Department of Electrical Engineering and Information Technology
Distributed Multimodal Information Processing Group
Prof. Dr. Matthias Kranz

Development of a Driving Simulator based on Satellite Images and Mapdata

Entwicklung eines Fahrsimulators auf Basis von Satellitenbildern und Kartendaten

David Zander

Diploma Thesis

Author:	David Zander
Address:	[redacted]
Matriculation Number:	[redacted]
Professor:	Prof. Dr. Matthias Kranz
Advisor:	Dipl.-Ing. Stefan Diewald
Begin:	17.04.2012
End:	17.10.2012

Department of Electrical Engineering and Information Technology
Distributed Multimodal Information Processing Group
Prof. Dr. Matthias Kranz

Declaration

I declare under penalty of perjury that I wrote this Diploma Thesis entitled

Development of a Driving Simulator based on Satellite Images and Mapdata

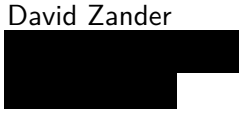
Entwicklung eines Fahrsimulators auf Basis von Satellitenbildern und Kartendaten

by myself and that I used no other than the specified sources and tools.

Munich, October 14, 2012

David Zander

David Zander



Kurzfassung

In dieser Diplomarbeit wird eine mögliche Realisierungslösung für einen Fahrsimulator im Detail dargestellt. Dieser wird benötigt um Szenarien von zukünftigen Vehicle-to-X Applikationen nachzustellen. Er wurde in C++ unter Linux programmiert und benutzt zur Darstellung der 3D-Umgebung und -Objekte OpenGL.

Das Terrain besteht aus quadratischen Kacheln, welche in einem frei definierbaren Radius um das eigene Fahrzeug herum gezeichnet werden. Die Textur für jede Kachel besteht aus einem Satellitenbild, welches aus dem Internet von diversen Quellen heruntergeladen und lokal zwischengespeichert wird. Zusätzlich zur Textur werden auch die Straßenverläufe mittels Fragmentshader auf das Terrain projeziert. Dem Fragmentshader wird dazu eine zusätzliche Textur bereitgestellt, die den Abstand zur nächsten Straße repräsentiert. Diese Entfernungen stammen aus einer SQLite Datenbank, welche zuvor mit Daten aus einer OpenStreetMap Datei befüllt wurde. Diese Daten werden ebenfalls verwendet um den Namen der nächsten Straße zu ermitteln.

Die Fahrzeug- und Objektformen werden aus einer Datei geladen und deren Verhalten mittels der *BulletPhysics* Bibliothek simuliert. Diese Physikengine ist quelloffen, simuliert Kollisionen und unterstützt Fahrzeuge standardmäßig. Zur Steuerung des Fahrzeuges kann sowohl die Tastatur, als auch ein Lenkrad mit Pedalen benutzt werden.

Die meisten Einstellungen werden aus einer *XML Datei* geladen, um möglichst einfach Änderungen vornehmen zu können.

Die Kamera kann per Tastendruck zwischen verschiedenen vordefinierten Perspektiven wechseln, sowie ihre Distanz zum Fahrzeug verändern.

Neben dem eigenen Fahrzeug können auch weitere Fahrzeuge via einer TCP/IP Schnittstelle dargestellt, oder auch über die serverseitige Physikengine gesteuert werden.

Als Information für den Fahrer werden ein Tacho mit aktueller Geschwindigkeit und gefahrenen Kilometern, sowie ein Schild mit derzeitigem Straßennamen ausgegeben.

Abstract

In this diploma thesis, an implementation option for a driving simulator is introduced. It is needed to reconstruct scenarios for future vehicle-to-x applications. It was programmed in C++ unter Linux and uses OpenGL to display the 3D-environment and -objects.

The terrain consists of quadratic tiles that are drawn around the vehicle in a definable radius. The texture for every tile consists of a satellite image that is downloaded from the Internet using different sources and cached locally afterwards. Additionally to the texture, the roads are projected onto the terrain using a fragment shader. The fragment shader is given an extra texture that represents the distance to the next street. This distances origin from a SQLite database, which was filled with OpenStreetMap data beforehand. This data is also used to identify the next street's name.

The vehicle- and objectshapes are loaded from a file and its physical behaviour simulated using the BulletPhysics library. This physics engine is open source, simulates collisions and supports vehicles out of the box. To control the vehicle, either a keyboard or a stearing wheel can be used.

Most settings are loaded from an XML file which allows modifications for fast.

The camera can switch between various predefined perspectives and change its distance to the vehicle.

Besides the own car, other cars can be displayed via a TCP/IP interface, or being controlled using the server-side physics engine.

As information for the driver, a speedometer with current speed and driven kilometers is shown, as well as a streetsign with the name of the street, the car is driving on.

Contents

Chapter 1.

Introduction

When looking at the State Of The Art (Section 2), it can be noticed that there already exist a countless number of driving simulators and software. Most are designed for first person use, connected to a real car interface and a large screen in front. All of these focus on building a virtual 3D city. A few others can also be used on a desktop computer. But looking deeper, there is no appropriate driving simulator that can be used to represent existing environments uncomplicatedly and matches the following requirements:

- Running on a Linux Operating System (OS)
- Displaying existing environments
- Open source, so the simulator can be extended

No simulator found matches these three conditions. Therefore a completely new software has to be written.

The main reason for using a simulator at the VMI group is to visualize driving scenarios from a third person perspective. For example for testing Vehicle-To-X applications [1], the simulator can serve as a visualization for the current situation [2]. So the main objective is not a realistic simulation of the car behaviour and all physical forces involved, and there is no need for high class rendered images like in up-to-date driving games.

Given an existing simulator, matching the three main requirements listed above, additional tasks have to be added, in order to be used by the VMI group:

- Network interface, allowing other simulators or applications to communicate with
- Information about the current street name
- Obstacles (e.g. a construction site), the car can collide with
- Stearing wheel support

The Network interface allows a realtime obersevation of actual cars equipped with a mobile device, sending continuously its position back to the office. Using an additional logger that saves the car's position, the route can be replayed later on trough the network interface and a playback application.

The stearing wheel offers the possibility of Operator-in-the-Loop tests in which a person drives the virtual car around the city. Combined with the obstacles (construction site), road information and network interface (sending the position and collision information), the driving behaviour of a test person can be observed by a researcher.

There was no use of an exact programming language or framework demanded. The choise fell on C++ and OpenGL. C++ is used in about 82% of todays games and Java would be too slow in performance [3]. C++ also has the benefit of a large variety of libraries, that were extensively used. These libraries are

- Standard Template Library (STL): templates for lists and queues
- Simple Directmedia Layer (SDL): joystick and images
- libCURL: receiving data via Hypertext Transfer Protocol (HTTP)
- libJPEG: reading and writing JPEG images
- BulletPhysics: physics engine
- libxml2: reading and writing Extensible Markup Language (XML) files
- sqlite: file based database-connection
- readosm: reading Open Street Map (OSM) files
- NVidia CG: library for Graphics Processing Unit (GPU) shader

Chapter 2.

State Of The Art

Other Simulators And Applications

Nearly all simulators today focus on building a virtual non-existent 3D world. They focus on a good first-person view, so that the driver who is sitting in an actual car gets the impression of being within the scene. All car functionalities and a very realistic car behaviour is integrated. The cities are designed in a 3D modelling programm or an external editor that is written for the specific simulator. A list of those simulators can be found on the french website of the institut national de recherche sur les transports et leur sécurité[1].

Driving Simulators are mostly used in universities and car manufacturers for the following researches[2]:

- User training
- Training in critical driving conditions
- Analysis of the driver behaviours
- Analysis of driver responses

Using simulators, experiments can be conducted that are illegal or unethical to do under real conditions [4]. Simulators have been used since the 1990's and offer [5]:

- Controlled experimental conditions and tasks
- Experimental repeatability and events observability
- Ease of experiment change through parameters or scenarios
- Schedule, cost efficiencies compared to highway or proving ground experiments

[1] http://www.inrets.fr/ur/sara/Pg_simus_e.html last accessed 08.09.2012
[2] http://en.wikipedia.org/wiki/Driving_simulator last accessed 14.10.2012

- Safety to the driver.

Other computer programs that provide driving a car are driving games. They focus on high end graphics and a realistic physics engine. They look very realistic and support driving wheels and multiplayer. But they all provide fictitious cities and are mostly closed source.

There is one simulator though, that already comes very close to the one introduced in this thesis. Geoquake Driving Simulator[3] (see Figure 2.1) is a simulator that allows the user to stear a car over a satellite-image based plane. The terrain construction is very similiar to the one introduced in this thesis. It can be sometimes seen, how new tiles appear at the border of the terrain. The camera angle is also third person, looking diagonally from behind on the car. But it is a browser based application. The source code is closed, and it offers no network support, road informations or physics engine.

Available Data

Satellite Images

Due to the large amount of data needed to store satellite textures from the whole globe, or even a single country, it is the best way to make use of online image providers such as Google Maps or Microsoft's Bing. This allows to store only images that are actually needed. Using this type of source, Hypertext Transfer Protocol (HTTP) can be used to receive the images. This way, it is also easy to switch to another image provider by simply modifying the Uniform Resource Locator (URL) string.

