Project Documentation

heins4: Stefan Heinemann
ischc2: Christoph Isch

An Event Driven Multiagent Traffic Simulation

Bern University of Applied Sciences
Engineering and Information Technology
Computer Perception & Virtual Reality

Module: Projektarbeit 2 (7302r)
Professor: Prof. Dr. Jürgen Eckerle

Stefan Heinemann (heins4)
Christoph Isch (ischc2)

1 Abstract

This document describes our work in the module *Projektarbeit 2 (7308)* at the Berne university of applied science. We chose to continue our traffic simulation which we started in the module *Projektarbeit 1*. The goal was to develop an event driven multiagent simulation which allows virtual cars to drive around independently and to react to the traffic situations autonomously. Autonomy and idependence were only possible when the vehicles or more precisely the vehicles drivers would perceive the environment on their own. We therefore built a world which consists of objects, that are easy to perceive and represent different real world aspects: the waypoints. Thus the world now consists of the following items:

- Road, a road has one or many lanes in one or two directions
- Junction, a junction connects roads
- Vehicle, a vehicle is operated by a driver which has a consciousness, the Animus

All of these are represented by waypoints which are processable by the Animus. As one of our main goals was to keep everything as generic as possible to open the simulation to a broad range of usages we decided to make cuts in the graphical representation in favour of the more complex world design.

Contents

2 Introduction

The purpose of this project was to develop a driver model to enable cars to drive around by themselves. We developed a simulation where we could implement and test this model. To provide a stable basis for the driver model we also had to solve problems concerning the road model and the environment in general. We also wanted some graphical display of the simulation. We did not focus hard on that though, it should only be rudimentary. It was important to us to implement the things as generically as possible, to have a code base that is open for even more features than are now possible to implement in the given time. Some of these problems with the simulation had already been solved in the previous project module.

Practical applications for such a driver model could be algorithms for cars that are able to roam independently or simulations which may predict traffic jams.

3 The waypoint system

3.1 Concept

To enable the drivers to see things in the world, we developed a system of objects that we call the waypoints. When a driver looks at it's vicinity, all things in this area are collected and passed to the driver as waypoints. With this concept, all objects in the world can be treated similarly.

The area that describes what the driver can see is implemented as the "driver view" (3.3.2) and is based on the manner how humans view their environment.

To have an efficient location based filter for the waypoints we implemented a quad tree (3.3.1).

3.2 Types of waypoints

We thought about the things that a driver would see when driving through a street that could have an effect on his behaviour. Such things could be

- Other vehicles (cars, bicycles, trucks,...)
- Pedestrians
- Road signs
- Junctions
- Pedestrian crossings
- ...

For this project, we decided to implement only the waypoints for cars, junctions and speed signs, but the design should still be open for other waypoints in a later stage of the project. So we came up with following class design:

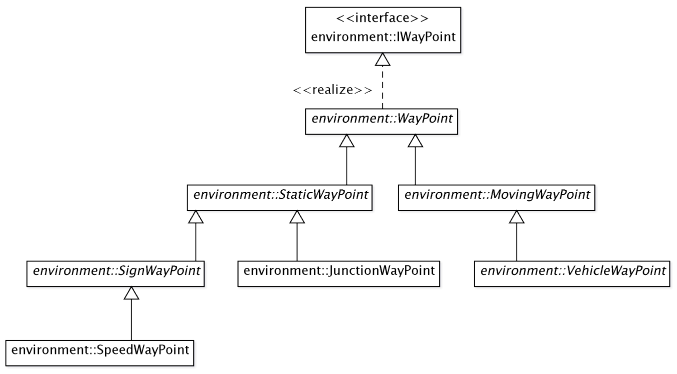

<<interface>>
environment::IWayPoint
<<realize>>
environment::WayPoint
environment::StaticWayPoint
environment::MovingWayPoint
environment::SignWayPoint
environment::JunctionWayPoint
environment::VehicleWayPoint
environment::SpeedWayPoint

Figure 1: The class design of the waypoints

The implemented waypoints are described further below.

3.2.1 Waypoint distance

A property that all waypoints have is a method to return the distance to another vehicle. To make it more realistic, some random error is added to this distance, since humans can only estimate distances.

3.2.2 Static waypoints

The static waypoints are those that will never be moved in the world. They are given by the XML file that describes the world (7.2).

