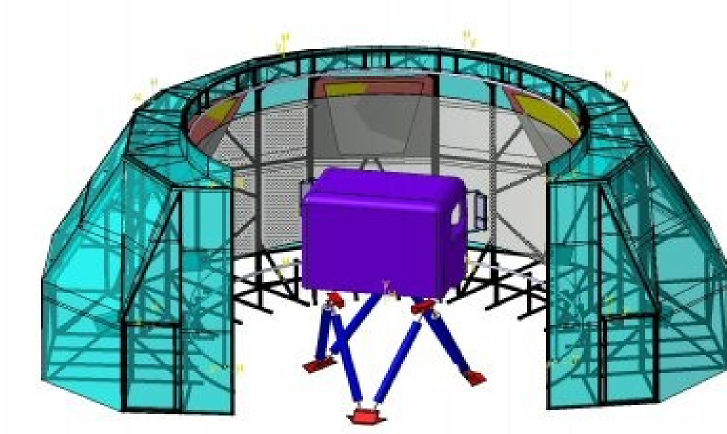


Figure 2.1: The TUM Driving simulator with its surrounding dome and Stewart platform [12]

The Stewart platform has been used in many simulators over the years to reproduce real motion from e.g. a vehicle or an airplane. Compared to the platform that the NADS simulator is based on (described below) using a Stewart platform can be fitted and used in much smaller rooms. However the freedom of the movement is limited.

One of the NADS simulators at University of Iowa [13] uses a different system than the TUM simulator. The NADS 1 system has a 13 degree of freedom system and like the TUM simulator the vehicle cabin is placed inside a dome. The difference however is that the dome can actually move around in a large room making the movement more realistic, depicted in Figure 2.2.

Figure 2.2: The NADS 1 simulator and its 13 degrees of freedom platform [13]

The platform system pictured in Figure 2.2 is said to be able to reproduce motion closer than any other vehicle simulator and makes this simulator one of the world's most advanced. Furthermore, the inside of the dome is fitted with a 360 degrees visual display system making also the visual impression realistic.

DaimlerChrysler's Virtual Reality Centre in Germany consists of much high performance equipment related to development of vehicle systems, especially in the area of visualisation. Two of their visualisation systems are called the “Cave” and the “Powerwall”. These systems are highly advanced screens for displaying both 2D and 3D images and are designed to provide the user with a better reality than other systems. The “Cave” can be seen as an advanced version of the TUM's dome.

As seen these three systems share many similarities. This is also the case for their use; testing and development of driver assistance systems. The DaimlerChrysler's simulators have been used in the development of numerous new systems for vehicles [14]. As in the case for autonomous driving [14] it is absolutely necessary to test the newly developed systems in a simulator before letting an autonomous car out in real-traffic.

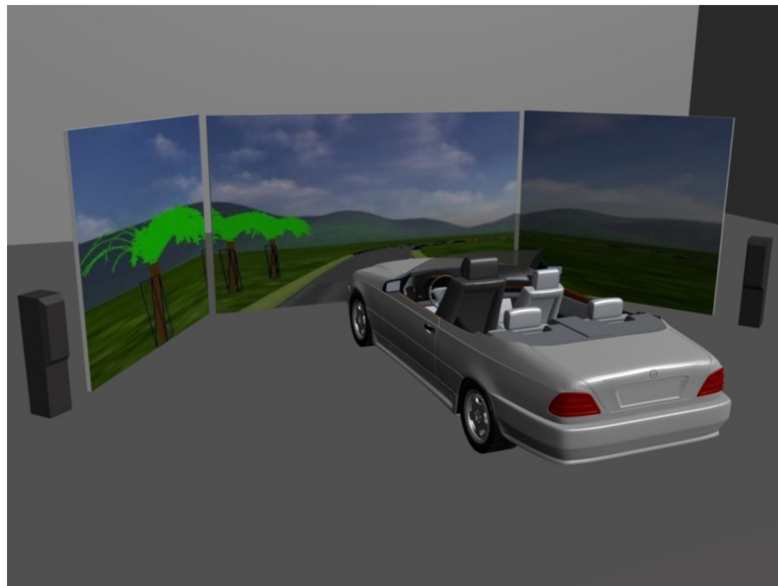


Figure 2.3: An example driving simulator, in this case the vehicle is fitted with a motion chair in the driver seat

2.2 Autonomous Vehicles

2.2.1 DARPA Grand Challenge

Defense Advanced Research Projects Agency (DARPA) Grand Challenge [15] is a competition for autonomous vehicles, also known as driverless vehicles. The goal for the competition is to encourage research environments to create and develop vehicles that are capable of driving on their own and completing a predefined track in the shortest amount of time. As of May 2008 three challenges have been held, with the first one in 2004. The two first one were held in a desert-like environment while the last one, the DARPA Grand Challenge 2007 was held in a closed air force base. The last one is often referred to as "DARPA Urban Challenge" because the scenario was a more similar to a city rather than desert.

In the first Grand Challenge no vehicles manage to complete the course. However, next year, in 2005, five teams completed the course. The vehicle that completed the track in the shortest amount of time and won was the Stanley robot developed at Stanford University [16]. Although, the challenge was carried out in a desert-like environment there were obstacles such as bridges, narrow gates and different types of surfaces making the competition far from trivial. The robots of the 2005 Grand Challenge had to traverse through 132 miles (212.4 km) of desert in less than ten hours [17]. This was a huge achievement by the five robots that managed to get through the track and in comparison to the previous year when the maximum distance driven by a robot was only 7.3 miles (11.7 km).

Figure 2.4: Stanley, the winner of the DARPA Grand Challenge 2005 [16]

As the next step, DARPA organized the Grand Challenge's inheritor, the Urban Challenge. In difference to the desert scenario in the Grand Challenge the scenario was now city-like. This time the autonomous vehicles had to interact with other moving vehicles and obey the traffic rules[18]. By utilising technologies like lasers, GPS and radars the winner was able to navigate through the almost 60 miles (96.6 km) long track interacting with as many as 60 other vehicles. Although the scenario for the Urban Challenge was simplified in comparison to a real-world traffic scenario, the challenge brought autonomous vehicles one step closer to a fully autonomous car.

2.2.2 EUREKA Prometheus Project

The EUREKA Prometheus Project was one of the first serious projects in the area of modern autonomous or driverless cars. It was established already in 1987 and one of the participants was Ernst Dickmanns, which by many is regarded a pioneer in autonomous cars [9].

Especially two famous robots were developed in the EUREKA Prometheus Project, the VaMP and the VITA-2. In 1994 these two vehicles drove more than 1000 km in heavy traffic near Paris, France, on a three-lane highway without human intervention. By using systems such as computer vision these vehicles were able to demonstrate autonomous

driving involving lane changing and over-taking at speeds up to 130 km/h [19] already in the 1990s.

The final achievement of the EUREKA Prometheus Project was a modified Mercedes car driving from Munich in Germany to Copenhagen in Denmark and back again. Although humans interacted with the car from time to time during the trip it did drive up to 150 km on its own reaching speeds up to 175 km/h on the German Autobahn. After the projects achievements from its birth in 1987 to its end in 1995 many milestones and new inventions in autonomous vehicles were reached and developed. Many later projects in autonomous vehicles are based on the results, experiences and developments from this project.

2.2.3 Space Robots

One industry that drives the development of autonomous vehicles is the space industry. Ever since the space race started in the 1950s more and more sophisticated systems and robots have been developed and many of them have successfully landed on other planets. One example is the Mars Opportunity Rover vehicle which is a part of the National Aeronautics and Space Administration (NASA) Mars Exploration Rover Mission [20]. The Mars Opportunity Rover landed on Mars in 2004 and still (2008) function, even though it was only expected to function for approximately 90 days.

In comparison to autonomous vehicles on the Earth, autonomous vehicles for the space must often take many other factors into account. Rough surfaces and completely different climates are two examples. And maybe most important; if something does not work you might never be able to fix it and enormous amounts of money are wasted. In the case of the NASA Rover Mission in 2004 the temperature at the landing site could vary up to 120 degrees Celsius every day. Such conditions can not be found on Earth and are also difficult to artificially create. Another thing worth mentioning is that the actual conditions of a new planet are sometimes partially unknown before the first robot lands there. This makes it even harder [21]. From of the facts mentioned above it is clear that thorough testing and simulations are key elements for a successful space mission with autonomous space vehicles.

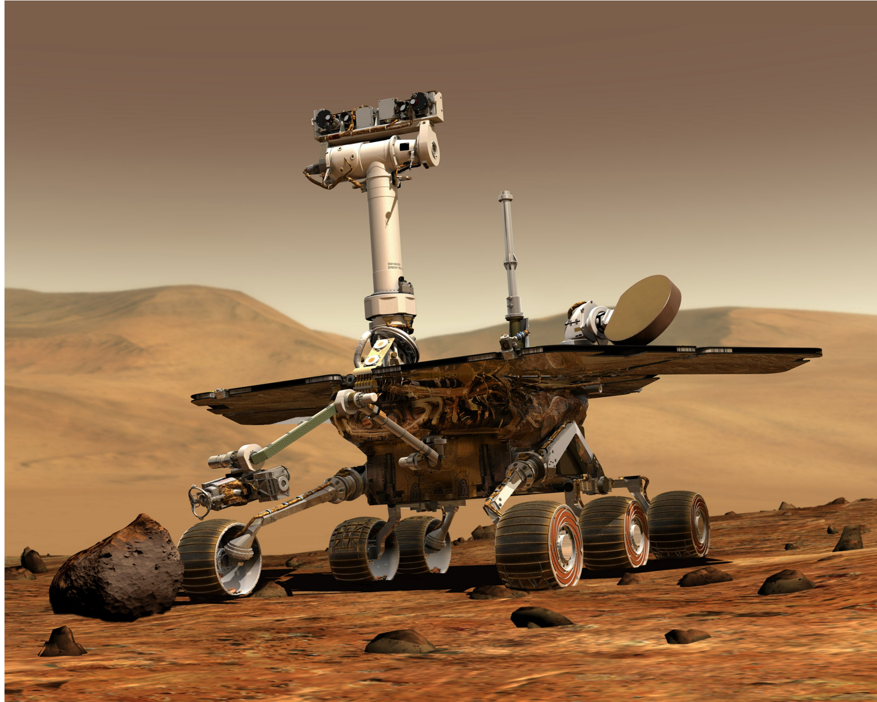


Figure 2.5: The Mars Opportunity Rover [20]

2.3 Available Subsystems

Building an automotive simulation system from beginning is a huge project which involves many time consuming and challenging tasks. Also, since there are many excellent subsystems available it is rather unwise not to take advantage of the work that is already done. In order to have the freedom of improving the subsystems, making user specific modifications as well as integrating towards external hardware, e.g. sensor and actuators, open source software is preferred. There are several well developed open source engines for both graphics and physics available as well as other useful libraries. Some of the subsystems used and also considered used for the automotive simulation system are briefly presented in the sections below.

2.3.1 SubSim and EyeSim

Simulators for various robots have been developed at UWA's Centre for Intelligent Information Processing Systems (CIIPS) before. SubSim [22][23] and EyeSim are two examples that are still in use. Using parts from these previous projects and gaining experience and knowledge from them will provide the automotive simulation system with

a good foundation which can speed up the project in its initial phase. SubSim is a simulation system for underwater autonomous vehicles which is freely available from UWA. The simulation system provides the user with the ability to customise the simulation scenario and to write own user or client programs (programs that control the behaviour of the underwater vehicle). These programs are written in C/C++ and compiled into a Dynamic Linked Library (DLL). Although this simulation system is for underwater vehicles, many sensors and actuators implemented for the SubSim will also be useful in a vehicle simulator, e.g. Position Sensor Device (PSD), compass, velocity meter and so on. SubSim and EyeSim's Application Programming Interface (API) with GET_DATA and SET_DATA for the different robot devices will also be implemented in a similar way for the automotive simulation system.

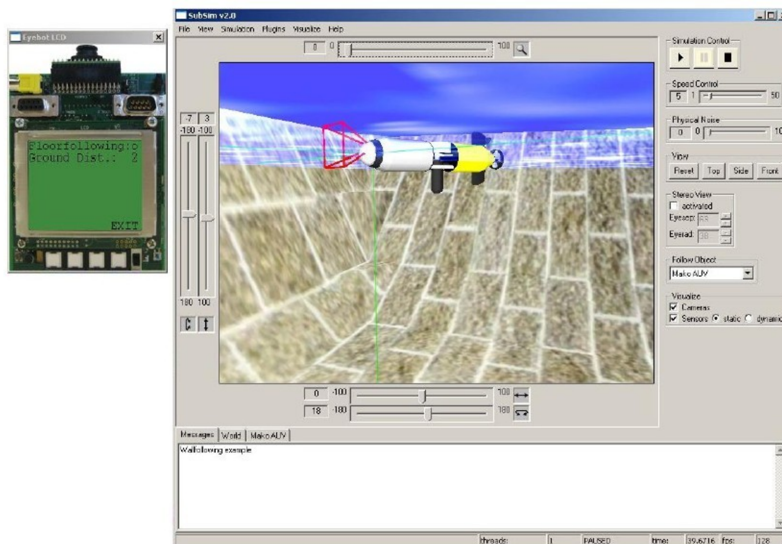


Figure 2.6: SubSim's graphical user interface [22]

2.3.2 Delta3D

The Delta3D [24] game and simulation engine is used in various sorts of game and simulation based projects. Especially the military seems to find the engine useful in their simulations, but driving simulators have also been developed based on Delta3D. Delta3D includes many features such as terrain editors, network capability, High Level Architecture (HLA) support among many others. Delta3D is a complete framework for game and simulation and includes networking, physics and graphics in the same package. This makes the whole framework quite large and some of the freedom of choosing different

subsystems disappears. After evaluating different approaches Delta3D was not chosen for the automotive simulation system, however looking at its implementation might be useful in the future.

2.3.3 The Open Racing Car Simulator (TORCS)

TORCS [25] is a racing simulator derived from an earlier simulator called Robot Auto Racing Simulator (RARS) which has ended its development. The simulator is under constant development for platforms such as Windows and Linux and features, help and specifications are well documented through user guides, forums and tutorials. Tutorials are available for creating a custom track from a Google Earth images which is interesting as the automotive simulation system will need virtual models of real cities in the future. Apart from the normal user controlled car, development of robots (cars) are also possible. This means creating a car with Artificial Intelligence (AI) that can drive on its own. These programs are similar to the user or client programs for the SubSim and something which will be implemented in the automotive simulation system as well. There are also competitions and championships for creating the fastest autonomous car to drive around a given track. As TORCS is open source we have the opportunity to use code fragments and also graphics such as textures from the system.

Figure 2.7: TORCS, an open source driving simulator [25]

2.4 Driver Assistance Systems

Apart from systems that have now become standard in most cars (ABS, ESP etc.) many cars in the high-end market have implemented many new and advanced systems in their cars. Below some systems will be briefly described in order to get an overview of what kind of driver assistance systems that are available, in development and use.

Adaptive Cruise Control (ACC) is an improved modification to the well-known cruise control. In addition to keeping the speed of the vehicle constant, ACC also adjusts the speed according to other cars and obstacles in front. This means that if a car in front of your car is getting too close your car will automatically brake in order to bring the car in front to a safe distance. There are basically two types of ACC systems available; one is based on laser and one on radar. Laser based systems are significantly cheaper than systems based on radars. Also the laser based systems are not as reliable as the radar based systems, especially in difficult weather conditions [26]. The first radar based ACC systems available to the public came in 1999 (Mercedes and Jaguar) and during the last years a number of car manufacturers have implemented ACC systems in their cars.

Various systems for lane detection are developed. These systems notify the driver in the case of an unwanted lane change. Realisation of such a system can be done using sensors that sense the lane markings or by using cameras and image processing. This can prevent drivers that tend to fall asleep during driving of driving off the road. This is one of the features in the Honda Intelligent Driver Support (HIDS) system [27].

Driving during night time is tiring and difficult, especially as you get older. Night vision cameras (e.g. infrared cameras) with monitors inside the cars can be used to assist the driver in spotting obstacles on the road. Night vision assistance systems can among others be found in selected Mercedes and BMW cars [28].

Blind spots around the car are a well-known problem and many have experienced dangerous situations due to them. Volvo's Blind Spot Information System (BLIS) [29] introduce cameras that notify the driver when objects are in the blind spot of the car.

Sensor technology is under constant improvement and will allow more complex and advanced sensors and systems in the future. Furthermore, more cameras are used for driver

assistance systems. Radars and laser based systems are also being used for ranging and distance measurements and will be more used in the future. As with all new technology it is often only available in the expensive models, however if the technology proves to be good and robust they will probably become standard among all car manufacturers.

More and more features are built into vehicles. The other way of tackling the problems is to build features into the road infrastructure as well to assist the systems that are already in the vehicles. During the Demo '97 (Automated Highway Systems) magnets where placed between the lanes in order to assist the vehicles in keeping within the lanes [3]. Later, more sophisticated methods, using Radio Frequency Identification (RFID) technologies, have been tested. An RFID system can provide the vehicle systems with a lot of useful information like location, intersections and obstacles [3][30].

3 AutoSim - The Automotive Simulation System

The automotive simulation system AutoSim is developed at CIIPS, UWA and is the work of several students doing their bachelor and masters projects the last year. Development and improvements of the system are expected to continue for several years. The goal of the simulation system is to be a highly modular and extendible system where new driver assistance systems can be tested. The image processing framework ImprovCV [31], described in chapter 6 is one system that will be used together with AutoSim to test such systems. The simulation system gives the developers of driver assistance systems a platform to test their systems in an environment similar to the real world. Currently the automotive simulation system is only available for Windows. However, all libraries and programs used in the project are cross-platform so a future version of the simulator for another platform should be possible with minimal effort.

The simulation system itself is based on a number of different and freely available libraries and programs. The most important ones are mentioned briefly in this chapter together with an overview of the simulator.

3.1 Overview of AutoSim

The simulation system is based on a client/server architecture. There are mainly two reasons behind the choice of this architecture. Firstly, the server and clients can be separated on different computers and the computations required for the simulations are distributed on several computers. Also, several clients are able to participate in the simulations. The server is responsible for doing the physics for the whole simulation scenario while the clients are doing the graphics. Hence, the server does not necessary need

a graphics card, but the clients do. Secondly, the server and client are communicating over an IP network which allows clients and server to be located at different locations. The architecture and a simplified overview of the communication between the server and clients are depicted in Figure 3.1.