Google Maps offers an Application Programming Interface (API) for receiving satellite images easily via HTTP. Therefore the longitude, latitude and zoom level have to be specified within the URL and downloaded. Another source for satellite images is Microsofts Bing Maps[4]

Map Data

The first choise for map data is Open Street Map (OSM), an up-to-date and free source that covers large regions around the globe. It is a crowdsourcing project with already $35,000$ users and more than $30,000,000$ track points in 2008 [6]. Roads are specified with their names, types and courses. The data can be downloaded for small, user defined regions or the whole globe.

[3] http://geoquake.jp/en/webgame/DrivingSimulatorPerspective/
[4] http://www.bing.com/maps/ last accessed 17.09.2012

Figure 2.1.: Geoquake Driving Simulator

Height Data

Height data for nearly the whole globe is accessable via the Shuttle Radar Topography Mission (SRTM)[5]. It is a collection of height data, gained from a shuttle mission using radar. The resolution for North America is $30\,\mathrm{m}$ and for the rest of the world $90\,\mathrm{m}$ [7]. Raw files that store these data can be downloaded from http://dds.cr.usgs.gov/srtm/version2_1/SRTM3/ (See Section 4.3). These files have a couple of disadvantages though. The height received is not the ground level, but the height where the radar is reflected, like roofs or treetops. Additionally these files have "holes", places the radar was not reflected correctly.

[5] http://www2.jpl.nasa.gov/srtm/faq.html last accessed 10.09.2012

Chapter 3.

Basic Program

The first step in the development of any 3D-application is building a basic framework. This includes to create a window, initialize OpenGL within it, process user inputs, store and load settings and content, etc.

3.1. Basics

The entry point of the application is the *int main(int argc, char* argv[])* function in the *FahrSim.cpp* file. The Operating System (OS) automatically passes the number of arguments *argc* and an array of arguments *argv[]* over to the program. They are used for the following special cases:

- convert an *.osm file to a *.sqlite file and then exit
- host a TCP/IP session
- connect to an existing session via TCP/IP

These cases are explained in detail in later chapters. If arguments are used incorrectly, help messages will be displayed for support.

The $Init()$ functions of various classes are then called, before entering the main-loop. In the main-loop, first all received events from the X server are being handled followed by calls of $Update()$ and $Draw()$. These two functions contain the corresponding ones of all other classes and call them in a specific order. The principle flow is illustrated in Figure 3.1.

Possible events are

- Press of a key
- Release of a key
- Resize of the window

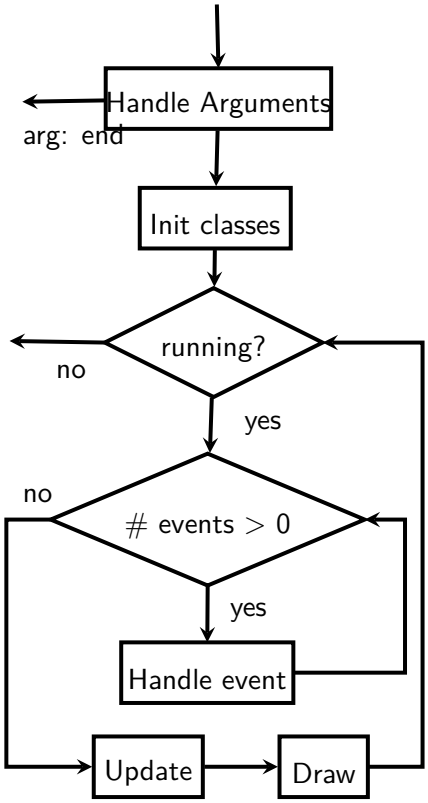


Figure 3.1.: Basic flow of fundamental function calls

- Closing of the window

.

3.2. Integrating OpenGL

Directly after window creation, the $InitGL()$ function within the BaseEnvironment class is called. This function defines a few variables in OpenGL and loads the shader from an external file. The shader is not precompiled. It is basically a textfile (See Appendix C) that is read and compiled during execution time.

In order to draw models in OpenGL, two functions have to be called beforehand within each single frame: $glClear()$ and $glClearColor()$. The first function clears the buffers and the second fills the screen with a given color. This Color represents a simple sky in the simulator and is therefore a light blue. The color may be modified depending on the daytime, becoming black in the night.

Now all drawings can be done using the $Draw()$ function of each object.

Although there exists a depthbuffer to identify which object is in front of another one, the drawing

order is as following:

1. Terrain
2. Objects
3. Local car
4. (Network cars)
5. HUD

The scene is drawn to the background buffer, which has to be set to the front using $glXSwapBuffers()$ in order to be seen by the user.

3.3. Handling Input

User Input from the keyboard is handled by receiving the X server events (see Figure 3.1) $KeyPress$ and $KeyRelease$. The function $XLookupKeysym()$ then extracts the corresponding key and passes it over to the corresponding function $KeyPressed(KeySymkey)$ or $KeyReleased(KeySymkey)$. In those two functions, the variable key is passed over to a switch-statement where the exact actions take place.

Additionally to the keyboard, a stearing wheel is supported by the driving simulator. Using the Simple Directmedia Layer (SDL), anything except the keyboard and the mouse is handled as a joystick [8]. $\mathrm{SDL\_NumJoysticks}()$ queries the number of joysticks available. If this number is zero, the joystick support will be ignored during this session. To connect the stearing wheel to a running simulator, a restart of the application is needed. If this number is larger than zero, then there are joysticks available. The simulator opens the joystick port, defined within the *config.xml* file. In this file it is also possible to deactivate the joystick support. Within every $Update()$ call the values of the driving, accelerate and break axis are received and passed over to the *car class*. The stearing wheel used during the development had two axes. One axis for stearing and the second one for the pedals. When the accelerate pedal is pressed, the axis value increased. When the break pedal was pressed, the axis value decreased. Therefore the accelerate- and break-values can be calculated as follows:

$$\mathrm{acc} = \begin{cases} x/\max & \text{for } x \geq 0 \\ 0 & \text{for } x < 0 \end{cases} \tag{3.1}$$

$$\mathrm{break} = \begin{cases} 0 & \text{for } x \geq 0 \\ -x/\max & \text{for } x < 0 \end{cases} \tag{3.2}$$

$$\text{stear} = -0.3 \cdot x/\max \tag{3.3}$$

x represents the current value of the axis and max the largest value it can gain.

3.4. Loading and Storing Settings

Settings will be stored in the *config.xml* file. Using the libxml2[1] library, the *Settings* class was created for reading and writing variables from and to the XML file. This custom class provides the following methods:

- GetX(path)
- SetX(path, value)
- GetXWhereAttr(path, attribute, attributevalue)
- GetXAttr(path, attribute)
- SetXAttr(path, attribute, attributevalue)

X represents one of the following datatypes: int, double, string or bool. An example would be the following function call

```
1 int port = Settings::GetSettings()->GetIntAttr("/FahrSim/Input/Joystick",
2 "Port");
```

which returns the value 3 in the following pseudo-config file

```
1 <FahrSim>
2         <Input>
3                 <Joystick Active="true" Port="3" />
4         </Input>
5 </FahrSim>
```

The other functions are kind of self explaining. $GetX()$ and $SetX()$ access the value within an object, like $< Object > value < /Object >$ and $GetXWhereAttr()$ considers that this objects attribute has got a given value.

[1] http://www.xmlsoft.org last accessed 27.07.2012

3.5. Head Up Display

The Head Up Display (HUD) is a 2D layer at the top of the rendered scene. It delivers the user information about the current speed and the street the car is on.

Figure 3.2.: HUD with streetname and speedometer

The Speedometer basically consists of two images, the background and the needle. The background image is drawn at the same position and orientation. The needle image has the same size, but changes its orientation depending on the users speed.

First an angle α is calculated for the needle depending on the cars speed. To draw the needle image, the value $r = \sqrt{2} \cdot \text{width}$ is also needed. This just represents the length from the center to a corner and requires a quadratic image. The four vertex positions are defined as:

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} \text{center.x} - r \cdot \sin\left(\left[\frac{\pi}{4} + i\pi\right] - \alpha\right) \\ \text{center.y} - r \cdot \cos\left(\left[\frac{\pi}{4} + i\pi\right] - \alpha\right) \end{pmatrix} \quad i = 0...3$$

To display the street name, a background texture is drawn first at the upper right corner. It must be large enough to display the complete streetname. A font was chosen where each letter has the same width. So an upper bound for the width can be easily defined by $w_{upper} = n \cdot w_{letter}$. The street names are stored in UTF-8[2] format. In this format special letters, like the german ä, ö, ü and ßare encoded with two bytes. Therefore the real streetname length may be shorter when these characters appear. To offer a variable street-sign-length, the sign consists of 3 rectangles and is stretched in the center region (see Figure 3.3).

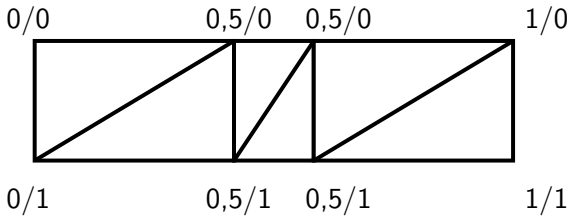


Figure 3.3.: uv-coordinates of street-sign

The letters are displayed using a custom texture that contains all common letters. OpenGL does not support native support for drawing text. Using this texture, letter for letter is taken out of the image and rendered onto a new box.