Junction waypoints A junction waypoint describes a junction. It provides the information, in which directions the driver can go. Junction waypoints are generated by the junctions and placed on the lanes (7.1) at the position of 80% of the lane length.

Speed waypoints The speed waypoints are road signs that restrict the speed limit to a certain value. To keep things simple, there is, unlike in the real world, no sign to abrogate the speed limit to the default value.

3.2.3 Moving waypoints

Some of the waypoints represent moving objects. Since we only have cars in this project so far, only those waypoints are needed.

Car waypoints These waypoints represent other cars in the simulation. Each car generates its waypoint and updates its position as the car moves on.

3.3 Waypoint finding

The search for waypoints that a driver can see when driving on ordinary lanes is achieved in three steps:

- Filter with a quad tree
- Filter with the driver view
- Check the lanes

The waypoint finding around junctions is slightly different, see section 3.4.

3.3.1 The quad tree

To be able to locate the waypoints in a certain area quickly, they are organized in a quad tree. The quad tree basically divides the world into rectangles as shown in figure 2 to a certain depth, which are nodes in the tree. The depth defines the resolution of the tree, and depending on the depth the nodes cover each an area in the world. The nodes are created only if there are waypoints in their area. They can contain more than one waypoint, since there can be more than one waypoint in an area.

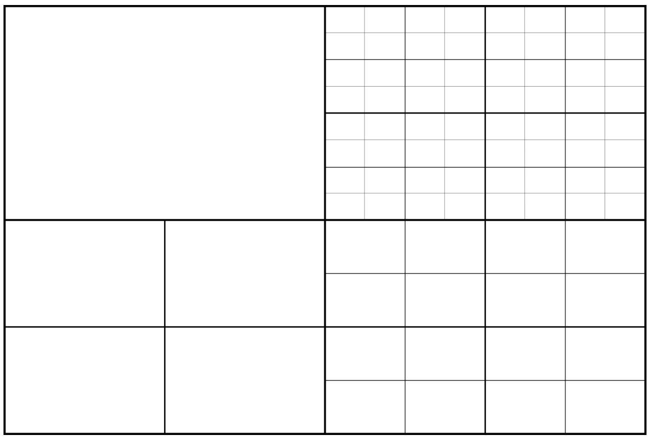


Figure 2: The fragmentation of the quad tree with different depths up to depth 4 in the top right corner

With this implementation, all the nodes in the vicinity of the vehicle can be found with O(d) where d is the depth of the three. Since the quad tree is based on rectangles, the results have to be filtered further to determine if the located waypoints really lie in the driver view, which is explained in the next chapter.

The rectangles to use for the search are calculated from the driver view. Rectangles that enclose the whole driver view are taken for the search.
Figure 3 shows the composition of the quad tree classes.

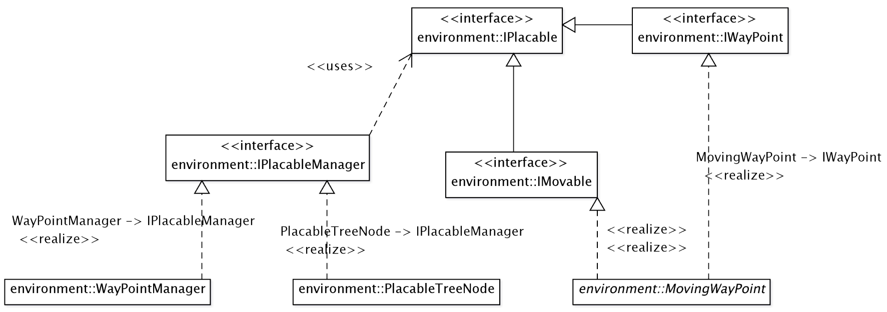

<<interface>>
environment::IPlacable
<<interface>>
environment::IWayPoint
<<uses>>
<<interface>>
environment::IPlacableManager
<<interface>>
environment::IMovable
MovingWayPoint -> IWayPoint
<<realize>>
WayPointManager -> IPlacableManager
<<realize>>
PlacableTreeNode -> IPlacableManager
<<realize>>
<<realize>>
<<realize>>
environment::WayPointManager
environment::PlacableTreeNode
environment::MovingWayPoint

Figure 3: The quad tree classes

3.3.2 The driver view

Description The driver view describes the area wherein the driver can see other objects in the world. It is described by following parameters:

- **Direction** The direction the driver looks
- **Position** The position of the driver view
- **Angle** The angle of the visual field
- **Distance** The distance that the driver can see

Drugs can be applied to these parameters. They can have a certain effect on each of the parameters.