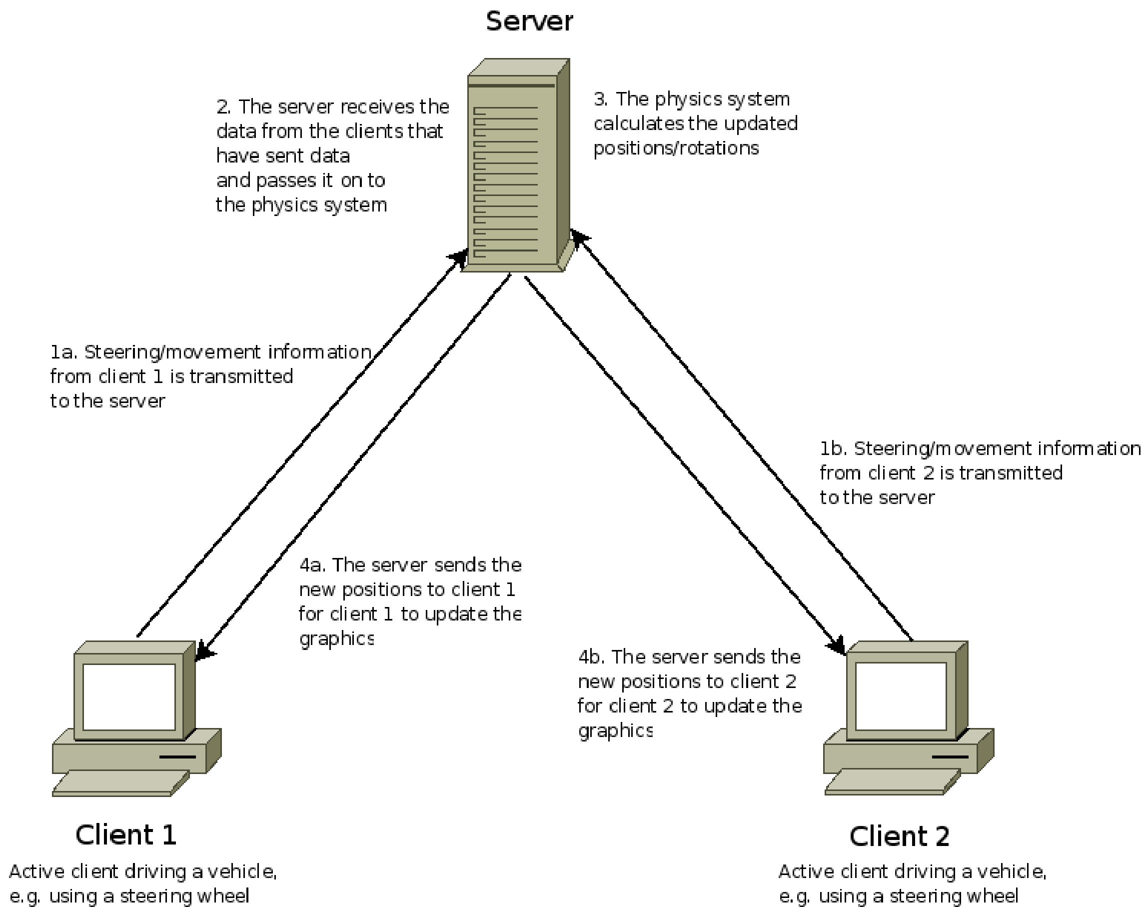


Figure 3.1: Overview of the client/server architecture of AutoSim

The behaviour of the vehicles (sometimes referred to as robots) can be specified in one or more user programs. These user programs are written in C/C++ and can be loaded into the simulator. Later, a user program for driving autonomously based on data from the lane detection in ImprovCV is used for my experiments. Another use of a user program is to connect a vehicle to a steering wheel. The user program takes care of connecting the given vehicle to the steering wheel.

All robots have an XML file that specifies what device a robot can access and the physics and position related to them. There are a number of currently supported devices such as

PSD, velocity meter, GPS and lights. If specified in the robot file the robot can access theses devices through a user program using GET_DATA and SET_DATA functions. The user programs and its structure and details are described in [32].

Similarly with robot files describing the robots, there are world files for describing the world. The appendix contains an example of how to create a world.

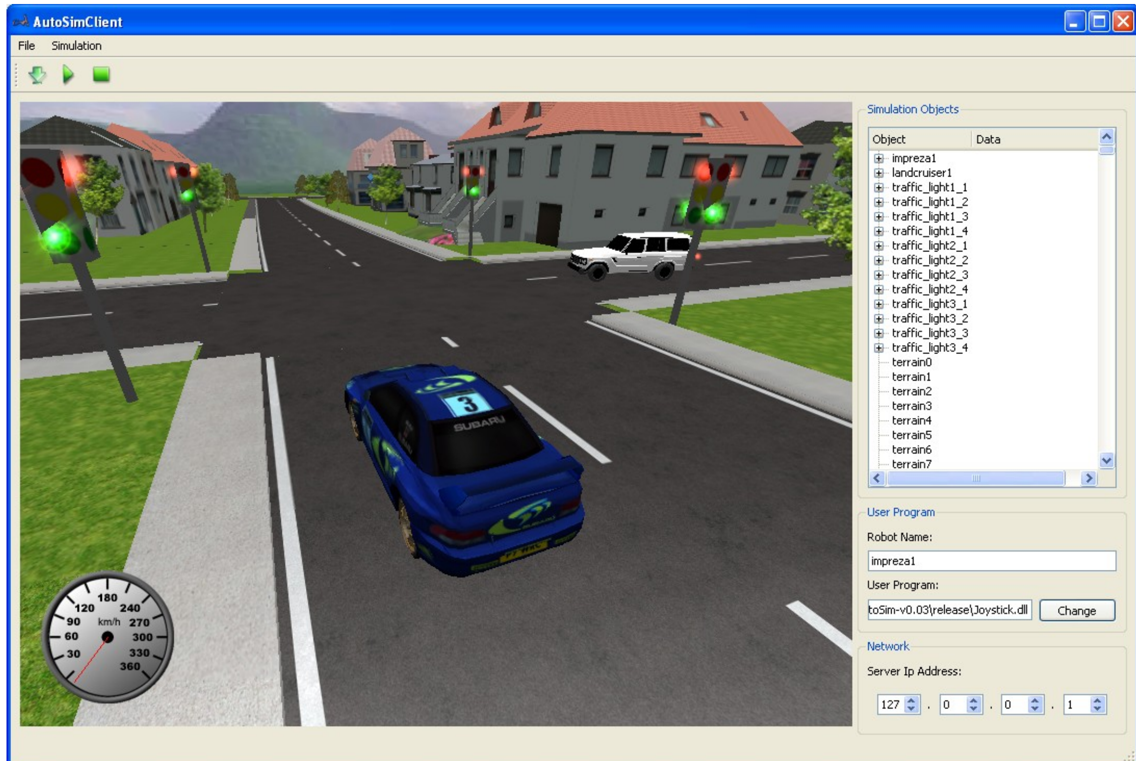


Figure 3.2: Screenshot from the AutoSim Client

3.2 Irrlicht

Irrlicht [33] is a 3D graphics engine and the one used in the automotive simulation system. It is also open source which makes it possible to make modifications according to the needs of the simulator. As of May 2008 some small modifications are in fact made. There are many freely available graphics engines available, however Irrlicht was chosen mainly because its good reputation, documentation and long list of features. Also, the engine is cross-platform, runs on Windows, Linux and MacOS, which means a future version for any of the platforms is possible.

The features of the engine make it easier to develop and improve the automotive

simulation system. Irrlicht has direct import of most 3D mesh formats and also image formats, which makes the job of finding and incorporating new models easy and minimal conversions are needed. Furthermore, a number of special effects are already implemented such as different types of lights, billboards (used in e.g. traffic lights and lights on the vehicles), skyboxes and particle systems (simulating fog, smoke, fire etc.). Irrlicht is used as the graphics engine in many applications and projects, also for the visualisation part in robot simulation packages [34].

3.3 OpenStreetMap

OpenStreetMap [35] is a free and editable map of the whole world made by users world-wide. The map project is similar to the well-known Google Maps, however Google maps are copyrighted and hence can not be used freely. As with Google maps some areas are not as good mapped, but the details of areas are increasing every day. Users all around the world can participant in making the maps better and better by using a GPS device, drive around and finally submit the GPS data to the OpenStreetMap web site.

For the automotive simulation system it is desired that the user can choose an area using the OpenStreetMap and further import the map area into the simulator to create a close to real representation of the real area or city. When a selected area is chosen the map can then be modified using the Java OpenStreetMap (JOSM) program [36]. The process of creating a world with the help of OpenStreetMap is described in the appendix.

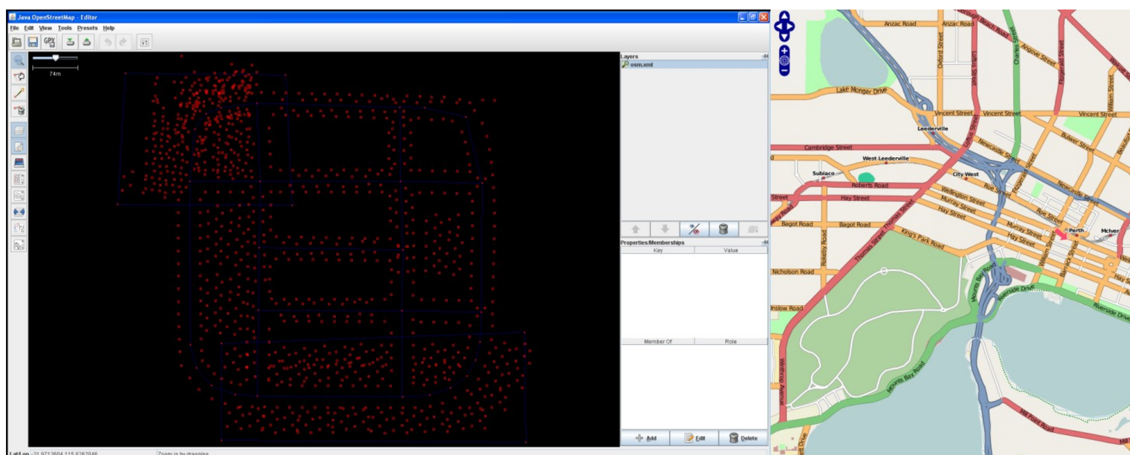


Figure 3.3: Left picture: The graphical user interface of JOSM. Right picture: The OpenStreetMap over UWA

3.4 TinyXML

As the name implies the TinyXML [37] library is a small and simple Extensible Markup Language (XML) parser written in C++. As the simulation systems consists mainly of XML files for storing preferences, robot information and world information an XML parser is important in order to retrieve and save data. TinyXML simplifies this job through its easy-to-use functions.

3.5 Blender

There are many commercial 3D modelling software packages available, e.g. 3D Studio Max and Maya, however, these programs are expensive to purchase. Blender [38] is another 3D modelling software package which is open source and available for all major platforms. The features and possibilities you have with Blender are similar to what you have in the commercial packages, but as the software is full of features and the Graphical User Interface (GUI) a bit different from other packages it takes a bit of training and time to get used to. However, there are lots of information available on the web such as forums and tutorials. Also, Blender has support for plugins and through this it supports import and export of the most popular used 3D mesh formats.

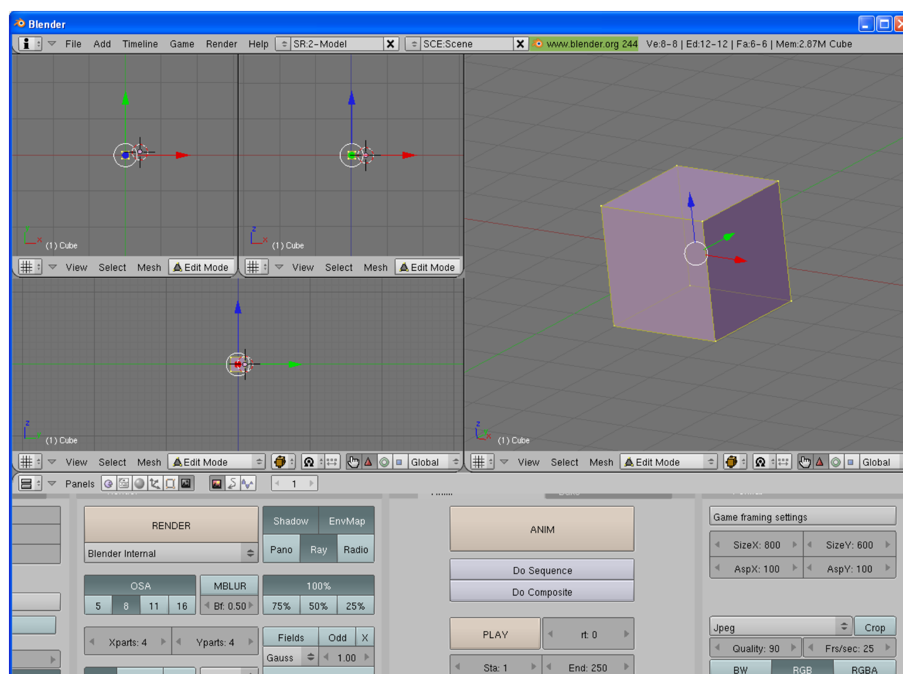


Figure 3.4: Blender's graphical user interface

4 3D Modelling and Content

In a simulation systems like AutoSim a lot of time has to be spent on the models that will take part in the simulation scenario in order to make it realistic and good looking. During the process of building up the simulation system and creating different scenarios many limitations in respect to the 3D models themselves were discovered. These limitations together with different techniques used will be described in this chapter. Detailed descriptions of how to create a virtual world, prepare new 3D models for the simulator are found in the appendix and in [39]. Note that robots are not only cars but all objects that have one or more user programs connected, for example the traffic lights. Also some minor differences in the way applications interprets coordinate systems will be discussed.

4.1 Differences in Coordinate Systems

Unfortunately, different applications interpret coordinate systems different ways which can result in confusion. The most notable difference in the programs and libraries used in the automotive simulation system is between the graphics engine Irrlicht and the 3D modelling application Blender. Basically this means that the up-axis and the axis pointing into the 3D space are swapped. The up-axis in Blender is called the Z-axis and in Irrlicht this is the Y-axis. Similarly the axis pointing into the 3D space in Blender is called the Y-axis and in Irrlicht the Z-axis. Also note that positive Z-axis in Irrlicht and positive Y-axis in Blender are pointing into the 3D space (which is common in computer graphics).

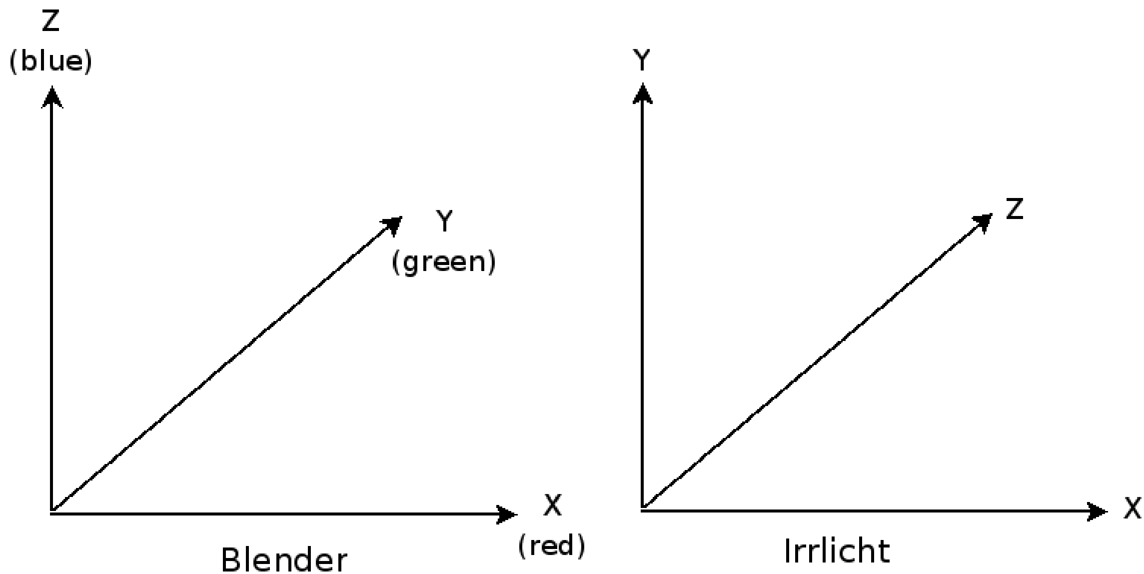


Figure 4.1: Showing the difference in coordinate systems for Blender and Irrlicht

As seen from Figure 4.1 both system are so called left-hand systems and the only difference is that the Z and Y-axis are swapped. This means that if you are using Blender to find the size of for example a building which you will use for collision detection in the simulator, you also have to swap the axis when inserting the numbers into the file containing the physics information. The swapped axis must also be considered when exporting models to a different format with Blender export plugins, more information can be found in the appendix.

4.2 Light Reflection

In computer graphics surfaces are often described using three components, ambient, diffuse and specular [40].

Ambient light is sometimes referred to as fill light. Applying ambient light or reflection on an object will only give the object a “flat colour”.

Diffuse reflection is the reflection from light on an uneven surface. One example of diffuse reflection is if you have a matte surface of an object.

Specular reflection is the often known as the perfect reflection of light, like in a mirror. Hence, if a light ray hits a surface with a certain angle the ray will be reflected equal to the

incoming angle. This is known as the law of reflection.

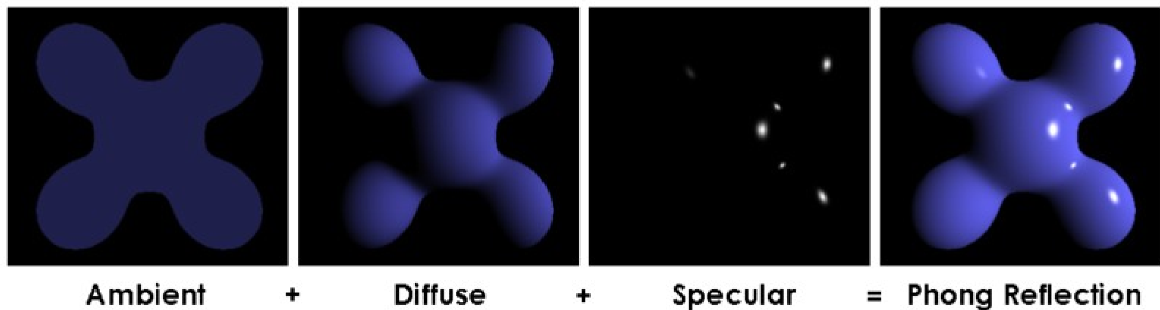


Figure 4.2 Illustrating the light parameters; ambient, diffuse and specular. Combination of the three is often known as Phong reflection or shading [40]

When using the Lightwave 3D format (.obj extension) which is a readable text format these parameters can be manually edited in the file to match and suit a particular simulation scene. Often the light reflection can cause confusion and manually editing the 3D files is a good way of debugging. Nevertheless, knowing the difference between ambient, diffuse and specular components is needed in order to understand it.

4.3 The Preferred Model Formats Used in the Simulator

4.3.1 OBJ and 3DS

Although Irrlicht supports direct import of several 3D model formats it is preferable to just use the .obj (Lightwave) and .3ds (3D Studio Max) format. By working with models and exporters there are a lot of small, but significant differences with formats that you have to be aware of. Hence, keeping the number of different formats to a minimum will reduce possible errors.

The advantage of the .obj format is that the model information is stored in readable text files. The .3ds format on the other hand stores the information in a binary format which makes it unreadable. A readable format is a big advantage. If a model is not looking as it should in the simulator, for example if the texture or material looks odd you can open the file in a text editor and manually edit the data. The other option is to edit the graphics data in a 3D modelling program, e.g. Blender.

4.3.2 The AS3D (AutoSim 3D) format

Building realistic worlds for the simulator requires a lot of 3D models. There are many free 3D models available on the web, e.g. houses and vehicles. However, many of these models are either to complex (consisting of too many polygons) or too simple (not realistic). A too complex model will slow down the performance of the simulator significantly and is therefore not wanted. The solution is to have models with minimum number of polygons and make them realistic using textures. Free models like this are however not easy to find. Hence we decided to buy a set of houses optimised for real-time applications. These houses are built using simple shapes, but textured with nice textures which make them look good.

To retain the rights of the creator of the models, the models had to be converted into our own proprietary 3D format. The format is an obfuscated version of their original format which makes it hard to use the models outside the simulator. A new file format loader for Irrlicht was developed in order to use the new 3D format. The models were given the extension *as3d* (AutoSim 3D).