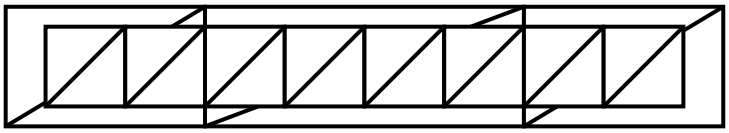


Figure 3.4.: street-sign and individual letters on top

The last feature of the HUD is displaying the driven kilometers. The principle of drawing the letters is the same as drawing the street name to the sign. The texture used is another one and displayed in Figure 3.5.

Figure 3.5.: texture used to draw driven kilometers

The kilometers of the current session automatically start at 0 km at each program start. The total kilometers driven are stored in the $config.xml$ and are updated after any new kilometer driven.

[2] http://de.wikipedia.org/wiki/UTF-8 last accessed 27.08.2012

Chapter 4.

3D-Space

After implementing the basic frame and drawing the HUD in a two dimensional plane, the third dimension is now introduced. When moving from a 2D environment to a 3D one, the first thing to consider is the coordinate system. In the OpenGL standard coordinate system, the y-axis directs up and the x-axis right like in a 2D cartesian coordinate system. The additional z-axis is defined to direct towards the viewer [9], as illustrated in Figure 4.1.

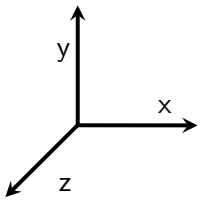


Figure 4.1.: OpenGL coordinate system.

The longitude and latitude of the 3D-Spaces origin is defined in the $config.xml$ and denoted with $\begin{pmatrix} x_0 & y_0 & z_0 \end{pmatrix}^{\mathrm{T}}$ in the following. The scaling can be chosen arbitrarily, but is set to $1\ \mathrm{unit} = 1\ \mathrm{m}$ for simplicity.

4.1. Building the Terrain

Within the 3Dspace, the terrain has to be drawn around the car. In this diploma thesis a flat terrain is used. In Section 4.3, the use of height information using the Shuttle Radar Topography Mission (SRTM) data is introduced, but this data is unsuitable for the purpose of this simulator. It causes many problems using the physics engine, since there are still "holes" in the data, and no functional improvements are gained.

In gerenal, there are two different approaches of the terrain shape. A *rectangular shape*, or an *oval shape*. As this decision only depends on ones preference, it was decided to use a round looking

shape for the following reason. As the car moves into arbitrary directions, the viewing distance stays the same, rather than in a squared shaped terrain, where the user looks farer to the corners, than to the borders. On the other hand, a round shape needs more complex algorithms when created and updated. The terrain size can defined by a radius variable that can be adjusted in the config file.

The atomic units of the terrain are quadratic shapes, called TP, see Section 4.1.1. The TPs do all calculations internally. The job of the terrain class is the creation and destruction of those parts, and the indexing. Indexing means each part is given two indices where they are within the terrain. By changing the indices of a terrain part, it automatically moves to the new destination and updates its look accordingly. The moving process is displayed in Figure 4.2.

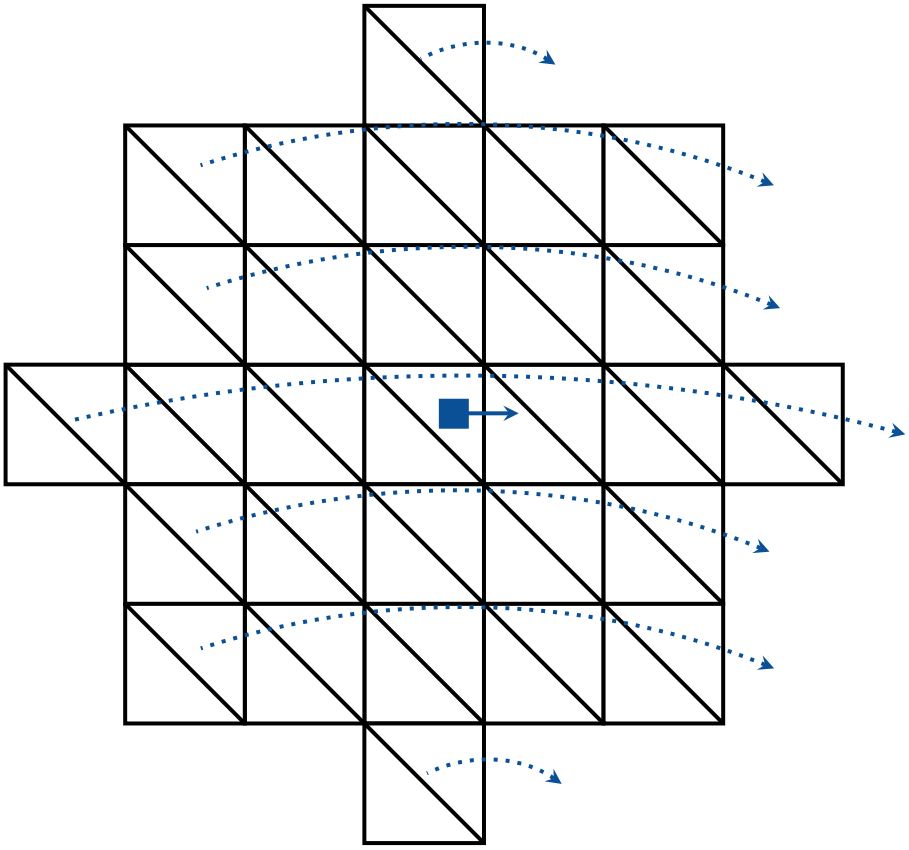


Figure 4.2.: Terrain update after car moves to a neighbouring TP. The car is displayed as the filled square.

It can be seen that only the outer parts move that will disappear after the update process. This makes the update more complex, because the outer parts have to be filtered out first and the indices changed by different values, rather than just adding the value 1 to every part's index. But the more complex code fastens up the process by only moving as few parts as possible, and

prevents flicker of all parts at the same time. If the car moves diagonal, the horizontal and vertical operations are performed one after the other.

Since checking whether the car has left the center part does not need to be done for every frame, and since the update process takes some time, a background thread is created for these operations. This prevents the application from getting stuck during the indexing. The thread is started directly at object creation and runs throughout the whole execution time.

4.1.1. TerrainParts

TPs are quadratic shapes, in OpenGL consisting of 4 vertices defining the edges. These are derived from the center point and the edge length. Furthermore, the satellite texture and the road information (Section 6.2.1) are stored.

The edge length of a TP is limited by the region covered by its corresponding satellite image. Although the edge length could be smaller (and adapting the (u, v) texture coordinates), it will for simplicity match with the satellite images covered region in this thesis. So the size covered has to be obtained first. For the Google Maps Static API [1], the following formula delivers the region (width and height) of a TP:

$$\text{region} = \frac{\text{pixel}}{256} \cdot \frac{\text{equator}}{2^{\text{zoom}}} \cdot \cos\Phi \tag{4.1}$$

Pixel is the edge length of the satellite image used. *Equator* defines the earth radius at the equator. *Zoom* defines the zoom level used when receiving the satellite image from an image provider. Φ is the latitude of the TP's center.

With increasing zoom level, the sector is halved, and by moving towards the poles, the cross-section through the earth parallel to the equator gets smaller and smaller ($\cos\Phi$).

The To define the vertex positions of the shape, the index numbers of the TP are passed over by the terrain class. The indices start from $(0, 0)$ in the origin and increase (right/bottom) or decrease (left/top) with every neighboring TP. The edges of the quadratic shape are calculated as:

$$x_{max/min} = i \cdot \text{region} \pm \frac{\text{region}}{2}$$

$$z_{max/min} = j \cdot \text{region} \pm \frac{\text{region}}{2}$$

[1] https://developers.google.com/maps/documentation/staticmaps last accessed 25.07.2012

The basic C++ drawing code (without the shader part) is listed in the following:

```
1  glEnable(GL_TEXTURE_2D);
2
3  glBindTexture(GL_TEXTURE_2D, this->m_texture);
4
5  glBegin(GL_QUADS);
6  glTexCoord2f(0.0f, 0.0f); glVertex3f(this->m_fXmin, 0.0f,
   this->m_fZmin);
7  glTexCoord2f(0.0f, 1.0f); glVertex3f(this->m_fXmin, 0.0f,
   this->m_fZmax);
8  glTexCoord2f(1.0f, 1.0f); glVertex3f(this->m_fXmax, 0.0f,
   this->m_fZmax);
9  glTexCoord2f(1.0f, 0.0f); glVertex3f(this->m_fXmax, 0.0f,
   this->m_fZmin);
10 glEnd();
11
12 glDisable(GL_TEXTURE_2D);
```

In the first line OpenGL enables drawing objects using a texture. This texture is defined in line three and passed over to the graphics card. The actual drawing of the TP is done in the lines 6 to 9, encapsulated with *glBegin()* and *glEnd()*. Every vertex is drawn by first declaring the texture coordinates and then the vertex position. Finally the texture support is disabled again in line 12.

Another important action takes place when the indices of a TP change. The principle function calls are illustrated in Figure 4.3.

When the background thread of the terrain class notices that the car moved from one TP to another, it starts to call the $SetIndices()$ function of all TP that have to be moved. This function calls the internal $Hide()$ function to prevent the TP from being drawn again. The internal street map is set to an invalide value and the edges of the new position are calculated.