With this system of parameters we tried to make the driver view as realistic as possible. At first we had a driver view shaped as a triangle, which was based on the human field of view. Figure 4 depicts this driver view.

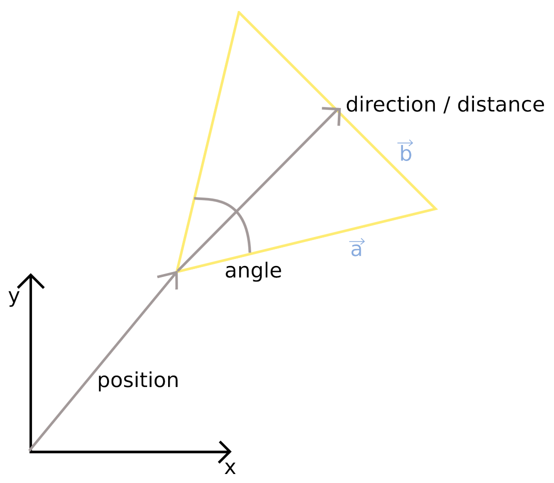


Figure 4: The first driver view

During the development we realized that, although this driver view worked well for straight lanes, it posed a few problems with situations like curves and junctions. In these situation the driver should have the possibility to look around. Instead of enabling him to do so we implemented a new version of the driver view that enables him to see more in the world. It is shaped like this:

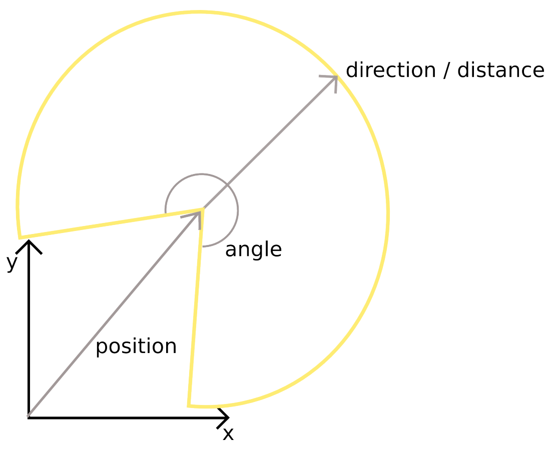


Figure 5: The new driver view

With this shape the driver is able to look around the curves and also see things on a crossing lane. Thanks to thoughtfull software design for the driver view we could just replace the triangular driver view with the new, circular one without having to adopt other parts of the code.

The filtering After the quad tree (3.3.1) has completed its search, the located waypoints are filtered further to take account of the angled driver view instead of using just a rectangle.

The algorithm to determine whether a waypoint lies in the triangular driver view, is based on a linear equation which we implemented globally in the vector class (9.2). If following constraints are fulfilled, the waypoint lies in the driver view triangle:

$$\lambda, \mu \in]0, 1]$$

$$\frac{\lambda * |\vec{b}|}{\mu * |\vec{a}|} \leq \frac{|\vec{b}|}{|\vec{a}|}$$

Where λ and μ are the scalar components of the linear combination

$$\vec{v} = \lambda * \vec{a} + \mu * \vec{b}$$

The algorithm for the Pacman-shaped waypoint is much simpler. Assuming $\vec{v}$ is the position vector of the waypoint, $\vec{p}$ the one of the driver view and α is the angle of $|\vec{v} - \vec{p}|$ to the x-axis, following constraints have to be fulfilled (here the distance is the radius of the circle):

$$|\vec{v} - \vec{p}| \leq \text{distance}$$

$$-\frac{\text{angle}}{2} < \alpha < \frac{\text{angle}}{2}$$

3.3.3 Lane check

In the last step of waypoint finding, the waypoints are checked with the lane queue of the vehicle (6.5). If the waypoint is not on any of the lanes in the vehicle's queue, it is dropped, since it doesn't concern the driver. An example of such a waypoint is a vehicle on the lane that goes in the opposite direction of the road.