4.4 Traffic Lights

Not only are the vehicles in the simulation system regarded as robots. All objects in the simulation system that are either controlled by a user or a user program are considered to be robots. This also applies to the traffic lights which are controlled by a user program to automatically change the between the states; green, yellow, red and red/yellow. Nevertheless, there is one difference between traffic lights and vehicles and how they are interpreted by the simulation system, namely the traffic lights are static robots. This means that if you for example place a traffic light two metres above the ground the traffic light will not fall down to the ground like a vehicle. The traffic light will just stay in the air. A result of this is that the traffic lights must be placed in the correct height (Y-axis), just above the ground. To get this absolutely right is time consuming so therefore the actual graphics pole of the traffic light is longer than it should be, however the physics is not. Hence you can place a traffic light for example 20 centimetres above the ground, but it will still look right. This is illustrated in Figure 4.3. Another thing to notice is that a static robot is only static as long as nothing is colliding with it. Crashing a car into a traffic light will

make the traffic light fall over. The reason why the traffic lights need to be static is simple. The physics box for the traffic lights is just a narrow high box which would be difficult to make standing unless the ground is exactly even, which it most likely is not.

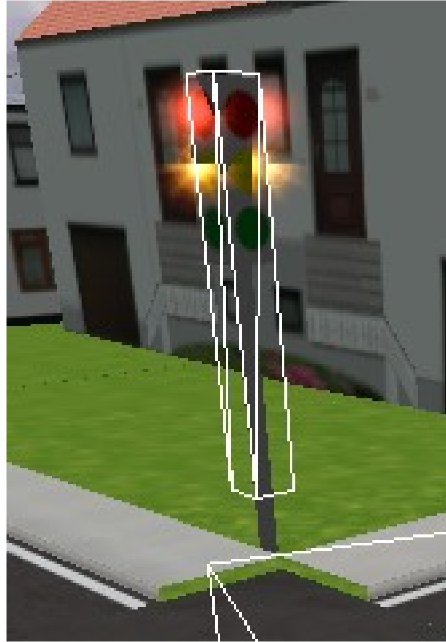


Figure 4.3: The traffic light including its physics box, as seen the actual physics box is a bit over the terrain

The current traffic light system is based on a simple fixed time control system. This means that after a given time the lights change for one direction of an intersection and at the same time the opposite happens for the other direction. Although this is a simple system it is still widely used in real traffic intersection, especially smaller ones. However, implementing a more complex dynamic control system with sensors for sensing vehicles is possible if desired. In that case a new user program for the traffic lights must be written.

As with other robots they are defined in the world file. This means that also their placement is specified in this file. Placing traffic lights in their correct position is a tedious process, but there is no need to change the rotation as all the sides of the traffic light have lights and hence the rotation will match the light states. Another useful program is developed in order to assist with placing robots, it is called *RobotPlacer* and maps between longitude and latitude to X and Z coordinates.

4.5 Other Objects

Loading complex models consisting of many vertices into the simulation system will slow down the system a lot. Using models with simple shapes and instead use good textures will give huge improvement in performance and rendered frames per second for the simulator. However, objects like trees are not easy to make realistic and good looking using simple meshes. The solution to displaying trees is therefore to simply display a tree as a billboard. A billboard in computer graphics is a texture that is always rotated straight at the camera. By using good textures of trees this method proved to be very realistic without making a big impact on the performance.

Figure 4.4: Showing the trees made as billboards

5 Physics Simulations

The simulations of the physics in an automotive simulation system like AutoSim are an essential part in order to get realistic results from the simulations carried out. As there are a vast number of aspects and relationships to consider in a complex scenario containing vehicles, other moving objects and static objects (like in a real traffic scenario) this is far from a trivial task to handle correctly. Luckily, there are physics engines available that will take care of the low level physics and provide its user with a more user-friendly interface.

This chapter will focus on the physics simulations used in AutoSim, the physics engines used and some aspects of vehicle physics. The experiments and results done with different physics engines are discussed in chapter 8.

5.1 Physics Engines

Physics engines are handling the physics for games and simulators such as AutoSim. Physics calculations related to a given scenario, e.g. scenarios in AutoSim can be very complex. Hence simplifications are done and different methods for solving the differential equations involved are used in different engines. The simplest numerical integrator used is the Euler method where

$y_{n+1} = y_n + hf(t_n, y_n)$, where h is the step size

The accuracy of the Euler method depends on the step size, a large h will give greater error but faster computation. On the other hand, smaller step size will give less error but longer computation time. Variations of the Euler method are often used in physics engines due to the simplicity [44].

The different ways of solving the differential equations is one of the reasons why different

engines not necessarily will behave similarly. To verify some of the results given later in the thesis two different physics engines were used for the experiments. The Bullet physics engine is the original engine used in AutoSim, but as AutoSim uses Physics Abstraction Layer (PAL) changing physics engine is fairly trivial. The second engine used for the experiments is the Newton Game Dynamics. A brief description of these two is given below.

5.1.1 Bullet Physics Library

The physics engine used for the simulation system is called Bullet [41] and is a free open source library used in several professional games and programs, e.g. in the Blender 3D modelling program. A physics engine is an essential part of an automotive simulation system in order to simulate real-world physics in a correct way to increase realism. The engine handles such things as gravity, collision detection and car physics. An important part is that Bullet contains vehicle physics model which is essential in an automotive simulation system. Physics calculations can be very complicated, but by using a physics library like Bullet a lot of the low-level details are hidden from the developer.

5.1.2 Newton Game Dynamics

Similarly with Bullet the Newton engine [42] is free to use, however its source code is closed. The Newton engine is said to focus on accuracy over speed and this might cause it to be a bit slower than other engines. However, this was not noticeable in comparison to Bullet in the later experiments with the engine. Similarly with Bullet, Newton is widely used for physics simulation and was the one used in SubSim [23].

5.2 PAL

The Physics Abstraction Layer (PAL) [43] can be seen as a physics wrapper library with extended abilities, hence an abstraction layer. PAL supports a long list of physics engines which makes it easier to test and benchmark your system with different physics engines in order to find the one that is most suitable for your system. As seen in [44] none of the tested physics engines are perfect, hence a good physic engine for one system is not necessarily the best for another system. In the automotive simulation system PAL is used

and as mentioned this makes it easy to replace the physics engine in the future if that is desired. PAL's ability to easily replace the physics engine was tested and demonstrated in my later experiments.

5.3 Vehicle Physics

The physics involved in moving objects are important for a simulation system like AutoSim. Although these details are handled by the physics engine some general concepts will be described in this section. Areas that will be covered are straight line physics, curves, wheels and springs. A short section on the implementation of the various aspects in the simulator is also given.

For this section the following notations will be used

$\vec{v}=$ a vector

$|\vec{v}|=$ magitude of the vector

5.3.1 Straight Line Physics

A number of forces are acting on a vehicle moving in a straight line. If the force from the vehicle engine was the only force involved, the vehicle would be able to accelerate to infinite velocity. However this is not the case.

The force contributed by the engine is given as [45]

$$\vec{F}_{drive}=\vec{u}\cdot\frac{T_{drive}}{R_{wheel}},\ \text{where}$$

$\vec{u}$ is the unit vector in direction of vehicle
R_{wheel} is the radius of the wheel

Furthermore, the engine's torque that is actually put on the wheels of the vehicle, T_{drive}, is given by [45]

$$T_{drive} = T_{engine} \cdot r_g \cdot r_d \cdot n \text{, where}$$

T_{engine} is the actual torque produced by the engine at a given RPM
r_g is the gear ratio
r_d is the differential ratio
n is transmission efficiency

As the torque varies according to the Revolutions Per Minute (RPM) for a specific vehicle and the ratios and efficiency also depends on a specific vehicle these numbers need to be looked up from the specification of the vehicle to be used.

Torque is given as force times distance. Hence applying 50 Newtons at 2 metres from the axis of rotation will give a torque of a 100 Nm (Newton meter). The torque for an engine is given as the force put on the drive wheel (T_{engine}).

F_{drive} gives the force in the direction of the vehicle. Now the forces that act in the opposite directions will be considered.

At high speed the aerodynamic drag is the biggest force acting in the opposite direction of the movement of the vehicle. The aerodynamic drag is proportional to the square of the velocity and is given as [46]

$$\vec{F}_{drag} = -C_{drag} \cdot \vec{v} \cdot |\vec{v}|$$

As seen from the equation above the velocity vector is multiplied with the magnitude of the velocity vector giving a new vector for the square of the velocity. The minus sign represent that the force is applied in the opposite direction as the velocity. Furthermore C_{drag} depends on the frontal area of vehicle, shape of vehicle and air density [46].

$$C_{drag}=\frac{1}{2}\cdot C_d\cdot\rho\cdot A\text{, where}$$

C_d is a constant
ρ is the air density $(\frac{kg}{m^3})$
A is the frontal area of the vehicle (m^2)

C_d varies with the shape of the vehicle and is usually measured in wind tunnels.

The final force that is acting in the opposite direction as the direction of the vehicle is the friction from the wheels, axles etc., known as rolling resistance. At low speeds the rolling resistance is the main resistance force and at high speeds the drag takes over. It is estimated that at 100 km/h (approximately 30 m/s) the rolling resistance is equal to the aerodynamic drag force [47]. Hence the C_{rr} can be approximated to be 30 times the value of C_{drag}.

$$\vec{F}_{rr}=-C_{rr}\cdot\vec{v}$$

Finally this gives the below equation for the total forces acting on the vehicle moving along a straight line

$$\vec{F}=\vec{F}_{drive}+\vec{F}_{drag}+\vec{F}_{rr}$$

According to Newton's first law an object where the total forces acting on the object is zero will continue at a constant velocity [48]. Hence if $\vec{F}$ from the above equation is zero the vehicle will move at a constant velocity. This will also be the vehicle's maximum velocity at a certain engine force. Moreover, if the forces acting on the vehicle is not zero the vehicle will follow Newton's second law [48]

$$\vec{F}=m\cdot\vec{a}$$

Acceleration, $\vec{a}$, in the above equation is the rate of change of the velocity [49] hence

$\vec{a}=\frac{d\vec{v}}{dt}$, or $\vec{v}(t)=\int \vec{a}\cdot dt$, furthermore the position vector, $\vec{p}$, can be obtained by integrating the velocity in respect to time.

$$\vec{p}(t)=\int \vec{v}\cdot dt,\ \text{or}\ \vec{v}=\frac{d\vec{p}}{dt}$$

When the vehicle is braking the engine force is replaced by a braking force acting in the same direction as the rolling resistance and the aerodynamic drag force. The equation is given as

$$\vec{F}_{braking}=-\vec{u}\cdot C_{braking}$$

This will cause negative acceleration and Newton's second law will still apply. The braking constant $C_{braking}$ depends on how much the driver of the vehicle brakes.

5.3.2 Wheels

The tires of a vehicle are an important part of the vehicle's behaviour as they are the only contact between the surface and the vehicle itself. It is also probably the most difficult part in terms of forces acting on the vehicle [46]. Steering of a vehicle produce an angle on the wheels (δ) which finally develop the force that turns the vehicle (yaw) [50].

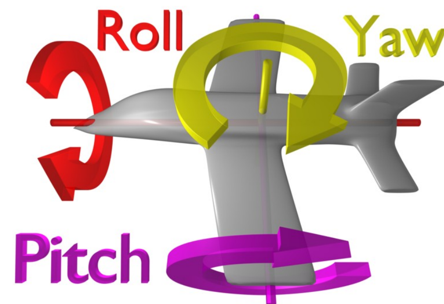


Figure 5.1 Roll, yaw and pitch axis definition [50]

Slip ratio is the common term used when discussing wheel forces. In a typical scenario with a vehicle driving straight forward the wheels with the power (front, rear or 4WD) will

experience some slip. This slip will produce a friction force and it is what is moving the vehicle forward. This means, e.g. for a rear wheel driven vehicle the rear wheels will actually rotate slightly faster than the front wheels which are just rolling without providing any forward motion. Example of a slip ratio curve can be seen in Figure 5.2. The important part to note from Figure 5.2 is the fact that some slip is required in order to get maximum force in the forward direction (or backward if driving in reverse). However, too much slip will reduce the force again. Also, a slip ratio of zero means that the wheels are only rolling without providing any drive. Curves like Figure 5.2 depend on factors like tire, surface and temperature so the one depicted is only applicable for demonstrating the concept of slip ratio.

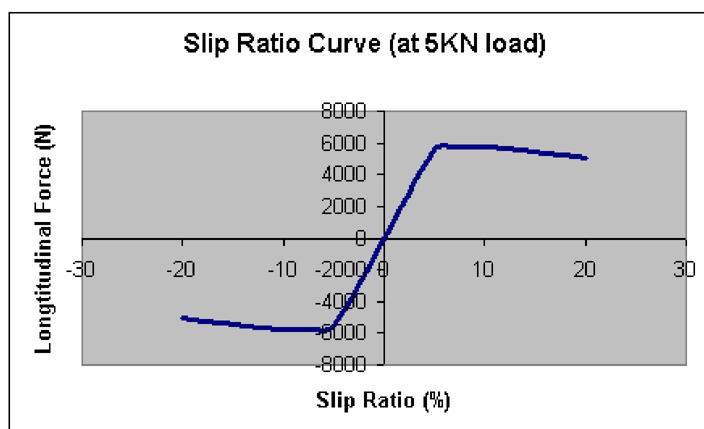


Figure 5.2 Slip ratio curve [47]

Similar concepts are also applicable when turning a vehicle, however it gets more complex. A model used in many applications is based on the work of Hans B. Pacejka, known as the Pacejka's formula [51]. Again, for real-time simulations the complexity of the models to be used must be considered as the computations might get too time consuming and be more accurate than actually needed.

5.3.3 Steering

In the case of the vehicle is turning there are two different scenarios to be considered. The first one is when steering in low speed and the other one is steering in high speed. Only low speed steering will be discussed here as this is most relevant to driver assistance systems and the simulator developed. High speed steering is when the angle of the front

wheels is not the same as the movement of the vehicle. This usually happens when the vehicle is driving too fast in a curve and either understeers or oversteers [52].

During low speed steering it is trivial to find the steering radius as long as the distance between the front and rear axle (L) of the vehicle is known. As shown in Figure 5.3 the turning circle radius (R) can be solved using the right triangle sine definition [53]

$$\sin(\delta)=\frac{L}{R} \Leftrightarrow R=\frac{L}{\sin(\delta)}$$

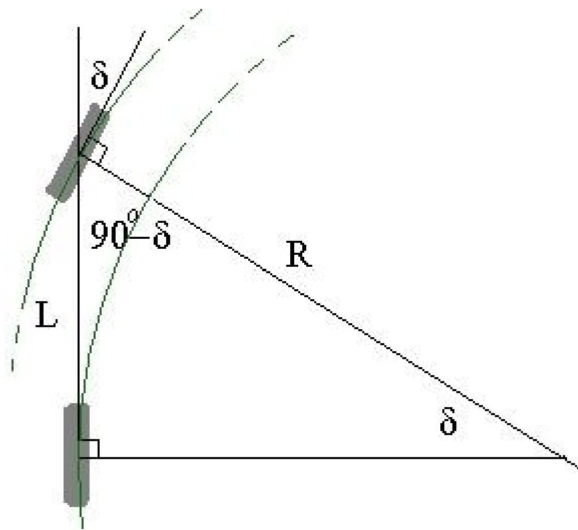


Figure 5.3 Illustrates a vehicle turning right at low speed [47]

The above equations give the turning circle radius and from this the angular velocity (in radians per second) of the vehicle can be found using [45]

$\omega=\frac{v}{R}$, where ω is the angular velocity

By using the two equations above the steering angle can be calculated based on the vehicle's velocity and turning circle radius.

5.3.4 Springs

Most springs are implemented and obey *Hooke's law,* this is also the case for the springs from the wheels to the vehicle chassis. Hooke's law states that the force in which a spring pushes back is linear proportional to the distance from its equilibrium (unstretched position) given as [48]

$\vec{F}=-k\cdot\vec{x}$, where

$\vec{F}$ is the force vector of the result
k is the spring constant (stiffness)
$\vec{x}$ is the vector of the direction and distance from the equilibrium

The negative sign in the equation indicates that the force is applied in the opposite direction as the load. Hence, if stretching a spring the force will be applied in the opposite direction and opposite when compressed.

5.3.5 Implementation of Vehicle Physics

The physics abstraction layer PAL implements the concepts about vehicle physics described in the above sections. However, most of the details are hidden from the user and only the respective parameters need to be considered when using PAL or any physics engines for that sake.

Initialising a vehicle in PAL is done using the below function

```
palVehicle::Init(palBody *chassis, Float MotorForce, Float BrakeForce)
```

The `Init` function transforms a `palBody` into a vehicle and the engine and brake force can be specified. The forces specified are the maximum values.

Furthermore, wheels to the vehicle can be initialised using the below function. For a normal car four wheels need to be initialised and further added to the palVehicle.

```
palWheel::Init(Float x, Float y, Float z, Float radius, Float width,
           Float suspension_rest_length, Float suspension_Ks,
           Float suspension_Kd, bool powered, bool steering,
           bool brakes)
```

Where the `x`, `y`, and `z` represent the position of the wheel relative to the vehicle's centre. `Radius` and `width` is the dimensions of the wheel. The suspension is specified as the `suspension_rest_length`, `suspension_Ks` and `suspension_Kd`. These parameters relate to 5.3.4 and Hooke's law where `suspension_rest_length` is the spring's equilibrium and `suspension_Ks` is the spring constant. Moreover, the dampening effect is given by `suspension_Kd`. The three last boolean parameters specify whether or not the wheel has power on it, able to steer and has brakes.

As the parameters for different vehicles are different the details are stored in the robot files for the AutoSim simulator. The robot files are based on an XML scheme and located in the folder `/robot/` in the installation folder of AutoSim.

5.4 COLLADA

In all forms of computing people are striving to develop an open interchange format for data between different applications. There are various formats available for 3D and physics information. However, some of them are proprietary and not all applications support all the formats. This makes working with these formats time consuming as much time is wasted on converting various formats into the desired format for the application you use. Furthermore, format conversions sometime introduce defects into the data.

To overcome the above mentioned issues the COLLADA (COLLAborative Design Activity) [54] has been developed. COLLADA's goal is to become the de-facto standard for sharing 3D and physics data between applications. As COLLADA stores the data in an open standard XML scheme the proprietary disadvantages of many 3D formats are removed. The format itself was originally developed for PlayStation 3 and PlayStation Portable by Sony. But as the format now is maintained and developed by the Khronos Group (a consortium) together with Sony several applications, game and animation studios have adopted the format.

As of version 1.4 of the COLLADA format, physics data was introduced to the format in addition to the original 3D data. This allows 3D and visual object to be linked to physical properties, all in the same file and format. However, for a standard physics format to be useful it must work with a range of different physics engines that have different constraints

and interpretations [54]. Examples of popular physics engines are: Bullet, Newton Game Dynamics, Open Dynamics Engine (ODE), PhysX and Havok. Hence, in order to achieve this compatibility among physics engines a generic representation of physics data is under development.

In a basic COLLADA file (with extension *.dae*) the scene can be represented by a visual and a physics scene. The visual scene describes the visual components related to the rendering of the scene whereas the physics scene consists of rigid bodies that are linked to their visual counterpart. In physics a rigid body is an idealisation of a solid body of finite size in which deformation is neglected. These are the objects that are of interest to the physics engine (that is based on rigid body dynamics) and through COLLADA various properties and geometries of these rigid bodies can be specified. Examples are friction, mass, position and rotation. Another aspect is the geometry which is important in collision detection. A reuse of the visual geometry may be used for this purpose, but is usually not done as visual objects may be of high complexity [54]. Simpler shapes or a collection of simpler shapes are often preferred for collision detection. E.g. in AutoSim the body of the vehicle is a simple bounding box around the body of the vehicle. Under many circumstances this will give a good enough approximation, but not in all cases.