In the next $Update()$ cyle the TP notices the previous steps and starts a new background thread for receiving the satellite image of the new position and creates a new street map.

After the creation of the new two textures, the $Update()$ function recognizes this, deletes the old satellite texture, converts the newly received texture into an OpenGL and makes the TP visible again. The conversion from SDL to OpenGL has to be done in the main thread, because the OpenGL context is only accessible from there.

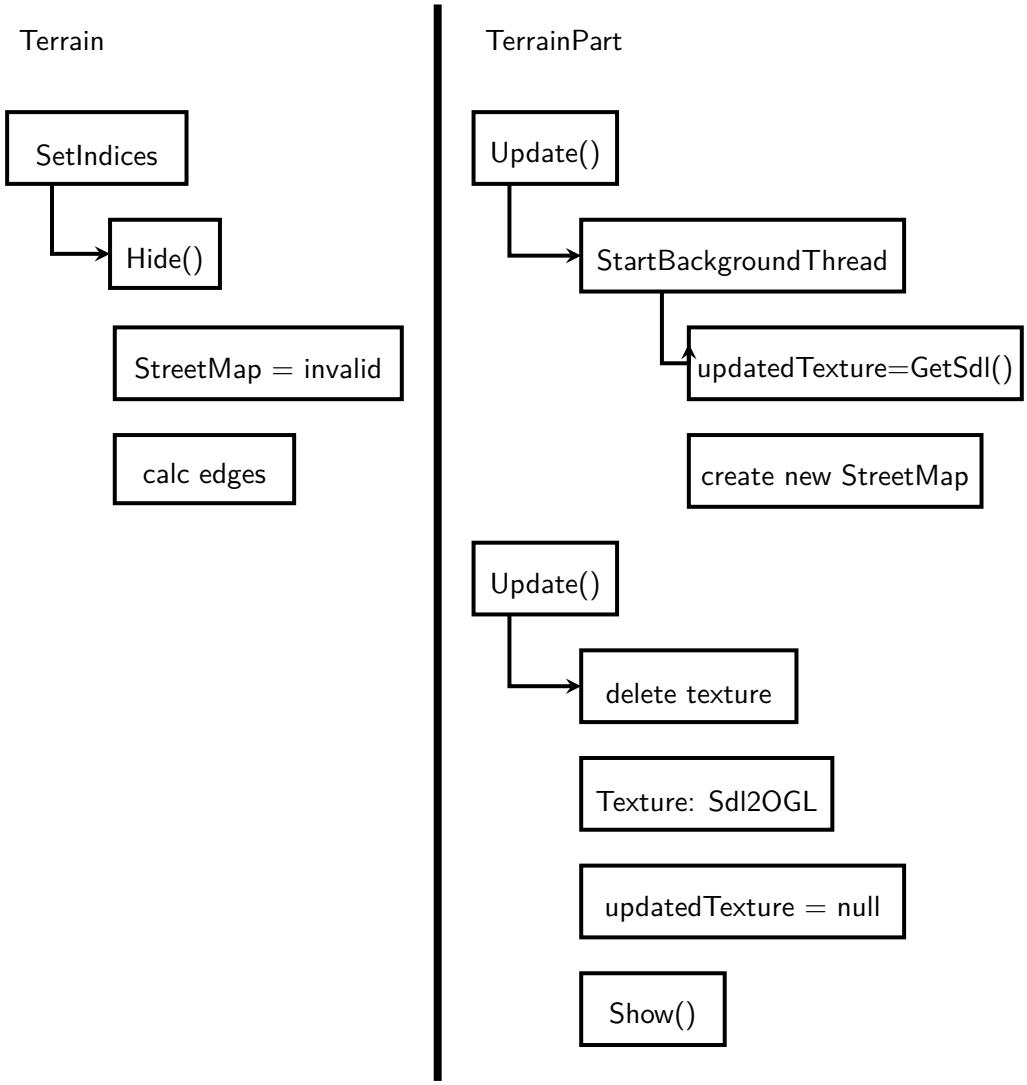


Figure 4.3.: principle flow when moving a TP to a new position

4.1.2. Receiving Satellite Images

Satellite images are the heart of this simulator. Due to the large simplicity and availability nowadays to access this data via the Internet, it is the simplest way to receive them via HTTP, using the libCURL library[2]. This way only actually needed images are used and stored, and large amount of disc space is saved. There exist currently multiple providers, like Google Maps or Microsoft Bing, each one with its own advantages and disadvantages and maybe limitations or availability. It is important to stay independent from a single provider to guarantee a working application, when one provider shuts down or gets problems.

[2] http://curl.haxx.se/libcurl/ last accessed 25.07.2012

Receiving the images is done most of the time in an extra thread. The TextureManager receives image, that is stored in system memory. But OpenGL cannot use this data type directly. It has to be loaded into the Graphics Processing Unit (GPU) first. This has to be done in the main thread, because OpenGL is not thread safe and all its objects do not exist in another thread. The TextureManager offers the $SdlToOgl()$ function for this purpose. It returns a $GLuint$ datatype that represents a handle in order to address the texture at the GPU.

4.1.3. Different Projections of Longitude and Latitude

Due to the previous discussed independence of a certain image provider, multiple different functions have to be implemented to describe the relation between the 3D-space and the used variables defining the satellite image region. The easiest case is a provider, that uses continual longitude and latitude variables to receive the image. The Google Maps would be such a provider. Here, only a conversion between game coordinates and earth coordinates is required. Additionally, there exist so called tile-based providers. They basically have divided a single large image of the earth into multiple pieces, which are addressed using index-numbers.

Therefore, this simulator includes a two step process.

1. 3D-space coordinates to longitude and latitude.
2. longitude and latitude to variables, defining the satellite image region (e.g. tile numbers)

Receiving Longitude and Latitude

The formulas and geographic informations in this section were all taken from the book "The Mercator Projections" [10].

Although the earth is more an ellipsoid than a sphere, the deviation is only $22\ \mathrm{km}$ between the equatorial ($6378\ \mathrm{km}$) and the polar radius ($6356\ \mathrm{km}$). Therefore, in this simulator we neglect this fact for easier and faster computations and assume the earth is a sphere with a radius of

$$a = 6377563.396\ \mathrm{m} \tag{4.2}$$

By looking at the shape of the sphere, a distance along a meridian (δy_{earth}) and a distance parallel to the equator (δx_{earth}) can be derived:

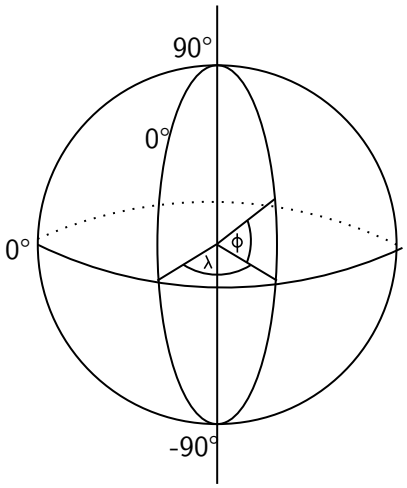


Figure 4.4.: Earth with longitude λ and latitude Φ.

$$\delta x_{\text{earth}} = a\delta\lambda\cos\Phi \tag{4.3}$$

$$\delta y_{\text{earth}} = a\delta\Phi \tag{4.4}$$

transformed to game coordinates yields

$$x_{\text{game}} = x_0 + \delta x_{\text{earth}} \tag{4.5}$$

$$z_{\text{game}} = z_0 - \delta y_{\text{earth}} \tag{4.6}$$

with x_0 and z_0 being the game coordinates, $\delta\lambda$ and $\delta\Phi$ are measured from. λ and Φ are given in radians. For receiving images, they often have to be transformed into degrees first.

These formulas are used inverted. Given a specific terrain part size, λ and Φ of all other parts have to be calculated to receive the matching image. Using the formulas above and the substitutions $x_{game} = i \cdot width_{part}$ and $z_{game} = j \cdot height_{part}$ yields:

$$\lambda = \lambda_0 + i \cdot \frac{\text{width}_{\text{part}}}{a\cos\Phi} \tag{4.7}$$

$$\Phi = \Phi_0 - j \cdot \frac{\text{height}_{\text{part}}}{a} \tag{4.8}$$

Mercator based tiles

When using tile based images, (λ, Φ) have to be transformed to an x- and y-tile number. Using the Mercator Projection, which is described in detail in the Book "The Mercator Projections" [10], the following formulas are used to receive the tilenumbers [3]

$$n = 2^{\text{zoom}} \tag{4.9}$$

$$x_{\text{tile}} = n \cdot \frac{\lambda + \pi}{2 \cdot \pi} \tag{4.10}$$

$$y_{\text{tile}} = \frac{1 - \frac{\ln(\tan\Phi + \sec\Phi)}{\pi}}{2} \cdot n \tag{4.11}$$

Transforming back to longitude and latitude:

$$\lambda = \frac{x_{\text{tile}} \cdot 2\pi}{n} - \pi \tag{4.12}$$

$$\Phi = \arctan\left(\sinh\left(\pi \cdot \left(1 - \frac{2 \cdot y_{\text{tile}}}{n}\right)\right)\right) \tag{4.13}$$

Using tile based images, one (x, y) is projected to a unique (λ, Φ).

$$(\lambda_c, \Phi_c) = \{(\lambda, \Phi) \mid (x, y) \to (\lambda, \Phi)\}\, \forall x, y \tag{4.14}$$

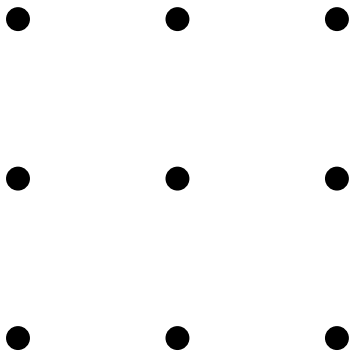


Figure 4.5.: (λ_c, Φ_c) obtained from transforming (x, y) to (λ, Φ)

But in the other direction, a range $([\lambda_i, \lambda_{i+1}], [\Phi_j, \Phi_{j+1}])$ is projected to a single (x, y). That means a $\Delta\lambda$ and $\Delta\Phi$ appears when projecting a $(\lambda, \Phi) \notin (\lambda_c, \Phi_c)$.