3.4 Junction waypoint finding

The finding of relevant waypoints while driving onto or on a junction is somewhat more complex. First of all other vehicles have to be considered not only the ones on the same lane but also those on the lanes that are crossing the own track. And then there's another important point to consider: the priorities so when to give way or not to. Therefore this process has been divided into three substeps:

3.4.1 Getting the relevant lanes

After the waypoints have been filtered by the DriverView we apply another filter based on the lanes leading through the junction ahead. The junction returns a list of lanes that are not related to the lane we are on. If a vehicle waypoint doesn't contain one of the lanes in the list it is ignored.

3.4.2 Detecting track intersections

When a driver makes a decision which way it will turn on a junction it saves a decision object in the animus which contains a Direction object. There are three different directions: *Left, Straight* and *Right*. If the track of another vehicle is not intersecting with the track of this vehicle ignore the vehicle. To determine if it intersects the directions are evaluated.

3.4.3 Process of giving way

So to be able to evaluate which car has to give way the priority Object has to be provided with the direction the own car is heading to, where the other car is coming from and where it is heading to. On this basis it returns true if we have to give way or false if we do not. This value is then stored so that driver is able to remember who has to give way. This is necessary because the directions change while the vehicles are driving through a junction. Meanwhile the distance to the vehicles which we have to give way to are evaluated and the shortest one is taken into consideration exactly as it is done when following another car. For instance there is only the PriorityRight object implemented which simulates the rules of giving way always to the vehicle coming from or turning right.

4 Simulation

We divided the simulation in 3 main parts (apart from the GUI):

- The driver model (chapter 5)
- The environment model (chapter 7)
- The vehicle model (chapter 6)

Figure 6 shows how the different sections of the simulation interact with each other.

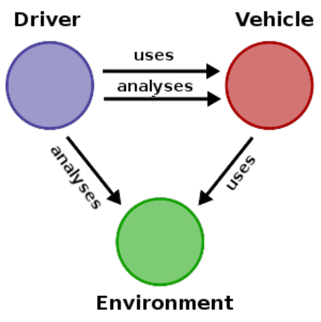


Figure 6: The interaction of the models

The core of the simulation is the event system, which is described in the next chapter.

4.1 The event system

The simulation is built as an event process, which means that everything is triggered by events that occur on a certain point in time. Currently, the following events are implemented:

The driver event This event can either indicate the driver to assess his situation (5.1) or contain a decision on the junction that he approaching. The event that indicates an assessment is created every few milliseconds, depending on the parameter in the driver's physics (5.2).

The crash event This event indicates the simulation that there has been a car crash.

The vehicle event This event contains changes in the acceleration of the vehicle. With this event we simulate a delayed reaction of the driver.

All events are put into the event queue on creation. The queue sorts the events according to their timestamps. They are therefore processed in the order that they have to occur.

Figure 7 shows the composition of the concerning classes.

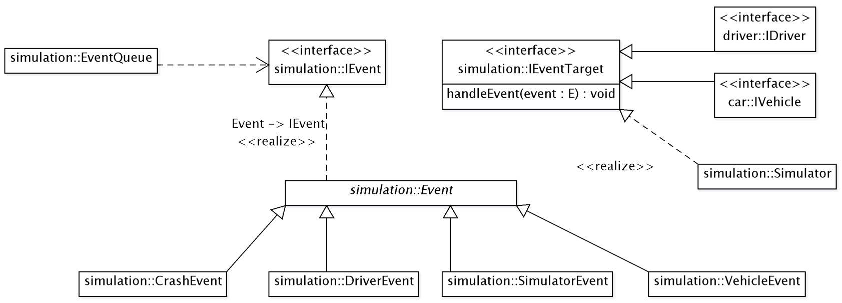
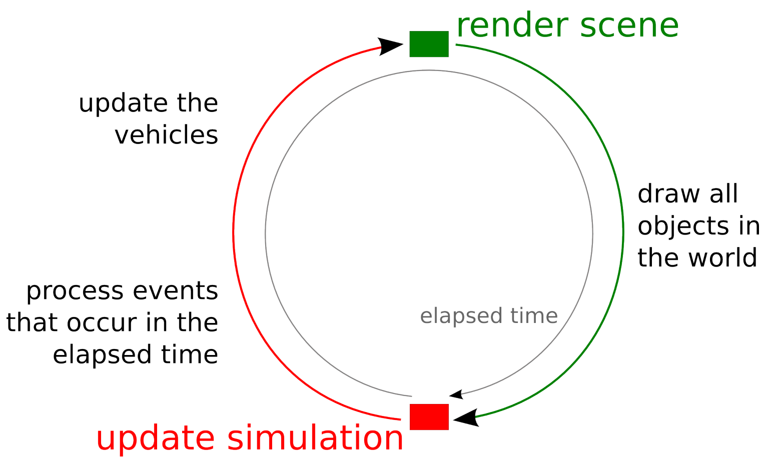