A future use of the COLLADA format for the simulator might simplify the interchange of formats and merging the graphics and physics data into one file.

5.5 Scythe Physics Editor

The Scythe physics editor [55] is a modelling tool for physics simulations. As the 3D modelling tools mentioned in the previous chapter are mainly related towards graphics modelling (although Blender has some support for physics) this modelling tool allows you to simulate the physics of objects in a fairly simple way. Complex objects and shapes can be constructed using primitives and joints and then simulated in different environments and using different physics engines.

Natively Scythe supports the three physics engines; Newton Game Dynamics, Open Dynamics Engine (ODE) and PhysX. However, as part of the project a Scythe loader for PAL was developed which means that Scythe can be used with all the engines supported

by PAL [56]. The Scythe Physics Editor is used to create collision detection shapes for the objects in the simulator.

5.5.1 Scythe Loader for PAL

Scythe uses its own physics format known as *.phs*. As Scythe and its format is open source, a Scythe loader for PAL was developed. This extends the number of physics engine compatible with Scythe to all the engines supported by PAL [56]. The loader developed demonstrates the way a complex physics model can be used with PAL to do physics simulations.

As physics models are invisible a demonstration program was developed that displays the physics objects as graphics and hence makes more sense to users. Currently the following features are supported by the loader:

- boxes, spheres and capsules (cylinders)
- convex meshes
- hinges and ball joints
- mass

However, not all physics engines support all this features, e.g. convex meshes.

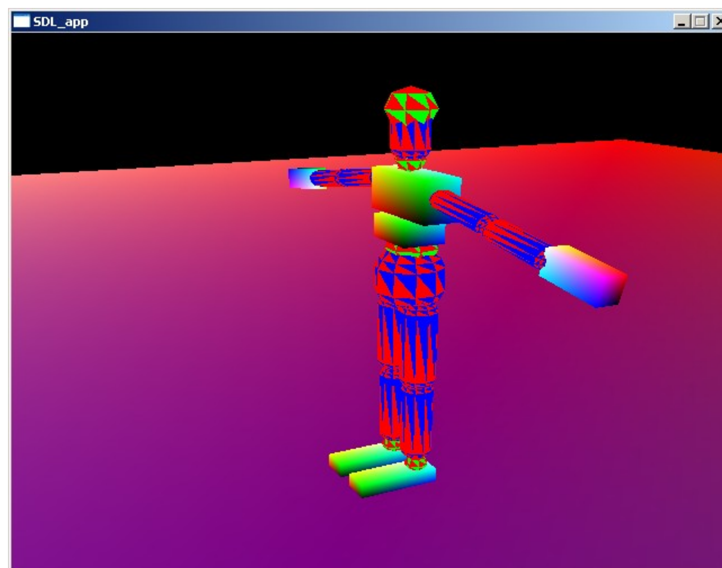


Figure 5.4: Simulation of a ragdoll in the PAL Scythe loader

6 Image-based Driver Assistance Systems

The image-based driver assistance system described in this chapter is based on the OpenCV computer vision library and the work done by S.A. Hawe while developing ImprovCV [31] at the University of Western Australia. The lane detection algorithm used for the later experiments is the algorithm developed for ImprovCV with modification to work with my experiments. This chapter will describe the lane detection algorithm and also the modifications done in order to use ImprovCV's lane detection as a driver assistance system for a vehicle in AutoSim.

6.1 ImprovCV

A widely used library for image processing is called OpenCV. This is an open source computer vision library originally developed by Intel, but now maintained by an online community of developers. The OpenCV library provides its user with already implemented functions of many frequently used filters and features used in the area of computer vision. Two examples are the Canny filter and the Hough transform which later is used in the lane detection algorithm which my experiments are based on. OpenCV is known to be used in many different areas of computer vision like face recognition, surveillance and robotic, in fact the 2005 DARPA Grand Challenge winner Stanley utilised the OpenCV library [17].

On the other hand, using OpenCV for testing various filters, parameters and such is a tedious process as it needs to be recompiled when parameters are changed. Thus, the image processing framework ImprovCV was developed at the University of Western Australia [31]. ImprovCV introduces a user-friendly user interface to OpenCV suitable for testing various filters with different parameters. Also, new filters can easily be developed and

included. By adding filters to a video sequence in real-time the effects of the filters can immediately be seen and further experimented with (Figure 6.1). The experiments on controlling a simulated vehicle (described in chapter 8) are based on the ImprovCV framework and its features.

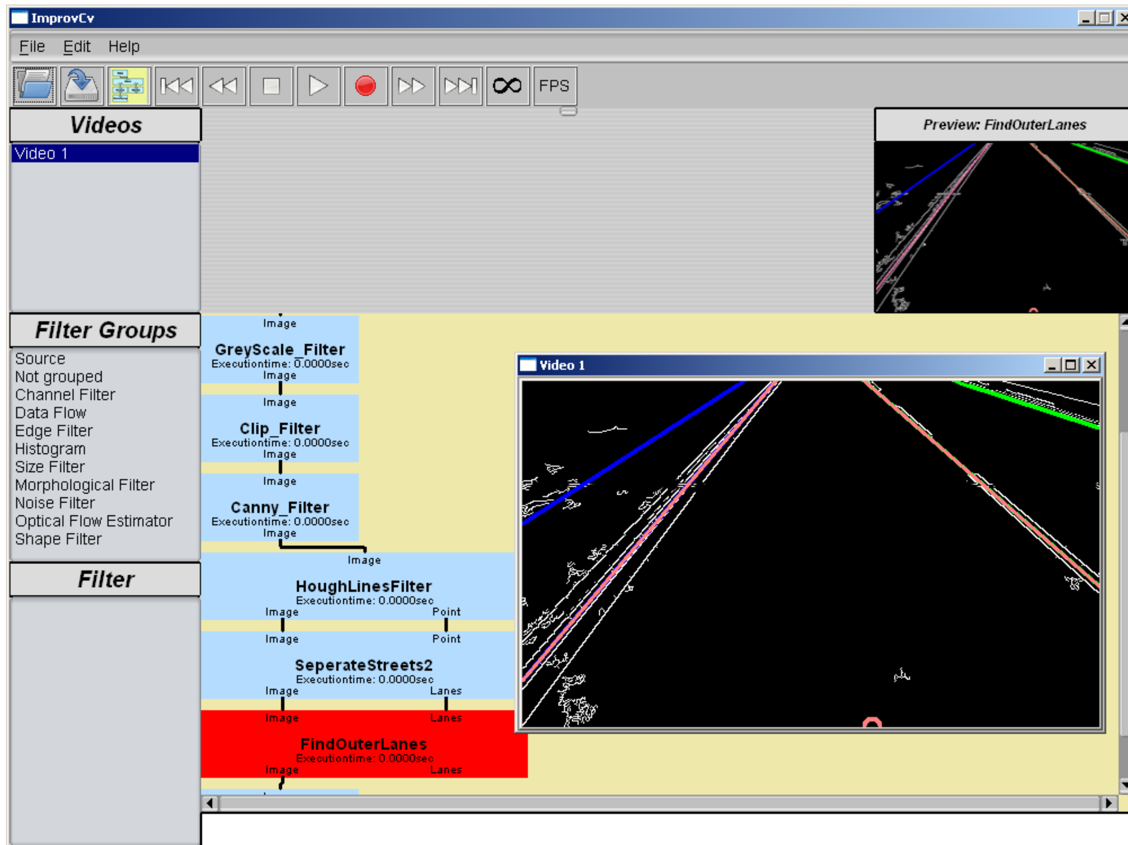


Figure 6.1: ImprovCV user interface. Filters can be applied by dragging and dropping

A specific setup of various filters can be stored as an XML file and later loaded into ImprovCV. Algorithms for doing e.g. lane detection which consists of many filters can hence be stored and later easily used. A part of a lane detection algorithm in ImprovCV can be seen in Figure 6.1.

There are currently two ways to input data into ImprovCV in order to do image processing. The first option is to use an already saved video file (AVI format) and the second options is to use real-time video from cameras. None of these methods are useful in order to do image processing based on the data from AutoSim. Hence a third option for video input into ImprovCV was developed. This method using shared memory is described in 7.2.

6.2 Lane Detection

Lane detection using image processing faces many issues and challenges. Firstly, lane detection systems should work regardless of the weather conditions. Sunshine, different lighting conditions and rain usually have negative effects on many systems and algorithms developed due to changes in visibility and reflections. Secondly, lane markings change from place to place and the type of road you are driving on. Depending on the system and what information that should be collected and analysed, dashed and solid lines with different colours might need a different interpretation. Old lane markings with poor sharpness and heavy traffic are also known to be issues with many image based lane detection systems [57]. A robust lane detection algorithm or system may have many applicable uses such as avoiding traffic accidents. Systems that warns the driver that he/she is about to make a lane shift are already available in vehicles. Another and more advanced use is systems that enable vehicles to autonomously follow a given lane without the driver interacting with the vehicle. A lane following system as mentioned above is simulated and described in chapter 8.

Several methods and algorithms are proposed for detecting lane markings in roads [58]. It is shown that developing a robust algorithm for a range of different scenarios and conditions is very challenging, hence proposed methods tends to focus on one particular problem or condition [59]. Nevertheless, many of the same techniques used are still the same. The techniques used for the ImprovCV lane detections are described in the below section.

6.2.1 The Lane Detection Algorithm

When image processing is to be used for real-time applications like in the example for lane detection to autonomously control a vehicle, it is desired to reduce the amount of calculations to be done as much as possible. As described in [31] it is estimated that up to 40% of a normal image used for vehicle image processing is useless in terms of lane detection. The upper part of the image usually consists of only sky and bottom part of the bonnet of the vehicle. By removing these parts of the image the overall computations needed later in the process are reduced significantly. This process is done using a *Clip Filter*. Furthermore, by first transforming the original image into a grey-scale image and

then further into a binary black and white image further reductions are achieved. Still the black and white image will contain the information needed to perform lane detection. This is achieved by an edge filter, in this case a Canny filter.

The next phase involves extracting the actual lane markings. A popular method is using Hough transforms, which is a way of finding imperfect instances of shapes, e.g. in an image. A slightly simplified Hough transform (in terms of calculations) known as probabilistic Hough transform was used for the lane detection algorithm in ImprovCV. Only a slight reduction in accuracy is the result [31]. Applying the Hough transform to a road scenario image will give many unwanted lines not relevant to lane detection. Horizontal lines found by the Hough transforms are the first ones to be discarded. To solve this and remove the unwanted lines and verify the lane markings, several methods are used and described in detail in [31].

Finally, the lane markings are detected and verified by comparing the current image frame with previous frames.

The lane detection algorithm used is summarised below with the actual name of the filters used in ImprovCV.

1. Using a *Clip filter* to reduce the image
2. Edge detection using a *Canny filter*
3. Finding lines in images using a *Hough lines filter*
4. Separating the actual lane markings from the Hough lines that are not lane markings using the filter *Separate streets*
5. Find outer lane markings that are of interest to the vehicle by using the *FindOuterLanes filter*

6.2.2 Finding the Outer Lane Markings

To be able to use ImprovCV's lane detection to autonomously drive a vehicle in AutoSim some improvements and modifications were done. Firstly, the lane detection in ImprovCV detected all visible lane markings, but only the two lane markings on each side of the vehicle are of interest for this control system (Figure 6.2). Hence a new filter for ImprovCV was created to find the two outer lane markings on each side of the vehicle. The

new filter takes two parameters as input as depicted in Figure 6.1, lanes and image. The image is the image that is modified in the previous filters and the lanes are the detected lane markings from the *SepreateStreets2 Filter*. Based on the lanes data from the *SeperateStreets2 Filter* the centre point between the outer lane markings can be found. The lane markings on each side of the vehicle will have the following equations

$y = m \cdot x + b$ for the lane on the left hand side and

$y = -m \cdot x + b$ for the line on the right hand side of the vehicles

Furthermore, the centre point between these two lines, and where the vehicle should be, can be found by finding the intersection points (x_0 and x_1) with another line e.g.

$y = 0$

Furthermore, the centre point can easily be found by

$$\text{centre point} = \frac{x_1 - x_0}{2}$$

This centre point can be seen in Figure 6.2 as the circle in the bottom of the figure. Before this centre point is fed back into the AutoSim control user program it is normalised to a value between -1 and 1.

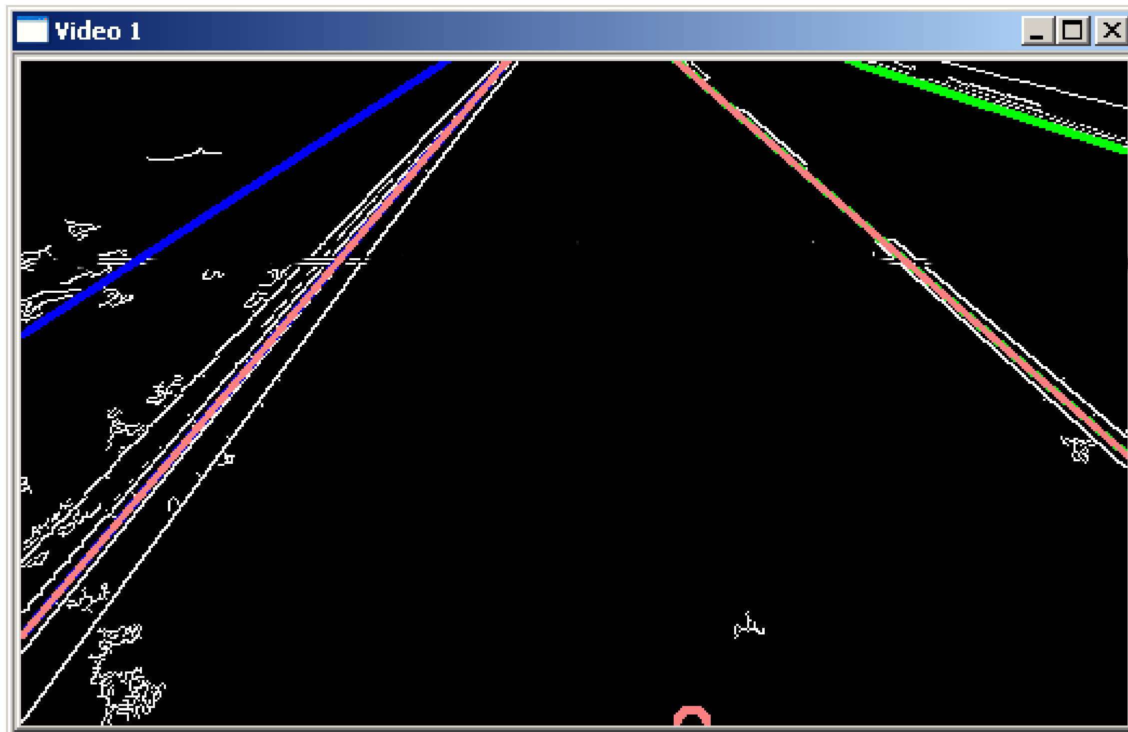


Figure 6.2: Lane detection where the two outer lane markings are found and the centre point calculated and displayed as the bottom circle

As seen in Figure 6.2 there are several lane markings detected in the scene. Finding the two lane markings that are of interest to us can be done by iterating through all the detected lane markings and find the two lane markings where the sign of the gradient swaps over from + to -. This is summarised in the pseudo-code below.

```
for (int i = 0; i < (number of lanes markings detected); i++) {
      if (current_sign != previous_sign) {
            // save the current i and this is where the
            // line markings swap from + to -, hence the
            // outer lane marking of interest
      }
      // save the sign of the m-variable of lane marking number i
      // as previous_sign
}
```

7 Control of a Simulated Vehicle

The goal is to make a simulated vehicle in AutoSim drive autonomously on a slightly turning road based on image processing from ImprovCV. ImprovCV is able to detect lane markings in the streets and based on this information it should be possible to keep the vehicle inside its correct lane of the street. Different control systems will be described in this chapter and later tested for the control of a simulated vehicle.

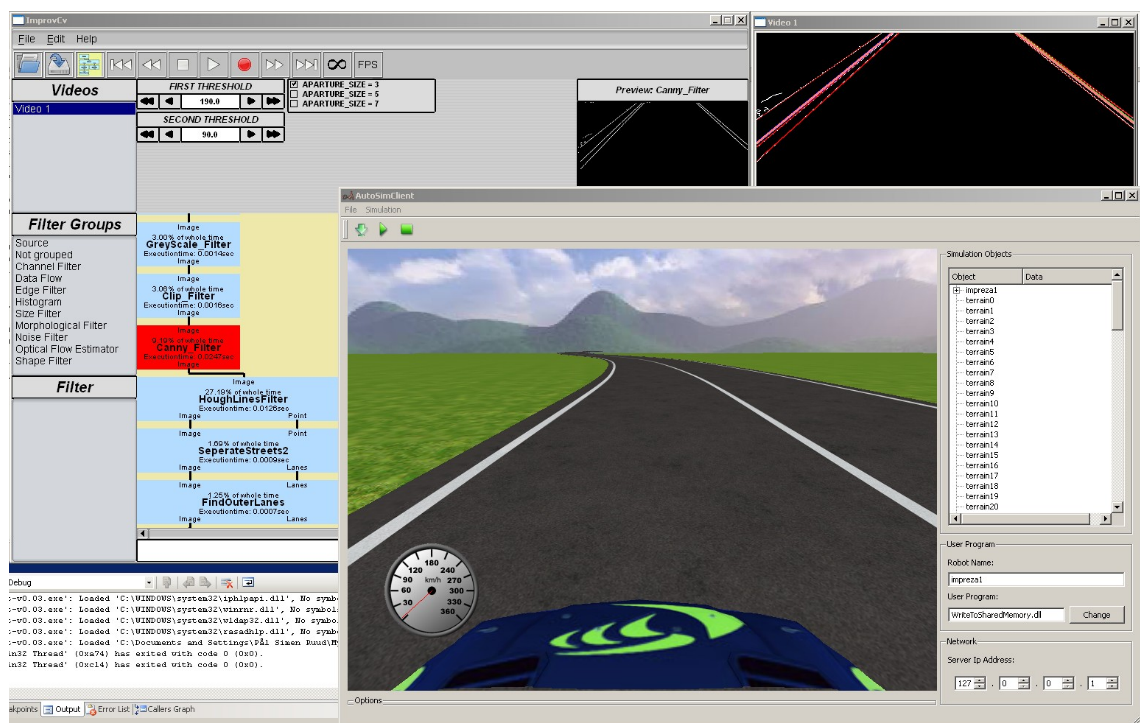


Figure 7.1 Lane detection in ImprovCV to drive simulated car autonomously

7.1 Control Methods

Controlling a system according to some specification is widely needed in the industry, e.g. controlling the speed of a car. Cruise control systems are designed to keep the speed of a

vehicle constant. Several methods can be used to achieve this. For this specific simulation four methods will be considered; on-off, P, PID and Fuzzy logic controller. A description of the four methods is given below. As it is the steering that is vital for the control of the vehicle being simulated, the four controllers are implemented for the steering of the vehicle. For maintaining constant speed of the vehicle a PID controller is used.

7.1.1 On-Off Controller

An on-off controller is often referred to as bang bang controller. The on-off controller is the simplest form of controlling a system and because of its simple nature it has some disadvantages. Maintaining constant speed of the vehicle will be used as an example for the on-off controller.

If the actual speed is lower than the desired speed the acceleration of the vehicle is increased and if the actual speed is higher than the desired speed the acceleration is decreased. By performing this check and adjustment at a fixed interval the desired speed will be achieved. However, the speed of the vehicle will continue to be either a little bit higher (overshooting) or lower (undershooting) than the desired speed and constantly change between faster or slower than the desired speed. An oscillating effect is achieved and this is not ideal. Nevertheless, on-off controllers are still used in many areas due to its simplicity.