[3] http://wiki.openstreetmap.org/wiki/Tilenames last accessed 25.07.2012

$$(\lambda, \Phi) = (\lambda_c + \Delta\lambda, \Phi_c + \Delta\Phi) \tag{4.15}$$

Figure 4.6 illustrates this. The dotted boxes are the regions covered by the tile images. By projecting them to the black TPs, they are shifted by the dotted offset arrows.

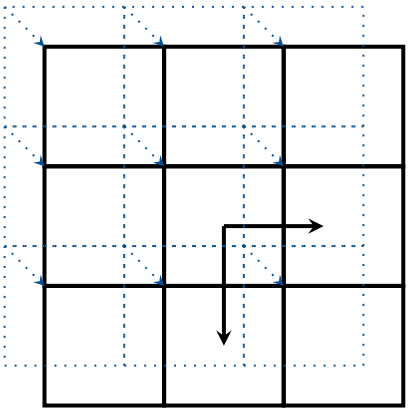


Figure 4.6.: dotted boxed representing tile textures projected to TPs, causing a $\Delta\lambda$ and $\Delta\Phi$

This shifting yields to an incorrect relation between the rendered satellite image and the real longitude and latitude. The car seems to be shifted, the roads do not match with the image, and so on. Instead of introducing a $\Delta\lambda$ and $\Delta\Phi$ in all calculations, this effect can be avoided by modifying the center TPs longitude and latitude to match the next tiles center beforehand.

$$(\lambda_{\text{corrected}}, \Phi_{\text{corrected}}) = (\lambda_{\text{center}}, \Phi_{\text{center}}) \rightarrow (x, y) \rightarrow (\lambda, \Phi) \tag{4.16}$$

This causes that $(\lambda_{\text{corrected}}, \Phi_{\text{corrected}}) \in (\lambda_c, \Phi_c)$. (λ, Φ) of all other TPs do not need such a correction, as they relate on $(\lambda_{\text{center}}, \Phi_{\text{center}})$. This also simplifies the caching of the images (Section 4.1.4) when not using tiles, because images can be reused from different starting points. They no longer differ in very small deviations. Using tile-based images there are no new formulas needed to achieve this modification. (λ, Φ) are transformed to the (x, y) integer tile numbers, and then transformed back to (λ, Φ). There is one difference though. The tile-based images use the upper left corner as the origin, whereas the simulator uses the center. To apply the correct longitude and latitude to the TP, the center between the actual and the next TP are taken.

4.1.4. Caching images

To improve the performance and to reduce the requests to the image provider - who may have limitations in the number of downloads per day - it is recommended to cache the received images

to the local Hard Disc Drive (HDD). The libjpeg [4] was used to save the images in a compressed format. Before downloading a new image from the network, the HDD is checked whether this file already exists. If it exists then it is loaded locally, otherwise the image is downloaded (See Section 4.1.2).

4.2. Camera

Unlike First Person games, it is not neccessary to rotate the camera around its own axis depending on user input. This simplifies the camera class, as no quaternions are needed for rotation operations. The camera for the Driving Simulator is basically defined by two points. The camera position and the camera target.

The following different camera angles are currently supported:

- Top-Down
- In-Car
- Diagonal
- Fixed-Points

Top-Down perspective will always stay above the car, keeping the own orientation with the top facing towards north. *In-Car* is the first person perspective, showing the view of the actual driver. *Diagonal* will stay behind the car in a given height. *Fixed-Points* perspective stays fixed at one location, keeping the focus on the car, and only updates its position when the car has moved too far away.

Except for the *In-Car* perspective, the camera-target is always the car position. The explicit formulas for the various camera perspectives are listed below. The first vector is the camera position, the second is the camera target, and the last one the up-vector. The up-vector is needed by OpenGL to define where the cameras upper edge is facing to. It is used to roll the camera, especially when looking straight up or down. The variable h is the height that is being stored for every perspective independently.

In-Car:

$$\begin{pmatrix} car.x \\ car.y + 1.8 \\ car.z \end{pmatrix}, \begin{pmatrix} car.x + h \cdot \sin(car.roty) \\ car.y \\ car.z + h \cdot \cos(car.roty) \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{4.17}$$

[4] http://libjpeg.sourceforge.net last accessed 27.07.2012

Diagonal:

$$\begin{pmatrix} car.x - h \cdot \sin(car.roty) \\ car.y + h \\ car.z - h \cdot \cos(car.roty) \end{pmatrix}, \begin{pmatrix} car.x \\ car.y + 3.0 \\ car.z \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{4.18}$$

Fixed-Points:

$$\begin{pmatrix} h \cdot \lfloor car.x/h \rfloor \\ car.y + h \\ h \cdot \lfloor car.z/h \rfloor \end{pmatrix}, \begin{pmatrix} car.x \\ car.y + 3.0 \\ car.z \end{pmatrix}, \begin{pmatrix} 0.001 \\ 1 \\ 0 \end{pmatrix} \tag{4.19}$$

For the Fixed-Points perspective, the camera position only updates when $\text{distance} > 2.5 \cdot h$. *distance* is the distance between the camera and the car. The value 0.001 in the up-vector is needed for the case that $h \cdot \lfloor car.x/h \rfloor = car.x$ and $h \cdot \lfloor car.z/h \rfloor = car.z$. The camera is directly above the car. If the up-vector is also directly upwards, a strange behaviour of the camera can be observed. To prevent this, a small ϵ can be used in one of the other two vector components, here 0.001 to the x-component.

Top-Down:

$$\begin{pmatrix} car.x \\ car.y + h \\ car.z \end{pmatrix}, \begin{pmatrix} car.x \\ car.y \\ car.z \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} \tag{4.20}$$

4.3. Height Data Applied To The Terrain

Yet the terrain is completely flat. By setting the *Fahrsim/Terrain/UseHeightData* attribute within the $config.xml$ to true, the SRTM height data is applied. There are a couple of changes that have to be made to read the data, apply them to the terrain, possibly refine the TP, and modify the physical world. This section is for research only, and has no advantages for the purpose of this simulator.

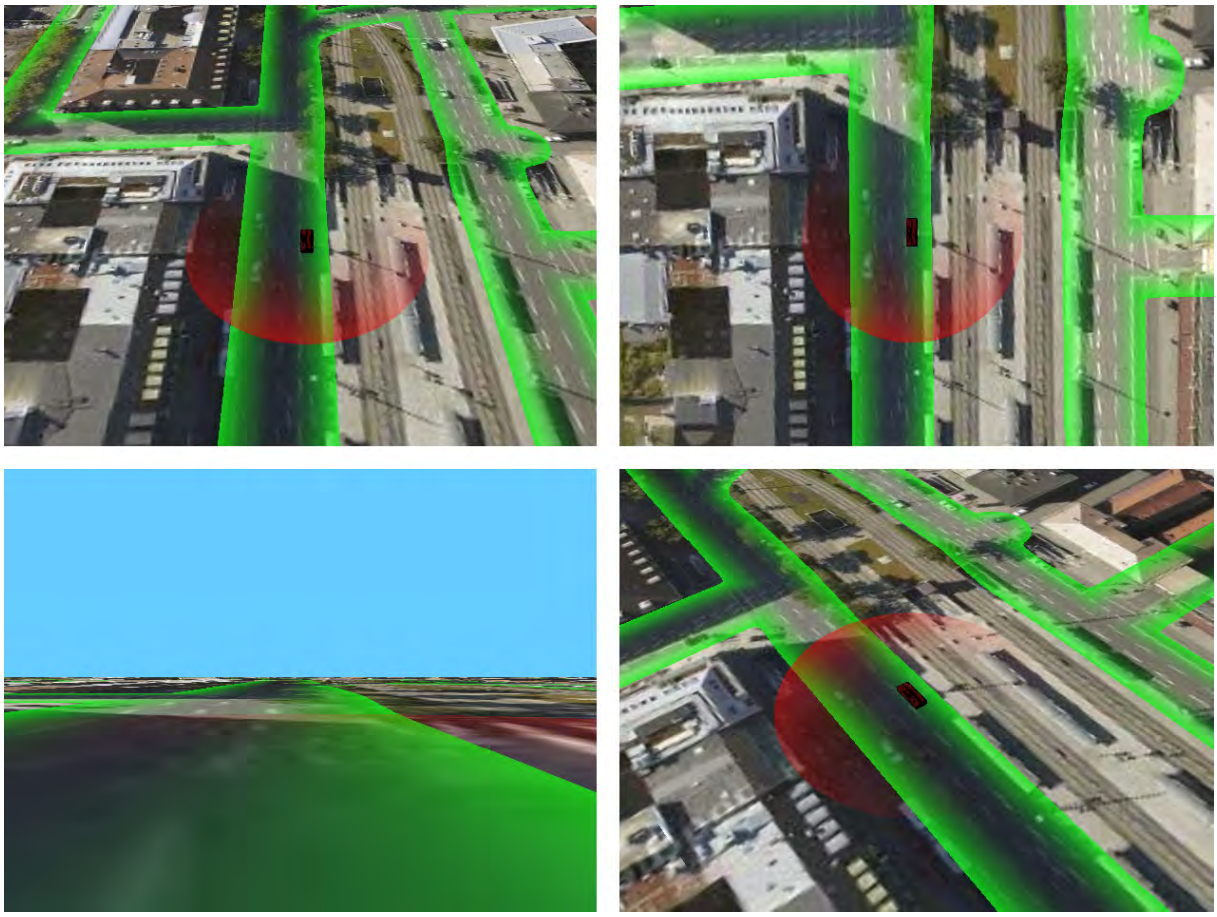


Figure 4.7.: different camera angles (First row: Diagonal, Top-Down. Second row: In-Car, Fixed-Points)