Figure 7: The composition of the event system

4.2 The process

The following figure shows how the main loop works:


render scene
update the vehicles
draw all objects in the world
process events that occur in the elapsed time
elapsed time
update simulation

The simulation is heavily connected to the GUI (8). Each time a frame has been rendered, the GUI tells the simulation to update the world whereat it passes the elapsed time. The simulation then processes all events that have to occur in the elapsed time, ordered by their time stamps. With this model no threads are needed, and the GUI renders exactly the amount of frames possible to keep the simulation running.

4.3 Collision detection

We added some collision detection to the simulation. Two vehicles that are close are checked for collisions periodically. This is done using a linear combination (9.2). If two vehicles really do crash, a crash event is created. The simulation then freezes the vehicles and removes their way points. Like that, they are still displayable in the GUI, but they do not affect the simulation anymore.

5 Driver

The part that does all the acting in this world is the drivers. A driver looks at the world (5.4), assesses the situation and reacts accordingly (5.1), decides about which way to go (5.5) and eventually steers the vehicle (6.2). Drivers have a character (5.3), physical conditions (5.2) and could even take drugs (5.2.1).

5.1 Animus

The Animus is the brain of the driver. Here all waypoints (3) are processed and decisions are made.

The Animus only knows the current speed limit on the lane (before the driver has seen any speed limit signs this is some global default value). It then assesses the situation in following order:

- Look at the world and collect way points (assessment)
- Process them and let them act on the acceleration and deceleration activators
- Do the way point specific stuff (e.g. remember speed limit)
- Calculate the resulting acceleration value

5.1.1 Assessment and decision making

Based on the way points the driver has to decide whether to accelerate or decelerate or keep the current speed. Since he can see several way points at once, we developed a system where all the way points have a certain effect on that decision. Figure 8 shows this process.

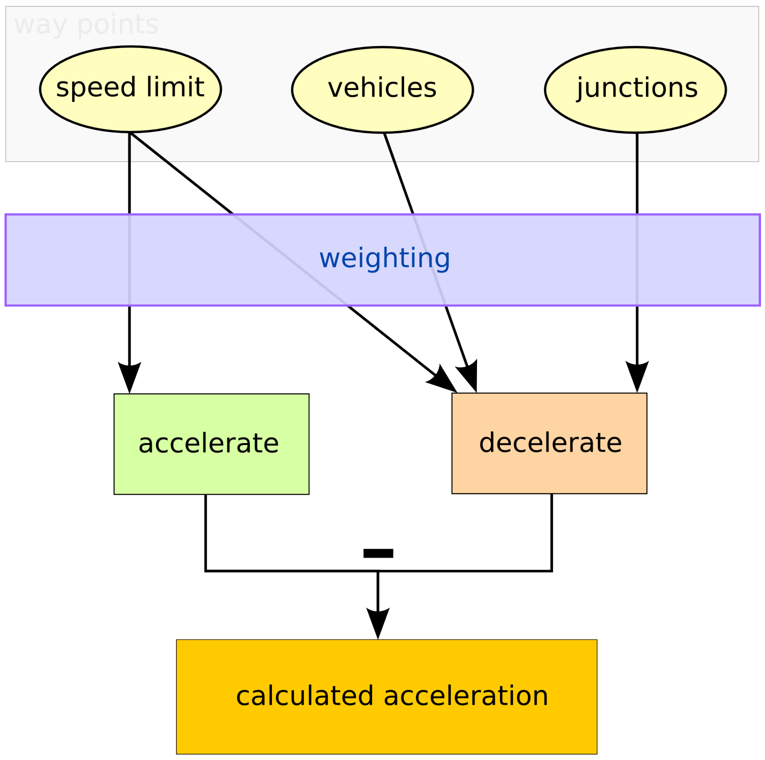


Figure 8: How the waypoints influence the acceleration

The influences on the acceleration and deceleration are weighted, so for instance the vehicle way points have a higher weight so it influences the deceleration more than a speed way point can influence the acceleration. The acceleration and deceleration in this picture are values ranging from 0 to 1. After all way points had their influence, the two values are then subtracted. The difference is then the effective acceleration value that goes to the vehicle; -1 is full braking and 1 is full acceleration which both depend on the vehicles properties.