7.1.2 PID Controller

A Proportional Integral and Derivative (PID) controller is a more advanced controller which removes some of the disadvantages of the on-off controller. By using a proportional, an integral and a derivative component a PID controller can be adjusted and tuned to fit many systems. For some systems a P, PD or PI controller gives a satisfactory result leaving out one or two of the components.

The proportional component of the controller determines the reaction to the current error in the system and is done by multiplying the desired value minus the actual value by a constant, the proportional gain.

$P = K_p \times (\text{Desired value} - \text{Actual value})$, where

$$(\text{Desired value} - \text{Actual value}) = \text{Current error in system} = e(t)$$

In order to achieve the desired effects for the system the controller should be used in the proportional gain should be tuned. An easy way of tuning a PID controller is described later on in this section. A large proportional gain will give the system quick response time however may lead to undesirable behaviour like overshooting and oscillation. Also, if error is reduced to zero it will output zero to the system, this will lead to that a P controller never settles at the desired value [60] seen in Figure 7.2. Depending on the proportional gain it might however settle fairly close to the desired value.

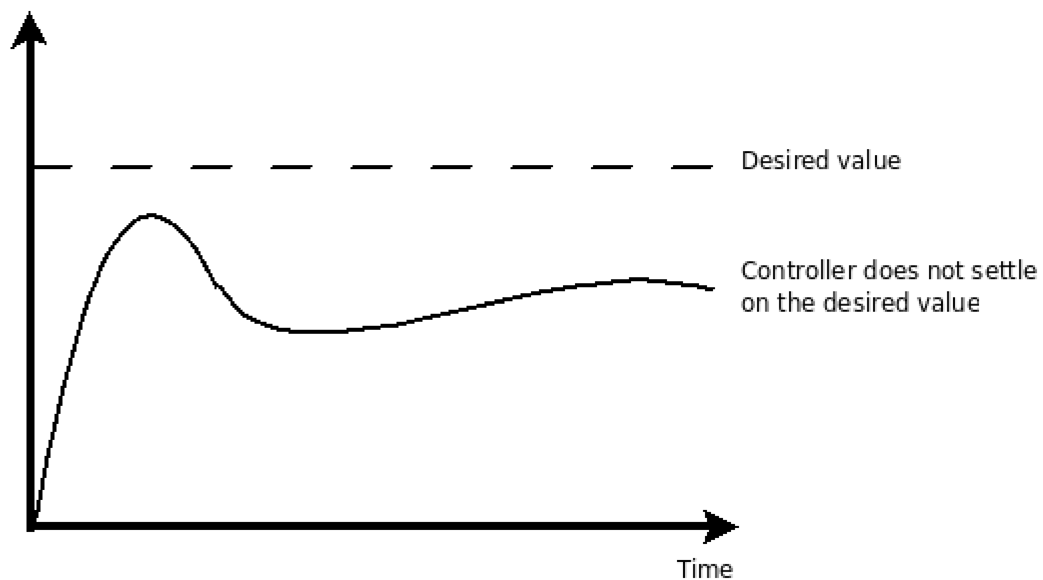


Figure 7.2: By only using the proportional component of a PID controller the controller will not settle at the desired value

To solve the issue with not settling at the desired value an integral component can be added to the controller. The integral component integrates the error (summing the error over time) and is multiplied to the integral gain, K_i.

$$I = K_i \times \int_{o}^{t} \text{e(t) dt}$$

The integral component can have two effects on the controller. Firstly, it can reduce the steady-state error to zero, hence the controller will achieve its desired value. Secondly, it can make the controller's response time faster by increasing the integral gain [60].

However, increasing the gain too much will again have downsides like overshooting and oscillation (depicted in Figure 7.3).

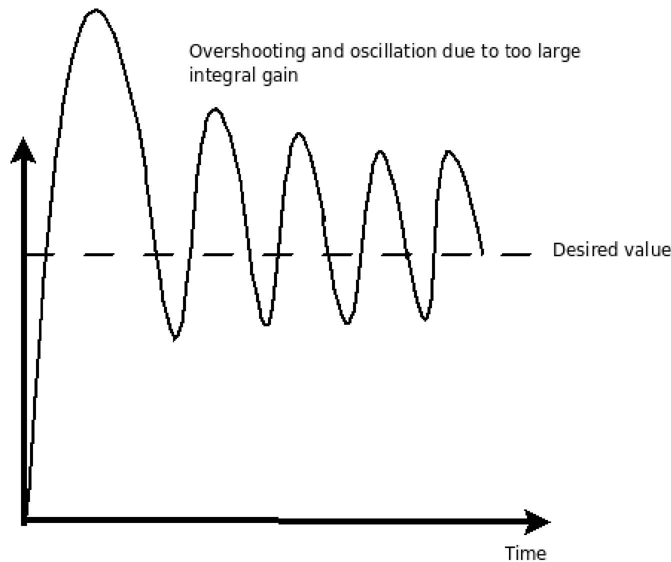


Figure 7.3: Overshooting and oscillation due to too large integral gain

The final component of a PID controller is the derivative component. The derivative component is mainly used in order to reduce the overshoot introduced by the integral component. Furthermore, the derivative component may give the controlled system more stability. As with the other components this also has a tunable gain, K_d.

$$D = K_d \times \frac{de}{dt}$$

Finally the PID controller will have the below equation. As it is sometimes sufficient or desired only to use some of the three components the gain of the components being left out can be set to zero.

$$PID = K_p \times (\text{Desired value} - \text{Actual value}) + K_i \times \int_o^t \text{e(t) dt} + K_d \times \frac{de}{dt}$$

The PID controller is summarised in Figure 7.4.

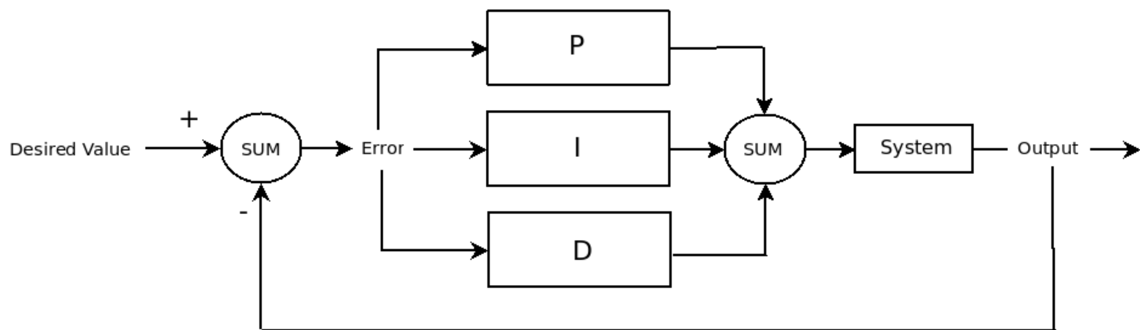


Figure 7.4: PID controller

Tuning the three gains in a PID controller is essential to make the controller behave the desired way. There are a large number of different methods developed [61]. For the purpose of the controller developed for controlling the simulated car, the tuning is based on [60].

1. Set K_i and K_d to zero and start increase K_p until oscillation occurs using a likely desired value.
2. If the system starts to oscillate, divide K_p by two.
3. Increase K_d and pay attention to its effects when changing the desired value by 5%. K_d should give a dampening effect on the curve.
4. Increase K_i until system starts to oscillate, then divide K_i by two or three.
5. Finally it is important to check if the controller gives you satisfactory results in terms of performance. Note that if the desired value changes, this might have a negative effect on the controller which is optimised for a certain desired value.

7.1.3 Fuzzy Logic

Fuzzy logic was introduced by Lotfi A. Zadeh in 1965 and is a concept of processing data that are imprecise or vague using an approximate reasoning [62]. This allows higher reliability e.g. when used in control of various systems that are noisy and do not have precise inputs.

Instead of modelling a system mathematically (which sometimes can be very hard or impossible) Fuzzy logic is based on simple linguistic rules of the form *IF x AND y THEN*

z. The rest of this section will describe the process of creating the linguistic rules, membership functions, inferences, fuzzification and defuzzification steps in order to design a Fuzzy logic control system. The description is based on creating a control system for the steering of a vehicle which is tested on the simulated vehicle in chapter 8.

The linguistic rules are based on a set of defined linguistic variables. Depicted in Figure 7.5 it can be seen that according to the values of *error* and *error_dot* (change in error) fed back from the simulator and ImprovCV the Fuzzy logic system will take the appropriate actions. For this specific example the linguistics variables are:

- error: too_far_left, too_far_right, just_right
- error_dot: going_left, going_right, no_change
- output: steer_left, steer_right, do_nothing

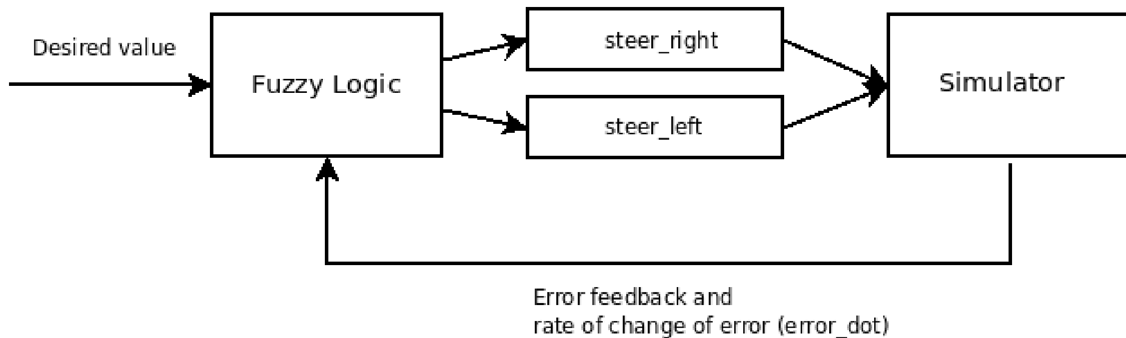


Figure 7.5: Fuzzy logic of a control system for vehicle steering

This example is based on the example in [63]. As shown in [63] the same method can be applied to a whole range of other control scenarios in a similar manor.

When the linguistic variables are defined the linguistic rules can be created as follow:

1. IF too_far_left AND going_left THEN steer_right
2. IF just_right AND going_left THEN steer_right
3. IF too_far_right AND going_left THEN steer_left
4. IF too_far_left AND no_change THEN steer_right
5. IF just_right AND no_change THEN do_nothing
6. IF too_far_right AND no_change THEN steer_left
7. IF too_far_left AND going_right THEN steer_right
8. IF just_right AND going_right THEN steer_left
9. IF too_far_right AND going_right THEN steer_left

Furthermore, the linguistic rules defined above serve as the basis for the membership functions, which in fact are graphs showing the relationship between the inputs. As there are two inputs, error and error_dot, these two will have different values since the rate of change of the error not necessary is the same as the error. An example of an error membership function is given in Figure 7.6.

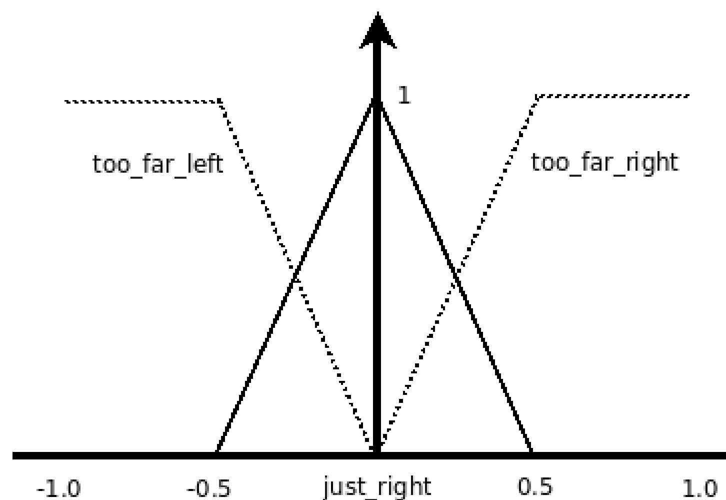


Figure 7.6: Membership function for error input for the steering control system

A similar membership function is created for the error_dot. However, the error_dot membership function will have different values along the x-axis. As the steering values used in AutoSim ranges from -1.0 to 1.0 these values are used in the above membership

function. The contribution from each of the inputs for a given value can now be calculated using the two membership function for the inputs *error* and *error_dot*. E.g. if the error is 0.25 then the contributions will be:

- too_far_left = 0.0
- just_right = 0.5
- too_far_right = 0.5

The same is done for the *error_dot*, note that the x-axis values in Figure 7.7 are just example values.

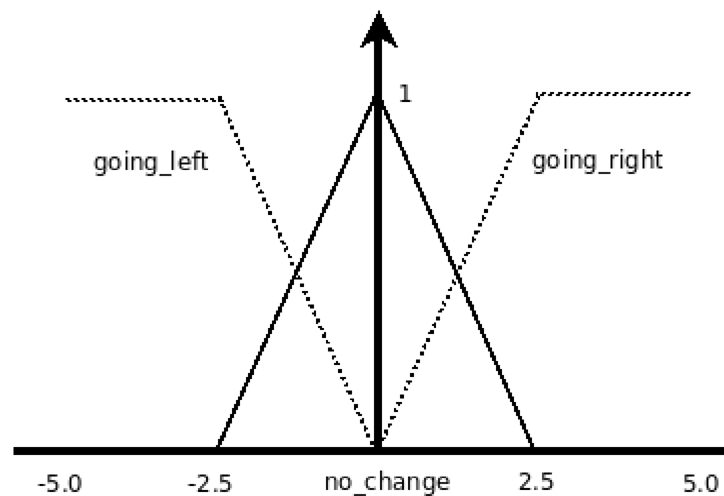


Figure 7.7: Membership function for the error_dot for the steering control system

Similarly as for *error* the contributions from *error_dot* at 1.25 will be

- going_left = 0.0
- no_change = 0.5
- going_right = 0.5

By using these values the interferences can be calculated based on the linguistic rules. For each rule the minimal value is to be chosen.

1. IF **too_far_left** AND **going_left** THEN steer_right → 0.0 AND 0.0 → 0.0
2. IF **just_right** AND **going_left** THEN steer_right → 0.5 AND 0.0 → 0.0
3. IF **too_far_right** AND **going_left** THEN steer_left → 0.5 AND 0.0 → 0.0
4. IF t**oo_far_left** AND **no_change** THEN steer_right → 0.0 AND 0.5 → 0.0
5. IF **just_right** AND **no_change** THEN do_nothing → 0.5 AND 0.5 → 0.5
6. IF **too_far_right** AND **no_change** THEN steer_left → 0.5 AND 0.5 → 0.5
7. IF **too_far_left** AND **going_right** THEN steer_right → 0.0 AND 0.5 → 0.0
8. IF **just_right** AND **going_right** THEN steer_left → 0.5 AND 0.5 → 0.5
9. IF **too_far_right** AND **going_right** THEN steer_left → 0.5 AND 0.5 → 0.5

Finally the interferences for all rules are evaluated and the output (what action to be taken by the system) can be calculated. There are several methods available, but the *root-sum-square* methods is recommended in [63] and hence used. When the *root-sum-square* method is applied to the values from the rules the defuzzification process can start which outputs a specific value to be used as the control value for the system.

For each output (in this case: steer_right, steer_left and no_nothing) of the current system the *root-sum-square* method must be applied.

Strength for output: steer_left= $\sqrt{rule3^2+rule6^2+rule8^2+rule9^2}$, where *rule3* is the interference for rule number 3.

An example for output *steer_left*

Strength for output: steer_left= $\sqrt{0.0^2+0.5^2+0.5^2+0.5^2}=\sqrt{0.75}=0.866$

Finally the strength for the three outputs is calculated and the defuzzification can be done. By using the strength for each output the final action to be taken by the system can be calculated as per below.

$$Output = \frac{\sum_{i=1}^{N} (strength_i \cdot centre_i)}{\sum_{i=1}^{N} strength_i}$$

, where N is the number of outputs (steer_right, steer_left and do_nothing) and $centre_i$ is the centre of that particular output from the membership function in percentage.

Hence, the output gives the number (in percentage) of the maximum that the steering should be adjusted to for the calculated *error* and *error_dot.*

Tuning of the Fuzzy logic can be done by changing the x-axis values of the contributions in the membership functions for the *error* and *error_dot*, e.g. wider or narrower triangles.

7.2 Communication Between the AutoSim Client and ImprovCV Using Shared Memory

In order to allow the two programs AutoSim and ImprovCV communicate with each other a shared memory approach was used. The shared memory allows different processes to access the same memory which is not the case in traditional programming. Usually the operating system assigns each process with its own memory space where the process can read and write.

In addition to shared memory there are two other possibilities of sharing information between processes without going through a network API.

- Read and write the shared information to and from a file.
- Share information that is located in the kernel of the operating system.

As reading and writing to file would be far too slow for sharing the data between AutoSim and ImprovCV and shared memory is regarded the fastest mechanism for communication between processes, the shared memory approach was chosen. For the shared memory

communication the Boost C++ library's interprocess features were used. The Boost library gives programmers a relatively easy approach to use shared memory without going into the low-level details of it. The way this works is that a memory region in RAM is allocated according to the size specified and then different processes can access this memory region by a given name. The different processes can also be assigned with different accesses depending on their role in the system.

Firstly, a user program for AutoSim was written. This user program extracts image from the simulator and stores the image as RGB (Red Green Blue) values into the shared memory. ImprovCV on the other hand is therefore able to read the image data from the same shared memory and use the image. However, one image will not make any sense and images have to be continuously written to shared memory from AutoSim. ImprovCV on the other hand has to similarly read the images continuously into a sequence which is usable to control a vehicle. As writing and reading at the same time to the same area of the shared memory will give unwanted results a method of using several image buffers was used. This seemed to be the best way, also in terms of performance and avoid getting choppy video sequences in ImprovCV. Nevertheless, doing lane detection in ImprovCV requires a lot of CPU time and the same for AutoSim. Hence, the two systems might be slow when communicating together, depending on the processing power of the computer running the two systems. Figure 7.8 shows how the image buffers in shared memory were implemented. A large chunk of shared memory was allocated to fit two images and a header containing which buffer currently being written to. This means that AutoSim client can write to buffer 1 while ImprovCV is reading from buffer 2. The header contains a count on number of images written. Experiments were made with using more buffers without any noticeable performance boost. In ImprovCV the reading from shared memory feature was implemented in the new filter *FindOuterLanes.*

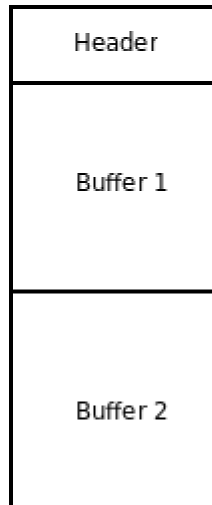


Figure 7.8: Shared memory with two buffers

Getting images from AutoSim to ImprovCV is one of the tasks related to shared memory. The other one is to feed information back to AutoSim in order for the simulated vehicle to keep inside its lane based on ImprovCV's lane detection. This is simply done by writing a floating-point number to another shared memory that another AutoSim user program can read from and hence make the necessary steering to stay in the correct lane.

8 Experiments and Results

Based on the control methods and lane detection described previously a series of experiments were conducted on autonomous driving using ImprovCV and AutoSim. Two different scenarios were created and each of the scenarios was tested with the four different control systems. Also, two different physics engines were used for the experiments. This chapter will give describe the scenarios and the results for each scenario in detail.

8.1 The Test Scenarios

Two different test scenarios were created in order to test the lane detection in ImprovCV and the control of the simulated vehicle in AutoSim.