4.3.1. Reading the *.hgt files

*.hgt files store elevation information, generated by the SRTM. A space shuttle orbited the world 176 times and received the height data using a radar[5].

Each *.hgt file covers an area of $1\,\mathrm{degree}$ by $1\,\mathrm{degree}$ and stores the data in raw format. The format is simply a list of 16bit signed integers. The files can be downloaded from http://dds.cr.usgs.gov/srtm/version2_1/SRTM3.

To read the file, a stream is opened. Only actual needed data is read on the fly using the following lines of code:

```
1 this->m_stream.seekg(pos, ios::beg);
2 char buffer[2];
3 this->m_stream.read(buffer, 2);
```

[5] http://www2.jpl.nasa.gov/srtm/faq.html last accessed 10.09.2012

```
4 signed short result = (buffer[0] << 8) + buffer[1];
```

The result value corresponds to the elevation im meters above or beneath the ocean level.

4.3.2. Modifying The TPs

Using the data received above, the TPs have to be modified to display the height properly. Before applying the data, the TPs have to be refined. For test purposes, a refinement level for all TPs is specified in the /FahrSim/Terrain/Detail attribute within the $config.xml$ file. Basically a TP with level n is divided into $n \cdot n$ smaller squares. The y-coordinate for the vertices is received from the $HeightManager$ class using the method from the previous section with the command:

```
1 HeightManager::GetManager()->GetHeight(longitude, latitude);
```

Image 4.8 shows an example of the simulator using the height data.

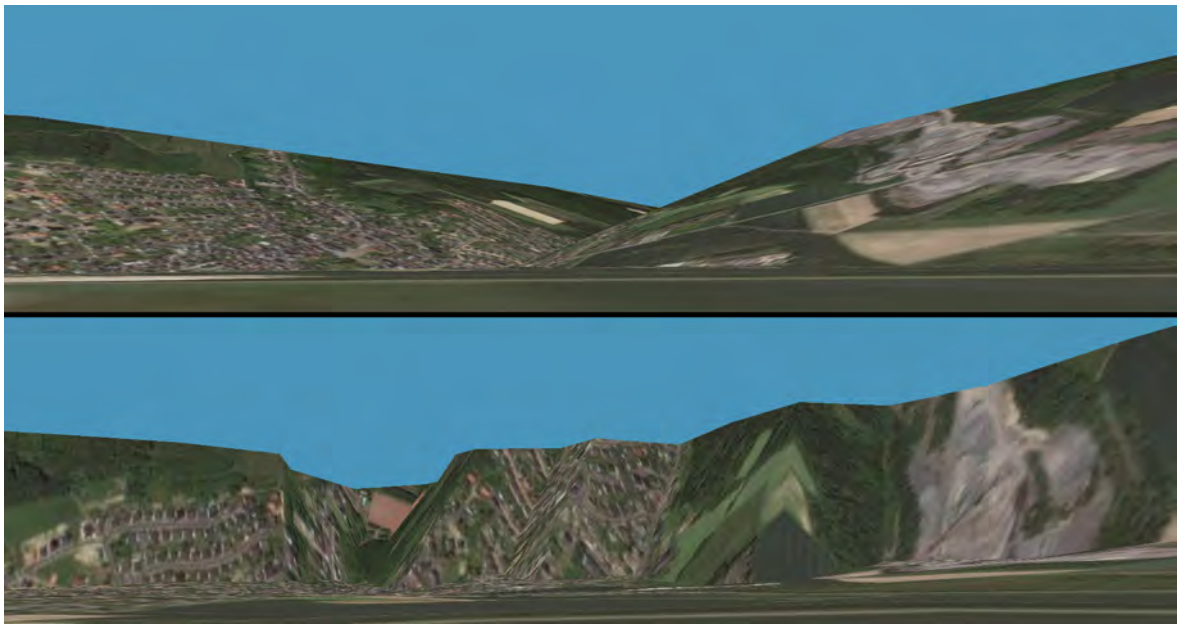


Figure 4.8.: terrain using height (same scene: First row: detail level 1. Second row: detail level 4)

4.3.3. Adding Height Information To The Physics Engine

When the simulator is now started, the car will fall through the terrain, until reaching the physical floor at zero height. Therefore, every triangle within the terrain has also be added into Bullet-Physics. When the TP moves, these triangles have to be removed, the new ones calculated again and added into the dynamics world.

4.3.4. Evaluation

At this point, including elevation data is not a good idea. The available data is much too inaccurate, which leads to a lot of problems. Looking at Figure 4.8, a higher detail level makes it impossible to drive through the village. With increasing detail level, the number of vertices increases by $\mathcal{O}(n^2)$. Even if more accurate data was available, much more advanced terrain algorithms have to be used (like Level of Detail (LOD)) for fluent display. It also can be seen that this drawing method is not a good solution, as with increasing detail level the terrain gets more and more upright at two neighboring height values. An interpolated terrain should be introduced which however adds a new challenge to the physics engine.

4.4. 3D Models

When displaying models such as the car shape or obstacles on the terrain, they first have to be modelled in an external program. There exist multiple different applications, such as $Blender$, $3dsMax$ or others to create the 3D modells.

After creation, a model has to be exported into a format the driving simulator can read, and that guarantees an independence from modelling applications. For simplicity reasons the ASE file format was choosen. It is an ASCII representation of the vertices and how the textures are projected onto them. The code for reading an $*.ase$ file was taken from the book "OpenGL Game Development" [9]. The file reading and drawing logic is encapsulated into the $GameMesh$ class.

This $GameMesh$ is used as a member by the Car and $WorldObject$ classes to handle the OpenGL drawing logic. Those two classes receive their position and rotation from external sources such as the physics engine or the network interface. These positional information will be passed over to the mesh in order to draw the car chassis or its wheels at the correct location and orientation.

Chapter 5.

Physics

This chapter describes the behaviour of the car and obstacles within the 3D world.

The car can be in two different modes:

- active mode: using physics to describe the behaviour
- passive mode: gets the position and rotation from external sources

Driving a single car in the world will remain the active mode. The passive mode is introduced when other cars join the session. There are two scenarios.

Playback of a recorded session

Although this thesis will not deliver applications for this purpose, it offers the interface for displaying car movements. The origin of these movements does not matter. The sources could be recorded in a car, sent live from a car, or exported from this simulator. They will all use the Network interface (Section 7). These scenarious do not need the Physics engine, just the positions and orientations for given moments to display the cars. Therefore these cars will stay in the passive mode.

Multiple cars

The multiplayer is another situation where cars are in passive mode. The server has for example a car in active mode, driving around the city. Another client connects and wants to interact with obstacles and the servers car. At the server side, there now exist two cars, both in active mode. All physical calculations are performed at the server. The client sends the stear, accelerate and break commands though for its own car to the server. But at the client both cars are represented in passive mode, receiving the positions from the server continuously for proper display.

The own car is accessable via *Car::GetLocalCar()* and all other cars are stored in an array, accessable via *Car::GetGlobalCars()*.

5.1. Cardynamics

All calculations are performed by the *BulletPhysics*[1] engine. This engine was the first choice as it is open-source, also programmed in C++ and is used in many other projects such as Hollywood movies and games (Shrek 4, Megamind, Cars 2 game,...). Another benefit for using this engine is that it has vehicle support out of the box.

The engine has internally a world that is independent from the simulators one. The coordinate system and the gravity are defined within. An infinite plane is added to the origin to prevent the objects from falling beneath the terrain.

Within this engine, a new vehicle has to be added first. It consists of a box for the chassis and four cylinders for the wheels. The box is assigned a mass and the wheels at given positions:

- Suspension stiffness
- Damping relaxation
- Damping compression
- Friction slip
- Roll influence

To drive the car, three values are neeeded:

- Stearing angle
- Accelerate value
- Break value

these are assigned to the specific wheels and the engine calculates the car behaviour.

[1] http://bulletphysics.org/wordpress last accessed 06.08.2012

5.2. Collisions with other cars and obstacles

When driving with multiple active cars, collisions are handles automatically by the physics engine. Additionally, the simulator offers adding obstacles to the world. They are once added into the world class for rendering as a mesh, and into the engine as spheres for fast physics calculations.