With this system the driver always drives as fast as he is allowed to as long as there is nothing else interrupting his way.

5.1.2 The estimation of another vehicle's speed

To know whether another vehicle is approaching (because it moves more slowly) or not, we could have just implemented a method on the way point that returns the speed of it's vehicle to the driver that is assessing. But we wanted it to be more realistic, the drivers should have to estimate the speed of the other traffic participants. They can therefore only retrieve the somewhat fuzzy distance (3.2.1) to the other waypoints. A driver then has to remember the distance of the nearest vehicle, until the next assessment cycle. It can then compare the remembered distance with the new one and knows, whether the vehicle is approaching or not and how fast.

5.1.3 Following another vehicle

The driver always tries to keep a certain security distance to the vehicle in front of him, which is twice the speed that he is going (e.g. $50km/h => 100m$). When the distance of the vehicle in front falls below that value, the animus begins to influence the deceleration value (5.1.1). The nearer the vehicle is, the higher is the influence.

5.1.4 Decisions in junctions

The behaviour in junctions is a bit extended. As described in 3.4, the way point finding is a bit more intelligent to regard the right of way. Once they are found, the process is the same though.

5.2 Physics

The physics objects define the physical conditions of the driver. Every driver has one of these objects. They provide the following properties:

- **The sight** How far the driver can see
- **The field of view (angle)** The opening of the driver view
- **Update interval** The interval of the assessment cycles
- **Drugs**

5.2.1 Drugs

Drugs are objects than can influence the properties mentioned above in the physics section. They are merely just implemented and prepared but not in use yet.

5.3 Character

The drivers are prepared to have some sort of character. This character contains values like "riskyness“ and temperament. A driver with a high temperament accelerates faster than a Sunday driver.

5.4 Sight

At this point of process the driver only sees things ahead of him on the lane. There are no mirrors, and it is not possible to look behind, hence he is not able to reverse.

What the driver can see is limited to a field of view formed like a "Pacman“. We implemented that as the *driver view*, more on that in section 3.3.2.

5.5 Junctions

When a driver approaches a junction, he receives the according way point from the way point system. He then makes a decision in which direction he wants to drive on. Since we don't have any path finding and the cars are just driving around, this decision is randomly taken. Once decided, a driver event is created and processed immediately which then triggers the junction routine of the vehicle (6.5).

6 Vehicles

Since this is a traffic simulation, the traffic participators play a heavy role in this simulation. Traffic participators can be anything that uses a road, e.g. cars, trucks, bicycles, even tramways.

In this project we limited ourselves to the cars. Figure 9 shows the according classes.

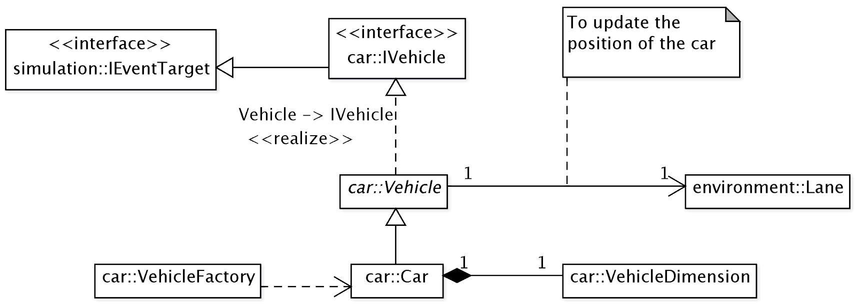


Figure 9: The composition of the vehicle objects

6.1 Cars

Cars are fairly easy objects in our world. Besides the obvious parameters like position, current speed, current acceleration, maximal acceleration and deceleration they also contain an object *VehicleDimension* that contains information about the dimensions of a vehicle. We implemented this to achieve a high level of detail concerning the estimation of distances by the driver.