1. Scenario 1 is a simple straight road where the car started slightly angled on the outer lane marking of the road. This scenario is created to be a good test scenario to tune the different controllers of the car. Also this scenario and the angle of the car will approximately reflect a normal scenario when a car is changing lane on a multi-lane highway. The objective of the simulated car is to stabilise itself in the middle of the lane as fast as possible without crossing over in the other lane and with minimum steering (oscillation).

Figure 8.1: Scenario 1 overview

2. Scenario 2 was created to test the simulated car's ability to drive autonomously for a long period of time on a turning road. A double S-shaped road was created and the vehicle was initially placed in the middle of the left lane of the road.

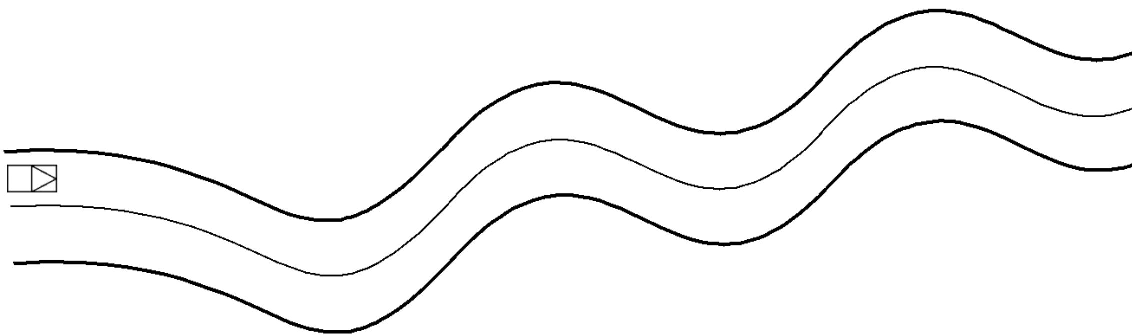


Figure 8.2: Scenario 2 overview (not exact representation of road)

8.2 The Controllers for Steering the Vehicle

For both the scenarios mentioned above four different controllers were used in order to see which one that was most robust and suitable for the steering of the simulated car. For all the scenarios the vehicle maintained constant speed after an initial acceleration phase. The below controllers were used

- On-off controller

- Proportional controller (P)
- Proportional Integral and Derivative Controller (PID)
- Fuzzy logic controller

As tuning of the controllers are essential in order to achieve the desired behaviour the first test scenario was mainly used for this purpose. This scenario is especially suited as the road is straight and the controller should stabilise itself as quickly as possible. After the tuning phase in this scenario the same controllers were tested on the scenario 2. However it turned out that tuning the controller for just scenario 1 did not give good results in scenario 2. Hence both scenarios were used for tuning.

8.3 Limitations

There are a few limitations related to the two systems AutoSim and ImprovCV that are important to keep in mind. Firstly, the lane detection algorithm used in ImprovCV only detects straight lane markings, hence it can not be used on a too curvy road. Secondly, running the two programs AutoSim and ImprovCV on the same computer is very demanding in terms of processing power. Although AutoSim can be run on two different computers (AutoSim server and client on separate computers) this will not result in noticeable higher performance. The graphics for AutoSim requires a lot of processing power and as the graphics are done on the AutoSim client which must be on the same computer as ImprovCV, the performance becomes an issue. Solutions to this problem are discussed in chapter 9.

The two limitations mentioned above results in that the roads can not be too curvy and that the vehicle itself must drive fairly slowly. A slow driving vehicle gives the system time to make the appropriate calculations related to the lane detection.

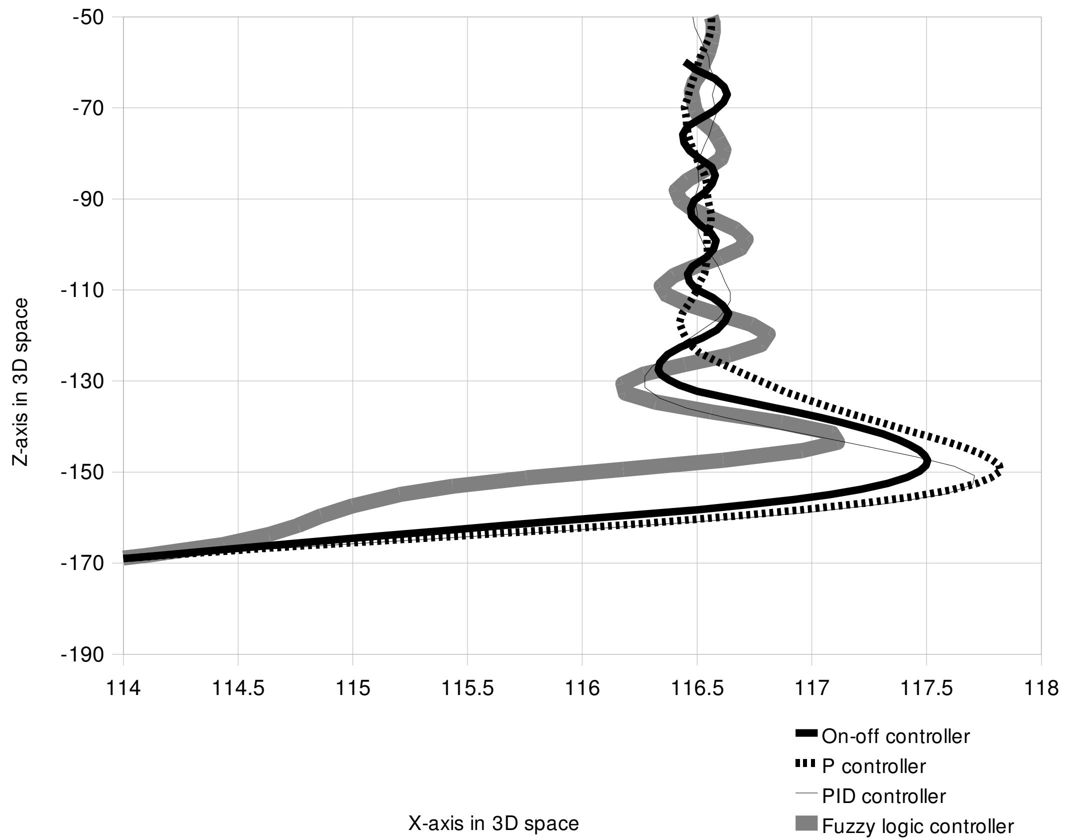
The simulations and experiments were carried out on a laptop computer with the following specifications:

- Dell Inspiron 8600
- Intel Pentium M Processor 1.7 GHz
- 1 GB of RAM
- ATI Mobility Radeon 9600 graphics card, 128 MB RAM
- Windows XP Professional Edition

8.4 Scenario 1 Results

Scenario 1 is especially suited to tune the controllers in addition to the real-world nature of the scenario. The four controllers were tuned for this scenario and the exact same controllers with the same parameters are used throughout the rest of the experiments. However, before the controllers were finally tuned, scenario 2 test were carried out and the controllers were further tuned for that scenario. The final tuning is based on getting acceptable results for both scenarios. As seen in Graph 8.1 oscillations and overshooting occur, but this is so the controllers also would manage the scenario 2 experiments in a satisfactory manor.

From Graph 8.1 the paths driven by the four controllers are plotted. As 118 on the x-axis is exactly on the centre line between the lanes it can be seen that all four controllers managed to stay inside the correct lane. As expected the on-off controller will keep oscillating forever and would not be very suitable for a steering control system in a vehicle. Too much oscillation will reduce the comfort of the passengers and increase the chance of motion sickness. A controller that stabilise fast with minimum amount of turning is desired. From Graph 8.1 the three other controllers will stabilise themselves eventually. The Fuzzy logic controller is eventually the most stable when z=-50 is reached, however it does have to make many correction. From Graph 8.1 it can also be seen that the use of a PID controller instead of P controller improved the path driven by reducing the overshooting.



Graph 8.1: Comparison of the different controllers

An image sequence of scenario 1 using the Fuzzy logic controller can be seen in Figure 8.3.

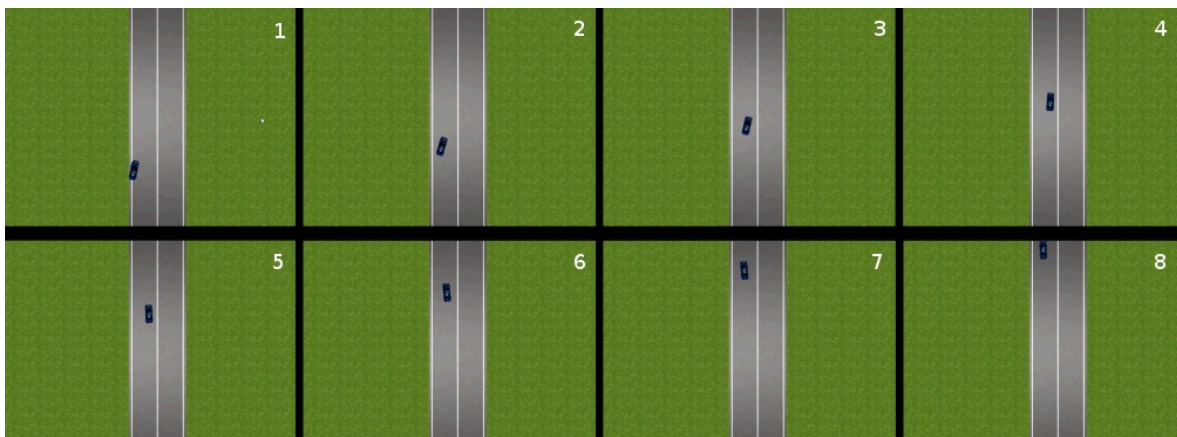
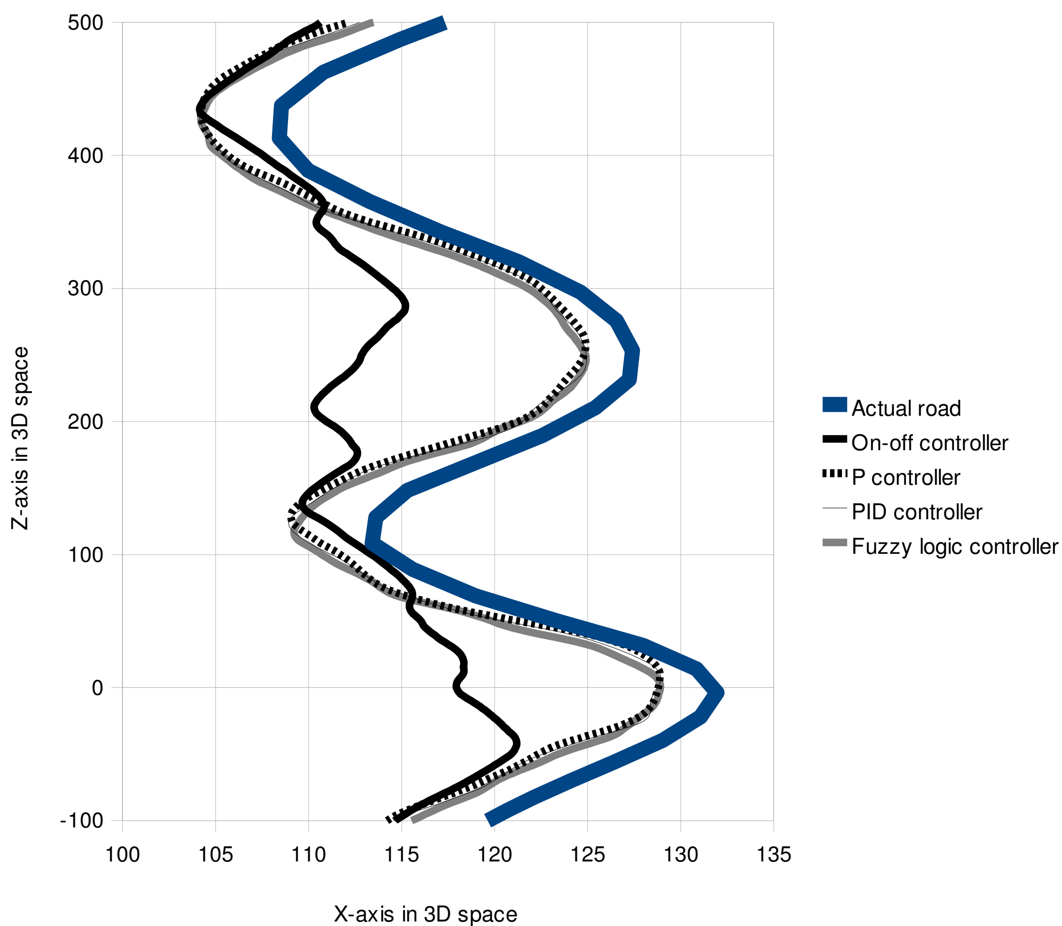


Figure 8.3 Image sequence of the start of scenario 1 using Fuzzy logic controller. The top left images is the first frame and the bottom right is the last frame.

8.5 Scenario 2 Results

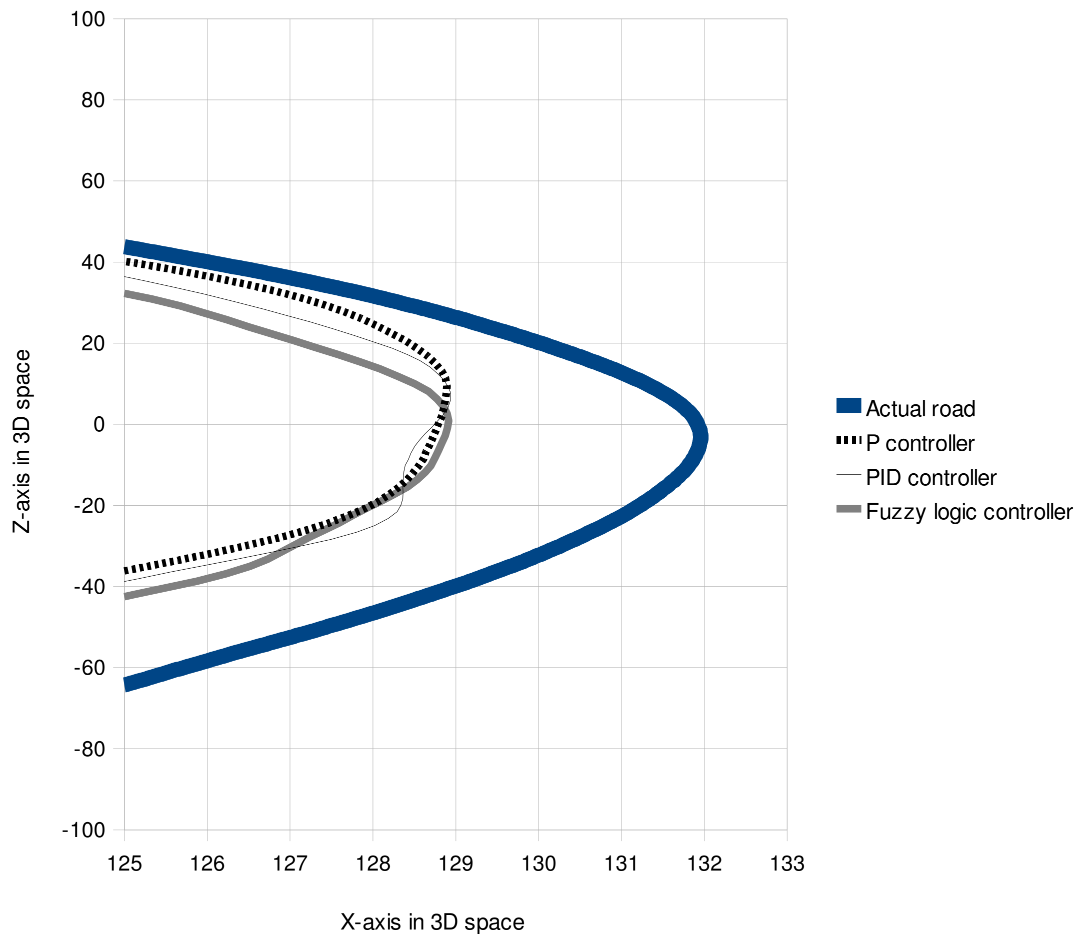
Similarly with the previous scenario the same four controllers were used in this scenario in order let the vehicle drive autonomously on the double S-shaped road. The vehicle position for each controller was saved in order to compare the paths each controller drove. Also to compare the paths driven with the actual road, the curvature of the road itself was saved. The paths driven and the actual road curvature are shown in Graph 8.2.



Graph 8.2: Paths driven by the simulated vehicle with different controllers compared to the real curvature of the road

Graph 8.2 clearly shows that the on-off controller was not able to stay on the road. And when the simulated vehicle first drove off the road it is random where it will go as the image input to the image processing does not contain lane markings. Furthermore, the

three other controllers seem to behave quite similarly and they all managed to stay in the correct lane and hence complete the road in a satisfactory manor. However, by zooming into an area of the road where there was a turn some differences in these three controllers can be seen. This is depicted in Graph 8.3.



Graph 8.3: Zoomed in on a turn to see precisely which controller controlled the vehicle using the most optimal path

Graph 8.3 shows that the Fuzzy logic controller drives the most optimal path of these three controllers. However the difference between the P, PID and Fuzzy logic controller are marginal.

8.6 Replacing the Physics Engine

As described in [26] there are differences in how physics engines behave. To verify the results from the above sections the same experiments are carried out with a different physics engine. The original physics engine in AutoSim, Bullet, was replaced with Newton Game Dynamics.

By running the same test cases again for the Newton physics engine different behaviour of the vehicle was experienced. Graph 8.4 shows the behaviour of the simulated vehicles in scenario 1. All of the controllers failed expect the Fuzzy logic controller. The reason behind this high failure rate is probably related to different interpretations between the physics engines. The simulations clearly showed that the acceleration of two vehicles using Bullet and Newton where quite different. Similar difference might be the case for the steering and this is probably the reason for the high failure rate. Also, when running the experiments for the Bullet physics engine in scenario 1 it was seen that ImprovCV had problems with detecting the correct lane marking and was close to fail as well (Figure 8.4). However, for the Bullet physics engine ImprovCV recovered and detected the correct lane markings just in time, this was not the case for the Newton physics engine and three of four controllers failed.