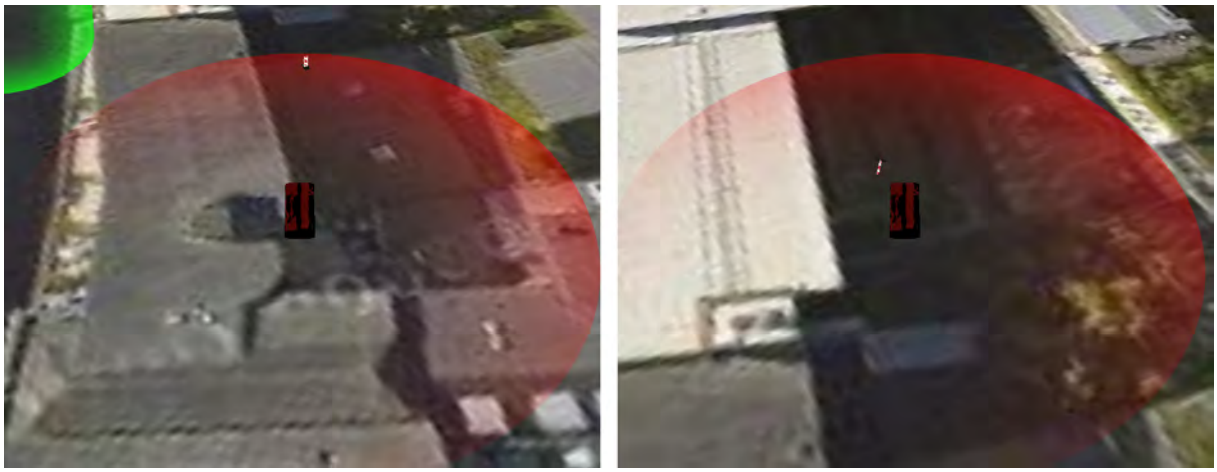


Figure 5.1.: Collision with a beacon

When a collision happened, a notification about this event may be send from the server to a client. This is done by calling the

```
1 NetworkInterface::GetInterface()->CollisionNotify()
```

function. This will add this collision to a Standard Template Library (STL) queue within the network interface. The STL provides a collection of containers, algorithms and iterators for arbitrary datatypes [11]. In the next send cycle, the notifications in the queue are forwarded to the specific clients.

A good example to check if a collision happened can be found in the CollisionInterfaceDemo of the BulletPhysics examples. Basically all combinations of physical objects are received. Because only collisions between cars and objects are of interest, all combinations that contain at least one static object (like the floor) are discarded. If the number of contacts is larger than zero, a collision is currently happening. To trigger only one notification per collision, a STL list is used that contains all collisions from the last frame. Only if this list does not contain the current collision id, the $CollisionNotify()$ function is called and the id is added into the list. If there is no collision in a later frame, the id is removed again.

manifold(0)	manifold(1)	manifold(2)	collision list	CollisionNotify()
0	0	0		
0	0	0		
1	0	0	0	0
1	0	0	0	
1	1	0	0,1	1
1	1	0	0,1	
0	1	0	1	
0	1	1	1,2	2
1	1	0	0,1	0
1	1	0	0,1	

An example of how this works is illustrated in Table 5.2. The Physics engine has internally a manifold table, assigning each combination of physical objects a value of the number of contacts. In Table 5.2, a manifold of 3 objects is displayed. The combinations would be:

- manifold 0: object 0 and object 1
- manifold 1: object 0 and object 2
- manifold 2: object 1 and object 2

if a collision between two objects happens, the manifold value would be greater than zero. The manifold value is then added to the collision list, and a *CollisionNotify()* is called if this manifold value did not exist in the list before.

Chapter 6.

Mapdata

For displaying the road on the terrain, checking if the car is on the street and receiving the current street name, the driving simulator needs street information. OpenStreetMap (OSM) as a free source, a large coverage and up-to-date maps is the choice number one. OSM files can be downloaded using several APIs. A list of these APIs can be found in the official OpenStreetMap wiki[1]. These files cover only a defined region. The whole world would be too much data, when the car only drives within one city. The OSM files have to be downloaded manually, there is no method implemented yet for doing this automatically.

6.1. Conversion to SQLite

Although the OpenStreetMap files contain all information needed, they are not good in terms of speed. The reason is that they have to be completely read beforehand and offer no database typical functions for accessing certain objects rapidly. Additionally they have information stored that is uninteresting for the simulator, such as places of interest. When creating a new TP, only information about near streets are of interest. A database is the best solution, where a single command will rapidly choose the streets of interest. A database server that can be accessed by multiple users, would be much oversized for this project. Therefore, the use of SQLite is a good basis. SQLite offers the functionalty of a database, by only using a single *.sqlite file. To start a conversion, the simulator is started using the $-parse[osm][sqlite]$ argument. After the conversion, the simulator exists.

Looking into an $*.osm$ file, basically three different object types can be found[2]

- Nodes
- Ways

[1] http://wiki.openstreetmap.org/wiki/Getting_Data last accessed 25.07.2012
[2] http://http://wiki.openstreetmap.org/wiki/Elements last accessed 21.08.2012

- Relations

Nodes are the elementary objects in OSM. They are simply points, definded by an ID and its longitude and latitude. Ways are build using a list of nodes that define the road-path. Ways also have an ID and the street name. Relations are not of interest for this simulator, but basically they are also a list of nodes, ways or other relations.

The complete OSM file is parsed and all nodes are copied into the sqlite database. Relations are always discarded. Ways are copied, but only if they can be used by a car. To determine this, the $highway$ value of the way tags is checked. The way is discarded, if the $highway$ tag is empty, or one of the following values: pedestrian, bus_guideway, raceway, footway, cycleway, bridleway, steps, path. The way also needs a streetname in order to be copied.

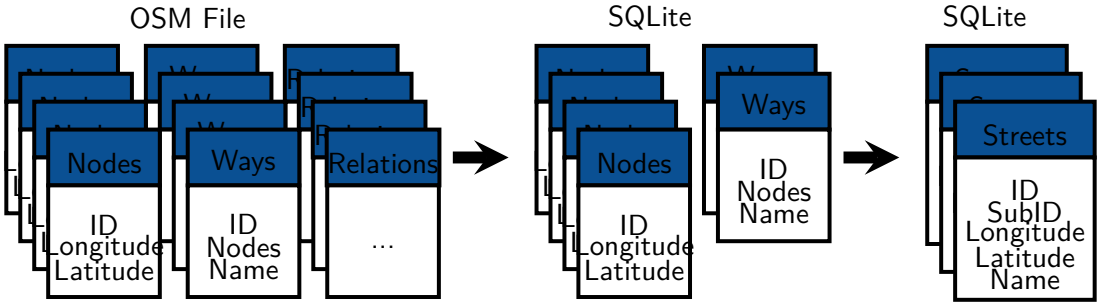


Figure 6.1.: Filtering and Modifying Process of OSM data

The Nodes and Ways format within the database is not suitable for later usage. The simulator needs to filter for ways within one TP. Therefore a new table is created in the database, called Streets. The Streets table consists of the following columns.

1. id: ID of original street
2. subid: segment number within street
3. node1long: longitude of first node
4. node1lat: latitude of first node
5. node2long: longitude of second node
6. node2lat: latitude of second node
7. highway
8. name: streetname
9. lanes: number of lanes

This table creates one row for each streetsegment. The nodes and ways tables are deleted at the end. The whole process is displayed in Figure 6.1.

6.2. Usage

The streets in the SQLite database are used to calculate a street-distance-map for each TP. Therefore all street elements have to be queried from the database using the following SQL command:

*SELECT * FROM streets WHERE (((node1long* $<$ *maxlong AND node2long* $>$ *minlong) OR (node2long* $<$ *maxlong AND node1long* $>$ *minlong)) AND ((node1lat* $<$ *maxlat AND node2lat* $>$ *minlat) OR (node2lat* $<$ *maxlat AND node1lat* $>$ *minlat)))*

This query only returns the streets that are visible on the belonging TP. This reduces the calculation amount immensely, as with every street element less, 128^2 point-to-street distance calculations are saved.

A new empty $128 \cdot 128$ bitmap is created in the system memory. Every pixel is then transformed to (λ, Φ) coordinates. Using this coordinates and the (λ, Φ) coordinates of a street, the distance can be calculated very fast (see Appendix D). The overall distance is the minimum of all distances calculated. The distance is then scaled with a factor of 25.5 and rounded to the next lower integer. If this value is above 255, then it is set to 255 to not exceed the value range. The scaling limits the distance to $10\ \mathrm{m}$ but offers a higher resolution when used later on.