In this simulation, the maximum acceleration and deceleration values are chosen by random within some range to simulate different types of cars. At this point of progress, the cars cannot reverse. We would have to adjust the driver view concept (3.3.2) so that the driver can see things that are behind him by either introduce mirrors or turn the driver view over to the back of the vehicle.

6.2 Driver's influence

The only influence that the driver currently has on the vehicle is the acceleration. It can range from -1 (full braking) to 1 (full acceleration) where 0 is to keep the current speed. These values are not applied directly, instead the vehicle receives them as an event (4.1), to simulate the natural latency between sight and reaction to the perceived situation.

6.3 Movement

To update the position of the car while it's moving around, it is given a time stamp from the simulation (4) that indicates the elapsed time. Based on the current speed and acceleration of the vehicle, the driven distance is calculated. This distance is then passed to the lane, which calculates the new absolute position in the world.

The current speed is calculated by the basic equation for acceleration:

$$v = v_0 + a * t$$

where the acceleration can also be negative to simulate braking. Similarly, the driven distance is calculated by

$$s = s_0 + v * t$$

In this process, the direction of the car and therefore the driver view are updated too, in case the car moves around a curve.

Additionally the position of the moving way point has to be updated.

6.4 Steering

To make the simulation simpler, we decided to constrain ourselves regarding the steering of the vehicles. Currently they can not move around quite freely, instead they drive on the lanes like they were on rails. This frees the driver from having to decide, in which direction she has to steer.

6.5 Lane changes

Due to the configuration of the world, lanes can only connect to junctions (7.1.2). In the process of driving through a junction, the junction offers the driver the possibilities to drive through. Once decided, the lane that connects to the next road and the lane on that road are put into a queue in the according order. When the car reaches the end of a lane, it dequeues a lane from the queue and drives on on that lane. The Problem with this was as the simulation is event driven, that actually a car can be already over the length of a lane when the update is processed. We circumvented this issue by keeping track of the driven distance an detecting such overflows.

7 Environment

7.1 The road model

The road model is defined by an XML file (7.2). It contains roads and junctions that are connected together. A road defines lanes, whereof there can be more than one even in the same direction. It also defines a path, consisting of straight elements. The simulation then creates the curves between those straight elements with bézier curves (7.1.3). For every lane there is a new path generated that goes along the given path.

7.1.1 Lanes

Lanes are composed of lane segments, which there are two types of: straight segments and curved segments. Straight segments are fairly easy and only consist of the start and end point. The curved segments have some more properties to be able to create the bézier curve. These lane segments have to alternate, so every straight lane is connected to a curved one and vise-versa.

Only the straight segments have to be specified in the XML file. The curved segments in between are generated automatically.

7.1.2 Junctions

Junctions can have different types, like a crossroads or a roundabout. For instance there's only an implementation of crossroads. The layout of the junction, however, is by default determined automatically based on the roads that it is connecting. All lanes with the same direction are connected together.

The algorithm for connecting the roads is the following:

- take two random roads and find out which ends are closer to each other
- find the closest ends of the other roads
- according to that build a matrix which contains all incoming and outgoing lanes
- connect all the incoming lanes to the outgoing lanes which belong not to the same road

When the junction is built, it places junction way points on the lanes that are connecting to it. These way points not only indicate the junctions to the drivers but also offer the possibilities of directions to take.

7.1.3 Bézier curves

To be able to create any type of curves in our simulation we implemented bézier curves for both the straight and the curved lane segments. The straight segments are done with linear curves, the curved lane segments with quadratic curves. The curved segments therefore have additionally to the start and end point a vector that indicates the bend. The reason that we implemented the straight lines as bézier curves too is to avoid having to treat them differently.

Since the curved segments are created automatically, the bend point is calculated automatically too. To achieve this, the linear segments are considered as straights and the intersection of them is then calculated. This intersection point is then used as the bend point of the bézier curve.

With this algorithm, some erroneous conditions in the XML file can be detected too, for example, lane segments that are crossing each other.