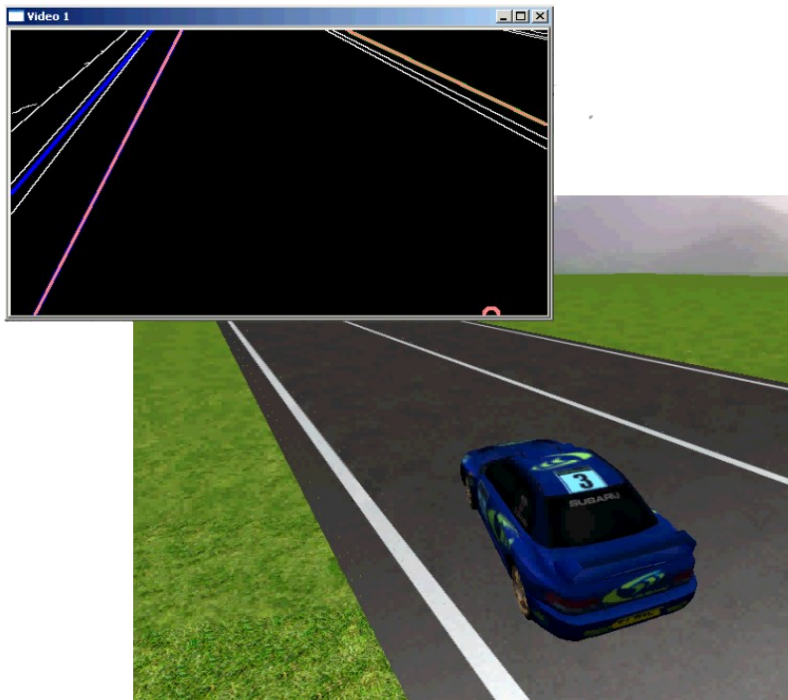
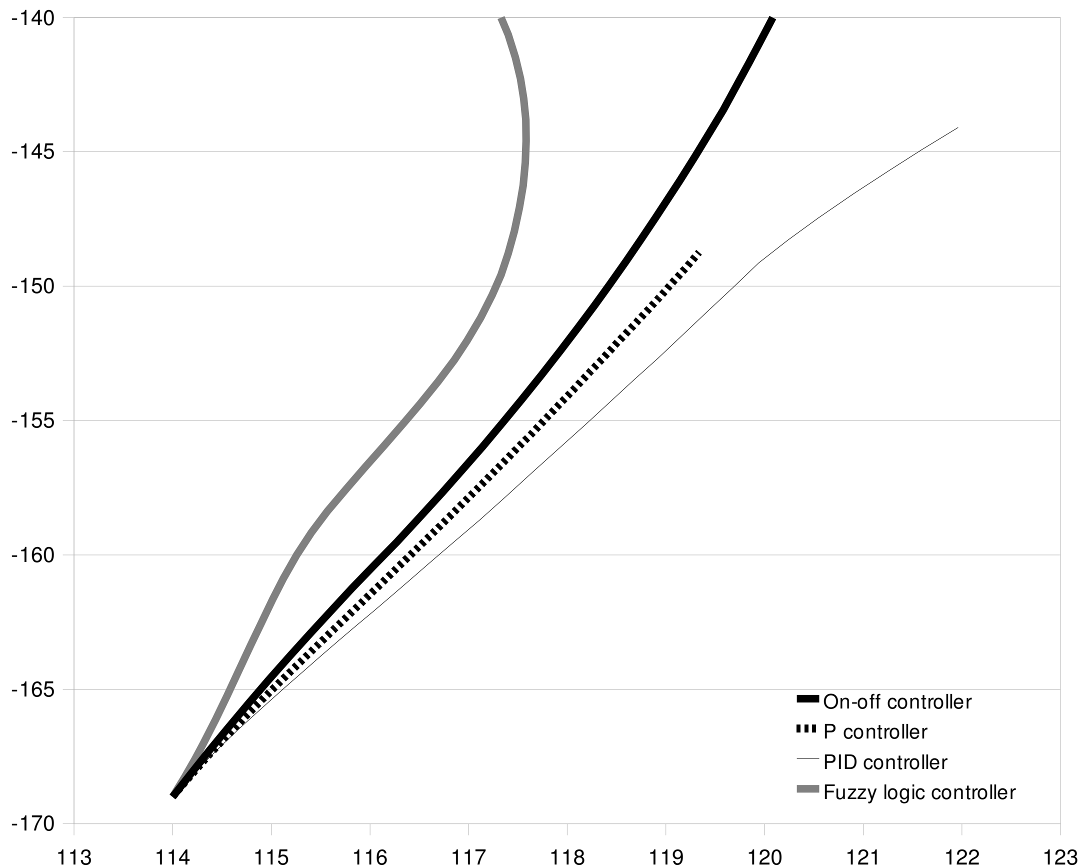


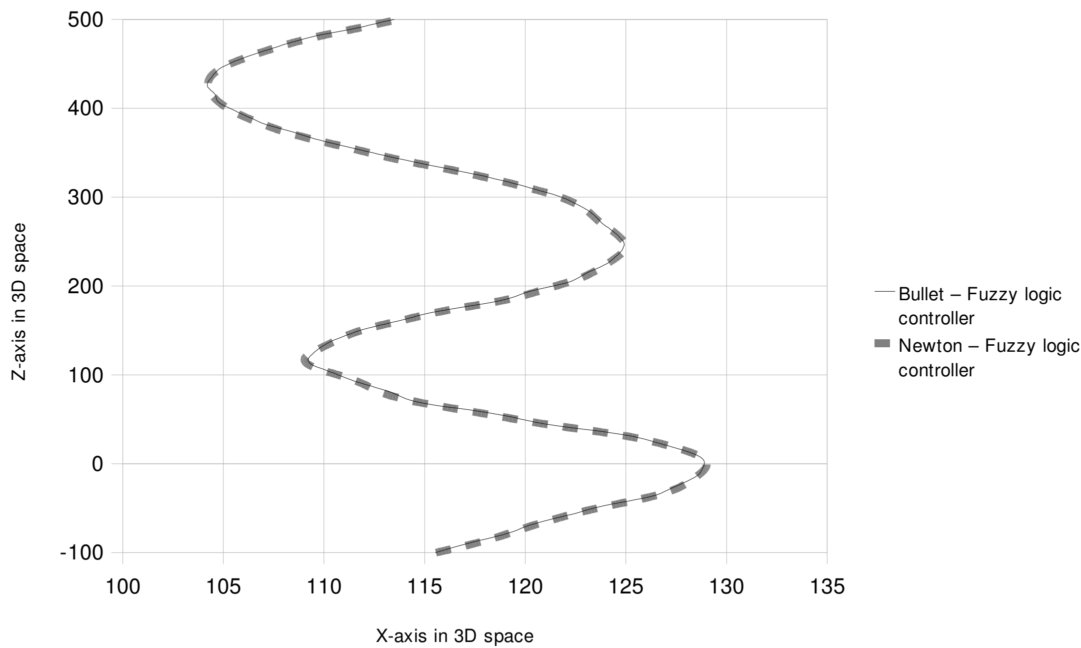
Figure 8.4 ImprovCV detects incorrect lane markings when vehicle is turning too quickly



Graph 8.4: Comparison of controllers for scenario 1 using Newton physics engine

Although the failure rate for scenario 1 was high, the results for scenario 2 proved to be consistent with the results from the Bullet physics engine. All controllers finished the scenario except the on-off controller.

As with the Bullet physics engine the Fuzzy logic controller also proved to be the best for the Newton physics engine. Graph 8.5 compares the path driven by the Fuzzy logic controller for both Bullet and Newton physics engine. As seen, the paths only differ slightly.



Graph 8.5: Comparison of the Fuzzy logic controller using Bullet and Newton physics engine

8.7 Robustness of the System

When replacing the physics engine used by AutoSim different behaviours were experienced. Scenario 1 appeared to be more challenging with the Newton physics engine and hence some of the controllers failed to complete scenario 1. In terms of scenario 2 similar results were gathered from using Newton as with Bullet and again the Fuzzy logic controller seemed to be the most suited controller.

As the Fuzzy logic controller managed to both complete scenario 1 and 2 with both Bullet and Newton verifies the robustness of the controller and its ability to behave satisfactory in different environments. By verifying that the controller works for two different physics engines this also enhance the chance of it would work in a real physics environment on a real vehicle.

Tuning one of the controllers for a given scenario proved to be fairly easy with respect of getting satisfactory results. However, tuning a controller in order for it to work with the two physic engines and different scenarios proved to be more challenging. Tuning e.g. the PID controller to be perfect for scenario 1 did not really work too well for scenario 2 and hence a balance in the tuning had to be made in order to complete both scenarios for a

given controller. Furthermore, replacing the physics engine showed that tuning a controller for one physic engine does not necessary mean it will work for another physics engine.

If the vehicle makes to sudden and sharp turn in the lane this will make the virtual camera attached in front of the vehicle to loose the sight of one of the outer lane markings (seen in Figure 8.5). Loosing the sight of one of the lane markings will make the vehicle drive in a random manor and probably off the road. Hence, if a large correction is received from ImprovCV to AutoSim the vehicle must adjust the steering gently. Changing the virtual camera position of the vehicle and the camera's height and width may improve this. Also, parameters for the Clip Filter in the lane detection algorithm can be adjusted and possibly improve the performance and make the system more robust. The comparison of the output from ImprovCV and the output of the four controllers shows that although a large correction is needed the vehicle is only steering gently (Graph 8.6).

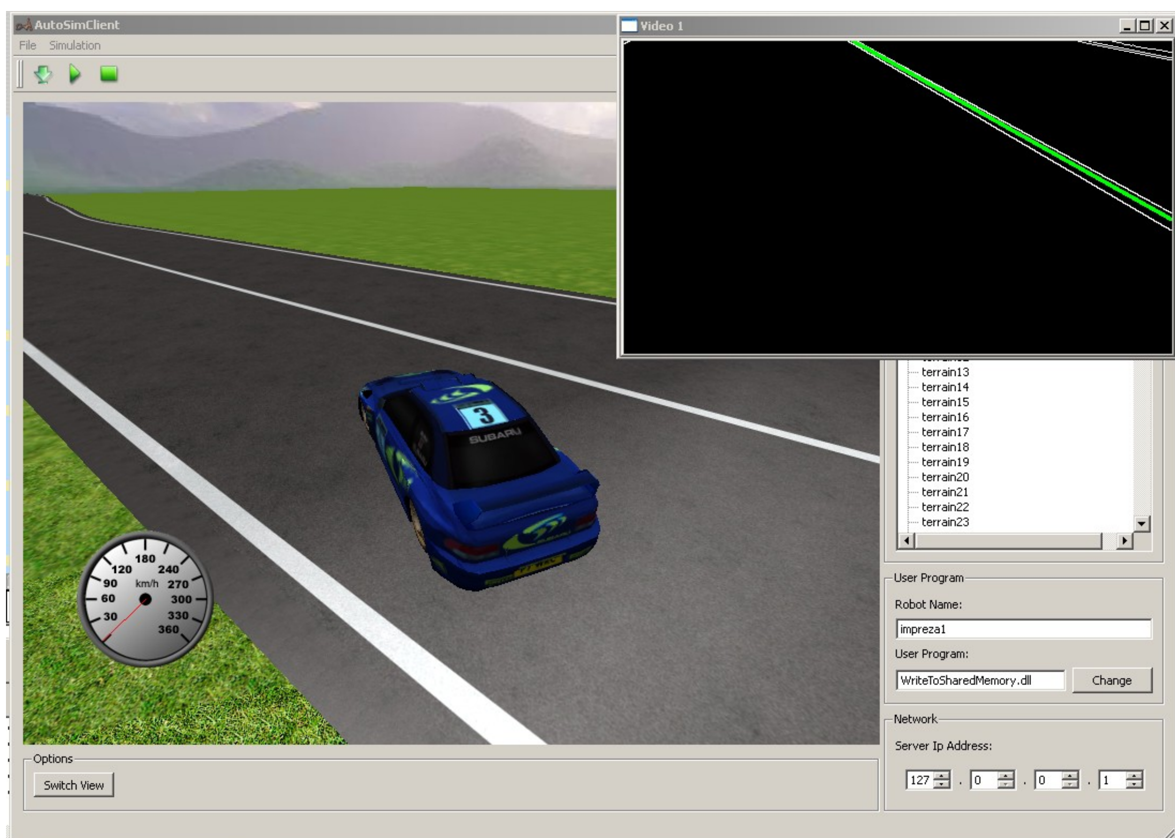
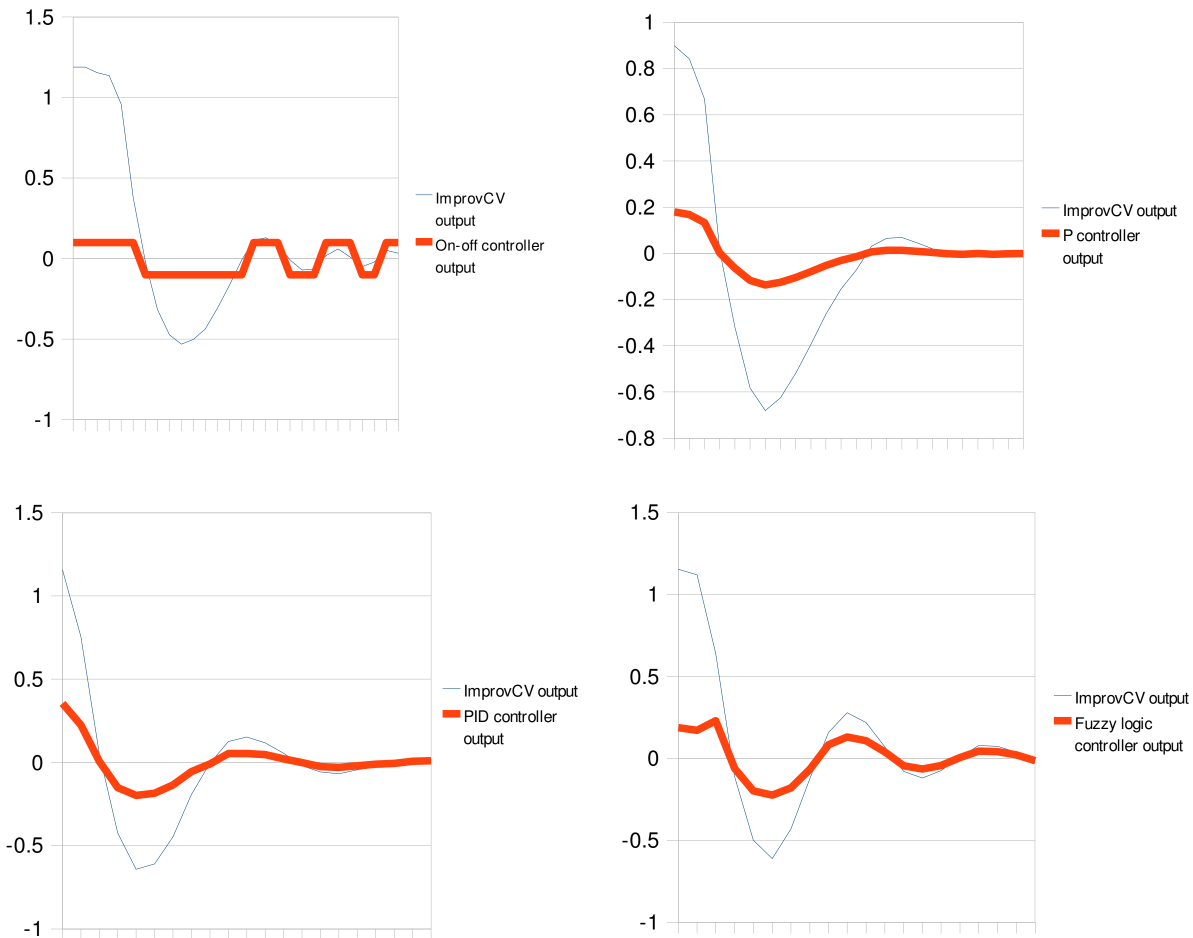


Figure 8.5 The vehicle making a too sharp turn and looses track of lane marking

The lane detection algorithm is already tested on real video recorded from a real car in a real traffic scenario. These tests show that it works also under non-ideal conditions [31].

For the simulations and experiments conducted the images used for the image processing were perfect in terms of noise, quality of lane markings and weather. In a real scenario this is not the case. Future experiments with different weather conditions and with added noise to the image could help with verifying the robustness of the lane detection algorithm. Hence implementing these features (change in weather and noise) in the simulator would be useful.



Graph 8.6: Comparison of the controller outputs (top left: on-off, top right: P, bottom left: PID, bottom right: Fuzzy logic)

From Graph 8.6 it can be seen that the controllers responded (thick line) in a gentle way to the output from ImprovCV. This way, the vehicle will not make a too sharp turn and avoid loosing track of one of the outer lane markings. The controller output in Graph 8.6 is taken from scenario 1.

9 Conclusion and Future Work

My involvement in this project has been two-sided. Firstly, I was assisting in the initial development of the automotive simulation system AutoSim. Although AutoSim is still in its infancy and will be gradually improved in the years to come the driving simulator is already able to simulate driving behaviour and test driver assistance systems. This relates to the second part of my project where an image-based driver assistance system was tested in the simulator. The driver assistance system was based on lane detection using the image processing framework ImprovCV [31] already developed at the University of Western Australia. Nevertheless, modifications and improvements to ImprovCV had to be made in order to achieve the desired behaviour when using the lane detection to autonomously control a simulated vehicle in the simulator.

Creating the content, both in terms of physics and graphics models, for AutoSim proved to be two major tasks. The physics in a driving simulator are essential in order to achieve a close to real world behaviour of vehicles and objects making up the scenario. An already developed physics engine, Bullet, was used for this together with the Physics Abstraction Layer (PAL) [44]. PAL allows the simulator to replace its physics engine to any of the engines that are supported by PAL [56] with minimal effort. This was demonstrated during my experiments when the Bullet physics engine was replaced with Newton Game Dynamics. In terms of the physics for the simulator there are still issues that need to be solved.

At the current state in the development of AutoSim the complexity and demand in processing power keeps increasing as new features are added. To be able to run the simulator on a fairly normal workstation, optimisations in all aspects of the simulator must be made. Complex graphics models versus simpler models proved to make a huge

difference in terms of performance. Hence, simpler models, with fewer polygons, are preferred and the realistic look of the models should be provided using other methods, e.g. texturing and normal maps.

As mentioned the final part of my project consisted of demonstrating autonomous driving of a vehicle in AutoSim based on lane detection from ImprovCV. Four different controllers were implemented in order to find the most robust one. The four controllers were; a simple on-off controller, P controller, PID controller and a Fuzzy logic controller. Furthermore, the four controllers were tested in two different scenarios and with two different physics engines. Physics engines are known to behave different depending on their implementation and simplifications done and this was further showed in the experiments where different behaviour was experienced when the physics engine was replaced. The two scenarios created for the experiments were; (1) straight road where the vehicle started on the side of the road slightly angled and should straighten up, (2) a double-S shaped road where the vehicle had to make several turns and corrections in order to complete the scenario. Tuning the different controllers to behave optimal in both the scenarios proved to be challenging. Optimal tuning for one scenario was not optimal for the other one. Some of the reasons for this are limitations discussed later. Nevertheless, my implementation of the Fuzzy logic controller managed to complete the two scenarios using different physics engines in a satisfactory way. The other controllers failed one or more of the challenges.

Running AutoSim and ImprovCV one the same computer whether or not you run the AutoSim server on a separate computer requires a powerful computer. If the simulated vehicle was set to drive too fast, the image processing in ImprovCV would fall behind and the vehicle would eventually drive off the road due to inexact information from ImprovCV. My solution during the experiments were to drive the vehicle fairly slow, this allowed the two systems to run in a satisfactory way. Utilising the Graphics Processing Unit (GPU) for the calculations related to the lane detection in ImprovCV, e.g. using CUDA [64], could speed up the systems. However the graphics in AutoSim are probably already using much of the available GPU resources. Faster computers can always be purchased and used, however the ideal would be to implement the image processing system on an embedded system for deployment in a real car.

The paths driven using the different controllers showed that none of the controllers drove

absolutely in the centre of the lane and behaved perfect, especially not for both scenarios and physics engines. Hence, further tuning and improvements of the controllers can be made. Furthermore, turning the vehicle too much inside the correct lane will make the camera loose sight of one of the outer lane markings and the vehicle would drive off the road. Further tuning of the filter parameters inside ImprovCV and changing the virtual camera view on the vehicle might solve this issue to some extent. Also the current lane detection algorithm in ImprovCV only detects straight lines and a too curvy road would not work with the current system. Due to the performance issue mentioned above ImprovCV sometimes loose track of the lane markings when the vehicle is either driving too fast or making to sudden turns. This feeds incorrect data back to the simulator and the vehicle might end up making wrong steering corrections. Implementing a filter to filter away the incorrect data would probably make the overall system more robust.

An initial demonstration of autonomous driving is made and there are several future scenarios and driver assistance systems that can be demonstrated and tested using AutoSim and ImprovCV. Adaptive cruise control, traffic light and sign detection and vehicle detection are just some examples. And as traffic lights, traffic signs and Position Sensor Devices (PSD) already are available and implemented these concepts can be demonstrated in the close future. In terms of the simulator, changes in the weather conditions (like sunshine and fog) and day/night would be useful to implement as driver assistance systems need to work in all conditions to be useful in real vehicles.

Abbreviations

ABS	Anti-lock Braking System
ACC	Adaptive Cruise Control
AI	Artificial Intelligence
API	Application Programming Interface
BLIS	Blind Spot Information System
CIIPS	Centre for Intelligent Information Processing Systems
COLLADA	COLLAborative Design Activity
CUDA	Compute Unified Device Architecture
DARPA	Defense Advanced Research Projects Agency
DLL	Dynamic Linked Library
ELROB	European Land-Robot Trial
ESP	Electronic Stability Program
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HIDS	Honda Intelligent Driver Support
HLA	High Level Architecture
NASA	National Aeronautics and Space Administration
ODE	Open Dynamics Engine
PAL	Physics Abstraction Layer
PID	Proportional Integral Derivative
PSD	Position Sensor Device
RARS	Robot Auto Racing Simulator
RFID	Radio Frequency Identification
RPM	Revolutions Per Minute
TORCS	The Open Racing Championship Simulator
UWA	University of Western Australia
XML	Extensible Markup Language

References

[1] D.J. Verburg, A. Knaap, J. Ploeg, "VEHIL: Developing and Testing Intelligent Vehicles," *Intelligent Vehicle Symp.*, IEEE, June 2002, pp. 537-544

[2] Margie Peden et al., World Report on Road Traffic Injury Prevention – Summary, World Health Organization, 2005

[3] P. Backlund et al., "Games and Traffic Safety – an Experimental Study in a Game-Based Simulation Environment," *11th Int'l Conf. Information Visualization* (IV'07), IEEE, July 2007

[4] D. Yang et al., "Development of Anti-Lock Brake System in Virtual Environment," *Int'l Conf. Virtual Environment, Human-Computer Interfaces and Measurement Systems* (VECIMS 2004), IEEE, July 2004, pp. 131-135

[5] M. Short, M.J. Pont, "Hardware in the Loop Simulation of Embedded Automotive Control Systems," *Proc. 8th Int'l Conf. Intelligent Transportation Systems*, IEEE, Sept. 2005, pp. 226-231

[6] S. Kubota, Y. Okamoto, H. Oda, "Safety Driving Support System Using RFID for Prevention Pedestrian Involved Accidents," *Proc. Int'l Conf. ITS Telecommunications*, June 2006, pp. 226-229

[7] Ü. Özgüner, C. Stiller, K. Redmill, "Systems for Safety and Autonomous Behaviour in Cars: The DARPA Grand Challenge Experience," *Proc. IEEE*, vol. 95, no. 2, Feb. 2007, pp. 397-412

[8] J. Steven et al., "Vision Based Vehicle Localization for Autonomous Navigation," *Proc. Int'l Symp. Computational Intelligence Robotics Automation*, IEEE, June 2007, pp. 528-533

[9] M. Maurer, E.D. Dickmanns, "A System Architecture for Autonomous Visual Road Vehicle Guidance," *Conf. Intelligent Transportation System*, IEEE, Nov. 1997, pp. 578-583

[10] The University of Iowa, “The National Advanced Driving Simulator,” Information Brochure

[11] K-Y. Tu, T-C. Wu, T-T. Lee, “A Study of Stewart Platform Specifications for Motion Cueing Systems,” *Int'l Conf. Systems, Man and Cybernetics*, IEEE, 2004, pp. 3950-3955

[12] Technische Universität München, “The FTM driving Simulator” http://www.fahrzeugtechnik-muenchen.de/content/view/11/64/lang,en/ [accessed May 2008]

[13] National Advanced Driving Simulator (NADS), http://www.nads-sc.uiowa.edu/ [accessed May 2008]

[14] U. Franke, et al., “The Daimler-Benz Steering Assistant – a Spin-off from Autonomous Driving,”, *Proc. Intelligent Vehicles Symp*., 1994, pp. 120-124

[15] DARPA, “Participants Conference Presentation,” May 2006

[16] S. Thrun et al., “Stanley: The Robot that Won the DARPA Grand Challenge,” *Journal of Field Robotics*, 2006, pp. 661-692

[17] S. Thrun, “Winning the DARPA Grand Challenge: A Robot Race through the Mojave Desert,” *21st IEEE Int'l. Conf. Automated Software Engineering* (ASE'06), 2006

[18] D. Zeng, “Self-Driving Cars and the Urban Challenge,” *IEEE Intelligent Systems*, 2008, pp. 66-68

[19] E.D. Dickmanns et al., “The Seeing Passenger Car 'VaMoR-P',” *Proc. Intelligent Vehicles*, 1994, pp. 68-73

[20] Mars Opportunity Rover, http://marsrovers.jpl.nasa.gov/gallery/artwork/hires/rover3.jpg [accessed May 2008]

[21] E. Tunstel, “Validation of Autonomous Rover Functionality for Planetary Environments,” *Proc. World Automation Congress*, 2005, pp. 447-452

[22] SubSim Submarine Simulator, http://robotics.ee.uwa.edu.au/auv/subsim.html [accessed May 2008]

[23] T. Bielohlawek, “SubSim - An Autonomous Underwater Vehicle Simulation System ,” thesis, School Electrical, Electronic and Computer Eng., Univ. of Western Australia, 2006

[24] Delta3D Game/Simulation Framework, http://www.delta3d.org/ [accessed May 2008]

[25] TORCS Racing Simulator, http://torcs.sourceforge.net/ [accessed May 2008]

[26] W.D. Jones, “Keeping Cars from Crashing,” *IEEE Spectrum,* vol. 38, no. 9, Sept. 2001, pp. 40-45

[27] S. Ishida, J.E. Gayko, “Development, evaluation and introduction of a lane keeping assistance system,” *IEEE Intelligent Vehicles Symp*., 2004, pp. 943-944

[28] Road-Ready Night Vision at Last, http://www.wired.com/science/discoveries/news/2006/02/70182 [accessed May 2008]

[29] L.-P. Becker et al., *Advanced Microsystems for Automotive Applications,* Springer, 2005, pp. 71-84

[30] S. Kubota, Y. Okamoto, H. Oda, “Safety Driving Support System Using RFID for Prevention of Pedestrian-involved Accidents,” *Proc. 6th Int'l Conf. ITS Telecommunications*, June 2006, pp. 226-229

[31] S.A. Hawe, “A Component-Based Image Processing Framework for Automotive Vision Applications,” diplomarbeit, School Electrical, Electronic and Computer Eng., Univ. of Western Australia, 2008

[32] T. Sommer, “Physics for a 3D Driving Simulator,” thesis, School Electrical, Electronic and Computer Eng., Univ. of Western Australia, 2008

[33] Irrlicht Graphics Engine, http://irrlicht.sourceforge.net/ [accessed May 2008]

[34] A. Ettlin, P. Buchler, H. Bleuler, “A Simulation Environment for Robot Motion Planning,” *Proc. 5th Int'l Workshop Robot Motion and Control*, June 2005, pp. 277-282

[35] OpenStreetMap, http://www.openstreetmap.org/ [accessed May 2008]

[36] Java OpenStreetMap (JOSM), http://wiki.openstreetmap.org/index.php/JOSM [accessed May 2008]

[37] TinyXML, http://sourceforge.net/projects/tinyxml/ [accessed May 2008]

[38] Blender 3D modelling program, http://www.blender.org/ [accessed May 2008]

[39] J.G. Brand, "Graphics for a 3D Driving Simulator," thesis, School Electrical, Electronic and Computer Eng., Univ. of Western Australia, 2008

[40] Phong Shading, http://en.wikipedia.org/wiki/Phong_shading [accessed May 2008]

[41] Bullet Physics Library, http://www.bulletphysics.com [accessed May 2008]

[42] Newton Game Dynamics, http://www.newtondynamics.com/ [accessed May 2008]

[43] Physics Abstraction Layer (PAL), http://www.adrianboeing.com/pal/ [accessed May 2008]

[44] A. Boeing, T. Bräunl, "Evaluation of Real-Time Physics Simulation Systems," *Proc. 5th Int'l Conf. Comp. Graphics Interactive Techniques Australia and Southeast Asia,* 2007, pp. 281–288

[45] B. Karlsson, et al., *CarSim – A Suspension System Model,* TNM032 Modelling Project , Dept. of Science and Technology , Linköping University , 2005

[46] T. Zuvich, "Vehicle Dynamics for Racing Games," http://www.gamasutra.com/features/gdcarchive/2000/zuvich.doc [accessed May 2008]

[47] M Monster, "Car Physics for Games," http://immi.inesc-id.pt/~brar/avt2006/projecto/Car%20Physics.htm, [accessed May 2008], 2003

[48] C. Callin et al., *ERGO – fysikk 3FY,* Aschehoug, 1998 [Norwegian physics book]

[49] C. Hecker, "Rigid Body Dynamics," *Game Developer Magazine,* Oct. 1996 – June 1997 [series of four articles]

[50] Yaw, pitch and roll described, http://en.wikipedia.org/wiki/Flight_dynamics [accessed May 2008]

[51] Racer Simulator, “Pacejka's Magic Formula,” http://www.racer.nl/reference/pacejka.htm [accessed May 2008], 2008

[52] M. Abdulrahim, “On the Dynamics of Automobile Drifting,” Univ. of Florida, Society of Automotive Engineers, Inc., 2006

[53] B. Beckman, “The Physics of Racing,” http://phors.locost7.info/contents.htm [accessed May 2008], 2008

[54] E. Coumans, K. Victor, “COLLADA Physics,” *Proc. 12th Int'l. Conf. 3D Web Technology* (Web3D '07), April 2007, pp. 101-104

[55] Scythe Physics Editor, http://www.physicseditor.com/ [accessed May 2008]

[56] Physics Engines supported by PAL, http://www.adrianboeing.com/pal/ [accessed May 2008]

[57] A. Watanabe, M. Nishida, “Lane Detection for a Steering Assistance System,” *Proc. Intelligent Vehicle Symp.,* 2005, pp. 159-164

[58] C. D'Cruz, J.J. Zou, “Lane Detection for Driver Assistance and Intelligent Vehicle Applications,*” Int'l. Symp. Communication and Information Technologies* (ISCIT 2007), 2007*,* pp. 1291-1296

[59] C-C. Wang, et al., “Driver Assistance System for Lane Detection and Vehicle Recognition with Night Vision,” *Proc. IEEE Intelligent Robots and Systems Conf.,* 2006, pp. 3530-3535

[60] T. Bräunl, *Embedded Robotics: Mobile Robot Design and Applications with Embedded Systems,* Springer, 2006

[61] G.J. Silva, A. Datta, S.P. Bhattacharyya, “On the stability and controller robustness of some popular PID tuning rules,” *IEEE Transactions Automatic Control*, 2003, vol. 48, no. 9, pp. 1638-1641

[62] L.A. Zadeh, “Fuzzy Logic, Neural Networks and Soft Computing,” *Communication of the ACM,* vol. 37, no. 3, March 1994

[63] T. Bräunl, “Fuzzy Logic,” lecture notes Fault Tolerant Computer Systems, Univ. of Western Australia, 2003

[64] NVIDIA, “NVIDIA CUDA Programming Guide Version 1.1,” 2007, http://www.nvidia.com/object/cuda_develop.html [accessed May 2008]

A Appendix

A.1 Autonomous Driving

This short tutorial will demonstrate how to set up both AutoSim and ImprovCV in order to drive a vehicle in AutoSim autonomously based on the lane detection in ImprovCV.

1. Start AutoSim client and server. If you want to use the lane detection in ImprovCV and make a vehicle in Autosim drive according to the detected line markings, specify the user program *FollowRoad.dll* in the world file to its appropriate robot.

2. Choose the user program called *WriteToSharedMemory.dll* from the AutoSim client. This user program writes images to shared memory in order for ImprovCV to read the images from AutoSim.

3. Run both the AutoSim server and client.

4. Start ImprovCV.

5. Using the shared memory images from AutoSim as input into ImprovCV can be done using two different methods.

   (1) From the file menu in ImprovCV choose *Load From Shared Memory.*

   (2) Open the XML file containing the filters you want to use, e.g. *lane2SM.xml* from the file menu's *Load.* Then click on source and a *Choose connection* box will appear on the top of the window. From the box choose *Connect to Shared Memory.*

A.2 Preparing a Car Model for the Simulator

From you have a nice looking model to you can use it in the simulation systems several steps must be carried out. Scaling, detaching the wheels and rotations are some of the steps. Below, a more thorough explanation is given. The steps below are all done in the Blender 3D modelling program.

1. The model is imported into Blender according to the model's file format. Furthermore, the model needs to be scaled to the correct size. One unit in Blender is one metre in the simulation system.

2. To give the car the correct orientation the car must be rotated. Because of the differences in the coordinate systems and differences in how the export plugins interpret the coordinate system you should at this stage choose what format you want the model exported in. If you are going to export it as a .3ds model the car should be rotated so it is pointing in the positive Y-direction (according to Blender's coordinate system). However, if the model is exported as .obj, rotate the car so it points in the negative Y-direction. This is because the obj-exporter interprets the coordinate system differently.

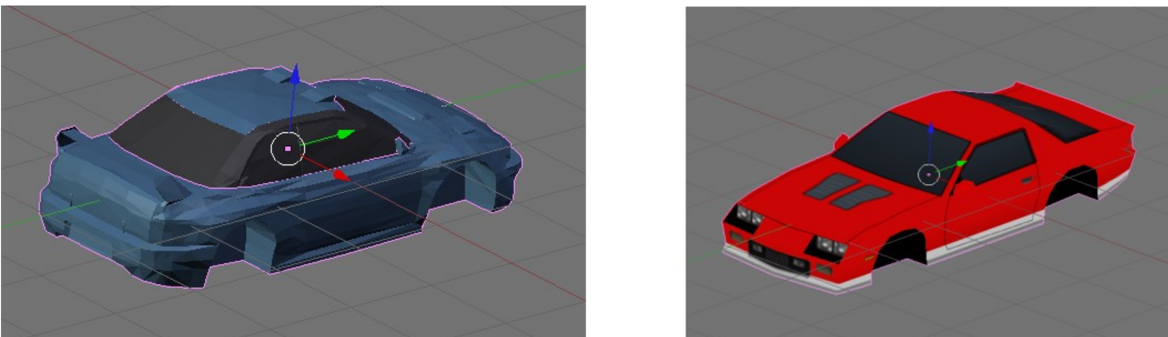


Figure A.1: Left picture: Correct rotation for .3ds model. Right picture: Correct rotation for .obj model

3. In order to get the position of the car correct in the simulator the car must be moved so that the coordinate system's origin is centred exactly in the centre of the car. It is important to get this as accurate as possible as this will simplify many of the later steps, e.g. placing lights and wheels.

4. The body of the car is selected and exported in the desired format. Name the model

as *model* with the appropriate extension. The folder structure is explained in Figure A.2.

5. Delete the body of the car from the scene together with three of the wheels, leaving only one wheel left in the scene. Move this wheel to the origin like with the car. Export the wheel. The name for the wheel should also be *model*, but saved in a different folder (Figure A.2).

6. Rotate the wheel to create the wheel for the other side of the car. Make sure it is centred in the origin before you export this wheel as well.

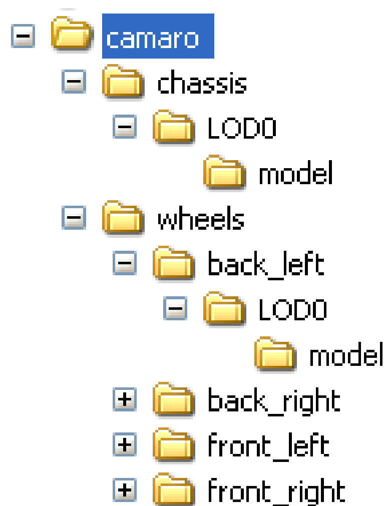


Figure A.2: Folder structure for a car

All the parts of the car are now ready for the simulation system, but the wheels need to be attached to the body of the car again inside the simulator. Also front, brake and turning lights must be attached to the car. This information is stored in the robot file, for example *camaro.xml*. As of May 2008 there is no easy and quick way of doing this and the only solution is to measure X, Y and Z distances in Blender and manually type the numbers into the robot file. Moreover, the numbers must probably be finely adjusted from the initial position because the car was not exactly centred on the coordinate system's origin.

Below are some of the important sections of a car robot file shown.

```
<?xml version = "1.0"?>
<Robot name = "impreza">
  <Parts>
    <Box name="chassis">
      <Position x="0.0" y="0.0" z="0.0" />
      <Size x="1.994999" y="1.265000" z="4.760004" />
          <Mass value="1400.0" />
      <CenterOfMass x="0.000000" y="0.376204" z="-0.764765" />
      <Model path="models/impreza/chassis" type="3ds" />
    </Box>
  </Parts>
  <Devices>
    <WheelDevice name="front_right">
      <Part name="chassis" />
      <DriveActuator name="drive_chassis" />
      <Position x="0.80" y="-0.45" z="1.35" />
      <Radius value="0.36"/>
      <Width value="0.245"/>
      <SuspensionRestLength value="0.1"/>
      <SuspensionKs value="200"/>
      <SuspensionKd value="23"/>
      <Powered value="true"/>
      <Steering value="true"/>
      <Brakes value="false"/>
      <Model path="models/impreza/wheels/front_right" type="3ds"/>
    </WheelDevice>

    <VelocimeterSensor name="velo0" >
      <Part name="chassis" />
      <!-- Axis along which the speed is measured -->
      <Axis x="0.0" y="0.0" z="1.0" />
    </VelocimeterSensor>
     <LightDevice name="brake_light_right1">
      <Part name="chassis" />
      <Texture file="media/particles/brakelights/red.bmp" />
      <Intensity value="1.0" />
      <Position x="0.72" y="0.0" z="-2.35" />
      <Size width="0.5" height="0.5" />
    </LightDevice>
</Robot>
```

A.3 Creating a World

This section will briefly describe the steps needed to create a new world. The first thing to consider is whether you want to create roads and streets based on actual streets from OpenStreetMap or if you want to create you own streets. Remember that the larger area you use and the more streets in the area the slower the simulation system will run.

1. There are two preferred ways of getting the streets from OpenStreetMap. The first options is to type in the below URL into a web browser and use the desired latitude and longitude values.

```
http://www.openstreetmap.org/api/0.5/map?bbox=-0.5,51.3,-0.4,51.4
```

   Where the bounding box (bbox) is defined this way; west edge, south edge, east edge, north edge. Depending on the size of your chosen area you will finally be asked to save a file, which is your map file in XML format. Coordinates south of the equator and west of Greenwich are negative.

   The other option is to use a feature in the Java OpenStreetMap program (JOSM) for downloading the map.

   Regardless of which method used you get an XML file containing the streets which can be opened in e.g. JOSM.

2. When the map file is downloaded it is a good idea to clean up the map by removing unwanted streets, this will speed up the simulation system. The clean-up process can easily be done with JOSM. Also, you can define different areas such as forest and residential areas. This will place trees and residential buildings in those areas respectively.

3. The next step is to place buildings next to the street. This is done using the OsmManipulator program. The OsmManipulator simply places buildings and objects next to the streets according to what is specified in the map file.

4. As the OsmManipulator only helps you place a lot of objects, the map needs to be

edited after the original map is manipulated. One problem is that some houses are placed in the middle of intersections. These need to be removed.

Another tutorial on world creation and further details can be found in: *J.G. Brand, "Graphics for a 3D Driving Simulator," thesis, School Electrical, Electronic and Computer Eng., Univ. of Western Australia, 2008*