7.2 XML

As mentioned above, the road model is defined by XML files. They are designed quite simple, since they have to be written by hand. Most work is done automatically by the simulation builder. Figure 10 shows an example of a road consisting of only one lane with two straight segments and a speed way point restricting the speed limit to 30. Road segment coordinates are relative to the start point of the road.

```xml
<road id="1" startX="20" startY="0">
    <leftlanes>

    </leftlanes>
    <rightlanes>
        <lane id="1" width="1" >
            <waypoints>
                <SpeedWaypoint id="1" value="30" position="0.2" />
            </waypoints>
        </lane>
    </rightlanes>
    <roadsegments>
        <roadsegment order="1" startX="0" startY="0" endX="380" endY=
            "0" />
        <roadsegment order="2" startX="400" startY="20" endX="400"
            endY="460"/>
    </roadsegments>
</road>
```

Figure 10: An example of a road definition in the XML file

Figure 11 shows a junction that connects 3 roads of the type "CrossRoad“:

```xml
<junction type="CrossRoad" id ="15">
    <roads>
        <road id="21" />
        <road id="23" />
        <road id="24" />
    </roads>
</junction>
```

Figure 11: An example of a junction connecting 3 roads

The XML file is checked with an XSD file. Like this we can automatically avoid certain errors in the XML file without having to check these things further in the code.

8 Graphical user interface

The focus of our project was more on the simulation than on the graphical display of it hence the GUI is kept quite rudimentary.

We used the 2D game library Slick [1] to implement the GUI. Slick relies on OpenGL to display the graphics. To make things exchangeable, we tried to avoid heavy relations between the GUI and the real simulation (except for the update cycle). Therefore for every element in the simulation that has to be displayed we made a wrapper class that takes care of the drawing.

Additionally we used the Nifty Library [2] which is a library to display buttons and other GUI elements. These GUIs can easily be described in an XML file.

An important thing to note is that the GUI calls the update routines of the simulation. This is documented in more details in section 4.2.

8.1 Bézier Curves

The Slick library comes with a routine to draw first grade Bézier curves. Unfortunately this algorithm drew very edged curves which did not correspond with the curves of the simulation. Therefore we had to draw the paths with our own Bézier algorithm.

8.2 Options

We added some options to the simulation GUI:

FPS Show the frame rate per second

Grid Show a grid of 100x100 units to be able to estimate distances for debugging

DriverView Toggle the display of the driver views

[1] http://slick.cokeandcode.com/
[2] http://sourceforge.net/projects/nifty-gui/

9 Miscellaneous

9.1 Vectors

We tried a few given vector classes, but we weren't satisfied with their features. It lies in the nature of vectors, that when an operation on a vector is performed you usually get a new vector and the base vector is left unaltered. Apparently the other people who provided implementations for vectors have not been having this impression which made us implement our own Vectors.
So we made a generic interface for vectors of any size and implemented a 2D vector with following operations:

- Addition
- Subtraction
- Norm
- Cross product
- Dot product
- Normalize
- Multiply by scalar
- Rotate
- Get angle to x axis

All operations that do not result in a scalar return a new vector object instead of changing the one that the operation is called on. We are aware that this is slower in process time than to change the objects, but most times we needed new vector objects anyway. Additionally it provides more safety, since it's not possible to accidentally change a vector object where it should be cloned.

9.2 Linear combination

Often we had to solve problems where the borders of an area had been defined by vectors and the question was if a certain point is in that area or not. This sort of problem is usually solved by a linear combination of the form:

$\vec{v} = \lambda * \vec{a} + \mu * \vec{b}$

Since the solution to this problem is a linear equation system based on the dimension of the vectors we decided to implement this as a static method in the vector implementation. We omit the math here and present just the solutions for λ and μ

$\lambda = \frac{-(b_x * v_y - b_y * v_x)}{a_x * b_y - a_y * b_x}$

$\mu = \frac{a_x * v_y - a_y * v_x}{a_x * b_y - a_y * b_x}$

10 Installation

The whole solution is made avalaible by us on linux systems, however the libraries needed to compile the java source on other systems are provided as is and without any warranty in the /lib/slick/native/ directory

10.1 Getting the source

Please download the source from our repository on github
(https://nodeload.github.com/schtibe/Projektarbeit-2-7302/zipball/master)

10.2 Preparing your environment

- unpack the data to a folder of your choice
- install the java sdk 1.6: sudo apt-get install openjdk-6-jdk
- install ant: sudo apt-get install ant
- change the locations in the build.xml file to match your system

10.3 Running the project

- build the project using ant: ant build
- run the project: ant run

List of Figures