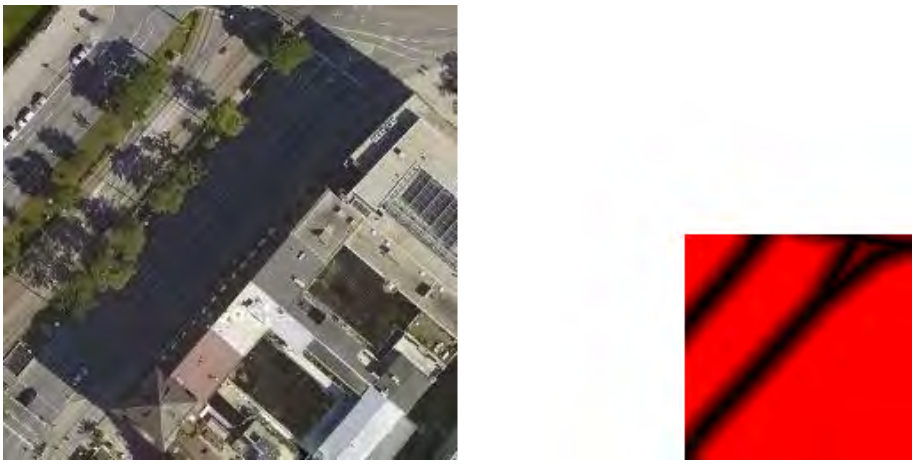


Figure 6.2.: texture and its corresponding distance map

6.2.1. Display road on terrain (Fragmentshader)

In the previous step, the street-distance-image was created, which will be passed over to the fragment-shader. A fragmentshader is executed on the GPU and modifies the displayed texture fragment for fragment. Many effects can be realized by this fast method. In this thesis, the NVIDIA Cg language was chosen as the shading language. Cg is a high-level, C-like language and

supports therefore program portability, improved productivity and makes it easier to develop more interactive programs [12]. Drawing the street basically consists of a street color (like light green) and an alpha value. The texture pixel (outcolor) is modified using the equation:

$$\text{outcolor} = ((1.0 - \text{alpha}) \cdot \text{outcolor}) + (\text{alpha} \cdot \text{streetcolor})$$

The alpha can depend on many factors. In this diploma thesis, the alpha value depends on

- Distance to car
- Distance to center of street
- Cosine pulsing variable to get some dynamics into the rendered scene

Additionally to the road, the shader can be used to highlight the own cars position. Therefore, the cars position is transformed into the TPs (u, v) texture-coordinate-system and passed over to the shader. The (u, v) coordinate-system ranges from $(0, 0)$ to $(1, 1)$ and is used to pick the textures pixel within the shader. Using this information, a circle can be easily drawn around the car on top of the terrain.

Figure 6.3.: scene drawn without (left) and with shader (right)

On Figure 6.3 it can be seen that the transition between two TPs is not that much visible using the shader. This originates from a correction code in the shader that saturates the (u, v) values near the border.

The complete shader code can be found in Appendix C.

6.2.2. Current streetname

Instead of calculating the distance to every street over and over again, for finding the nearest one, it is easier and especially faster to use the previously generated image and just perform a look-up operation. The image will be extended by the use of the blue color channel which will indicate the road. Due to later possible compression algorithms used to cache the image, some deviations in the pixel colors may occur. Therefore, the road-indices are not consecutive $(0, 1, 2, ...)$, but spread linearly over all possible 256 values. The blue value (b) will be encoded by the following formula, using the street's index number withing the street's name array ($index$) and the total number of streetnames in this array ($numstreets$):

$$b = \left\lfloor 255 \cdot \frac{\text{index} + 0.5}{\text{numstreets}} \right\rfloor \tag{6.1}$$

and transformed back:

$$\text{index} = \left\lceil \frac{b \cdot \text{numstreets}}{255} - 0.5 \right\rceil \tag{6.2}$$

In addition to the pixel in the image, an array of street names is generated, as well as the number of names contained. They are created beforehand to assign the correct values to the blue channel.

To obtain the current street name, the pixel value assigned to the current car position is taken, converted back to the index and used to address the name in the array.

6.2.3. Car on or aside of street

The red-channel of the image is taken to determine the distance to the next streets center. To decide whether the car is on the road, or not, the pixel associated to the car is taken. It is converted to the actual distance to the next street. If this distance is smaller than a predefined value, the car is on the street, otherwise not. To consider different street widths, the number of street lanes were taken from the SQLite database and encoded to the green image channel, analogical to the street names in the blue channel. The received number of lanes multiplied by a default lane width, yields an estimation of the actual street width, used for the decision.

Chapter 7.

Network interface

The Driving Simulator introduces a network interface to offer communication with other applications. Using this interface, it is possible to receive all cars and objects positions as well as driving a second car within this world and manipulating objects.

To enable the network interface, the program has to be started using the $-host$ argument. This will start three new threads. One thread waits for new user connections, one receives messages from the clients and one sends the data to the connected clients. The three threads are illustrated in Figure 7.1.

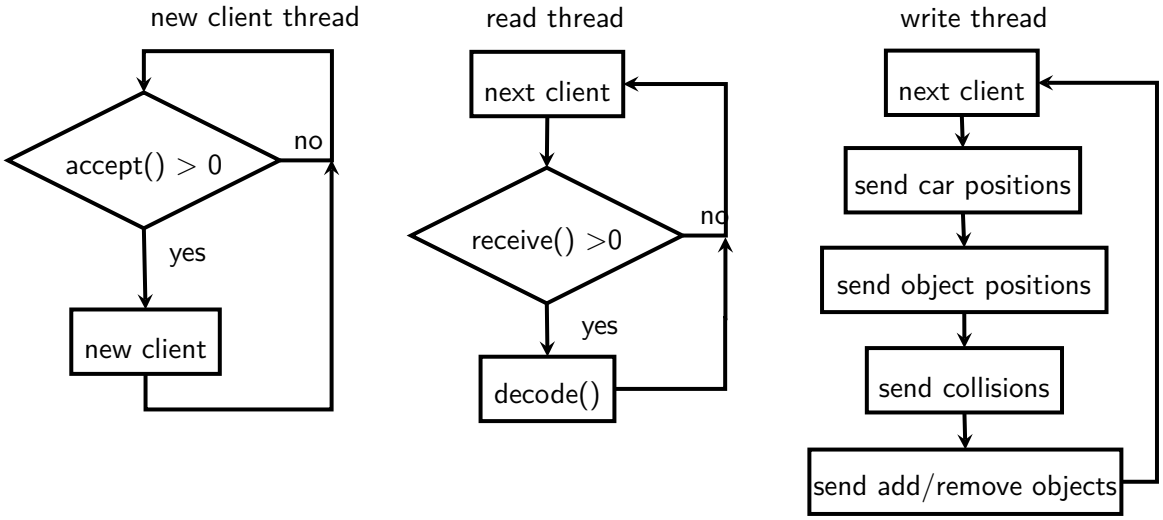


Figure 7.1.: Three server networking threads

To connect to an existing session at a server with another simulator, the $-connect[destination]$ argument is used. The client starts two new threads, one for reading and one for writing operations. The *new client thread* is not necessary. The write thread continuously sends the own position or stearing information, and the read thread receives all informations sent by the client. There are no peer-to-peer connections between multiple clients. The server is always in-between.

Tests show that the separation of the read and write operations by individual threads results in a significant performance boost.

7.1. Messages

All messages sent are encoded into a 128 byte character array. In the following section all messages are explained in detail. The first char will represent the type of message, followed by one char defining the protocoll version. The other bytes are message dependent informations. All Message formats can be found in the appendix.

Hello (0x01)

The Hello Message is the first message to be sent from the client to the server. It tells what informations will be sent from and are desired to be received by the client. These informations are:

- Client drives actively a car (impact on physics)
- Client drives passively a car (no physics, e.g. playback from recorded sessions)
- Client wants continuous position informations of all cars
- Client wants collision notifications

If the active or passive flag is set, a new car is added at the server side. If the car is active, it is added to the physics engine too. Driving such a car will need the $0x20$ messages from the client. If the car is passive the messages $0x21$ are sent from the client to the server.

If the third parameter is set to 1, the server will send continuously the positions of all cars. This is needed to display the cars at the client side. Exceptions where this flag would be 0 could be administrative clients to add/remove objects.

The collision notification is sent from the server to the client if a collision between an objects and a cars, two cars or two objects happened (See Section 5.2).

Hello R (0x02)

Response by the server to 0x01. Basically the same information is sent back. If a car is added, the server checks that the active and passive flag are not both set to one. Additionally the server sends the ID of the added car back.

The client also receives the servers longitude and latitude of the origin.

Drive (0x20)

If using an active car, this message is sent from the client to the server to transmit the

- Accelerate
- Stear
- Break

informations of a car.

Driving State (0x21)

The passive equivalent to the Drive messge. The difference is that no accelerate, break and stear informations are sent, but the actual car position and orientation. This message also serves as a response from the server to the client when using an active car and sending car positions to all other clients.

Add Object (0x30)

A new object has to be added into the world. This command could be sent from an administration application to the server for example.

Add Object Response(0x31)

Response from the server to 0x30. The object ID assigned to the new object is transmitted back.

Remove Object (0x32)

The remove object message can be either sent to the server to delete an object from the world, or being forwarded from the server to the clients to notify them. If the object ID is set to -1, all objects will be removed.

Object Position (0x34)

When a client connects, it receives this messages continuously to get the updated positions of all objects within the world.

Collision Info (0x40)

If the client sent a hello message that requested collision information, this message is sent from the server every time a collision happens.

Disconnect (0xF0)

This message will remove a client from the server.

7.2. Test Program

To test the network interface, a small console application was developed. It basically connects to the server without adding a new car. Instead, the two flags for receiving the cars position and collision notifications are set. The messages MSG_DRIVING_STATE and MSG_COLLISION_INFO are then printed to the console as depicted in Figure 7.2.

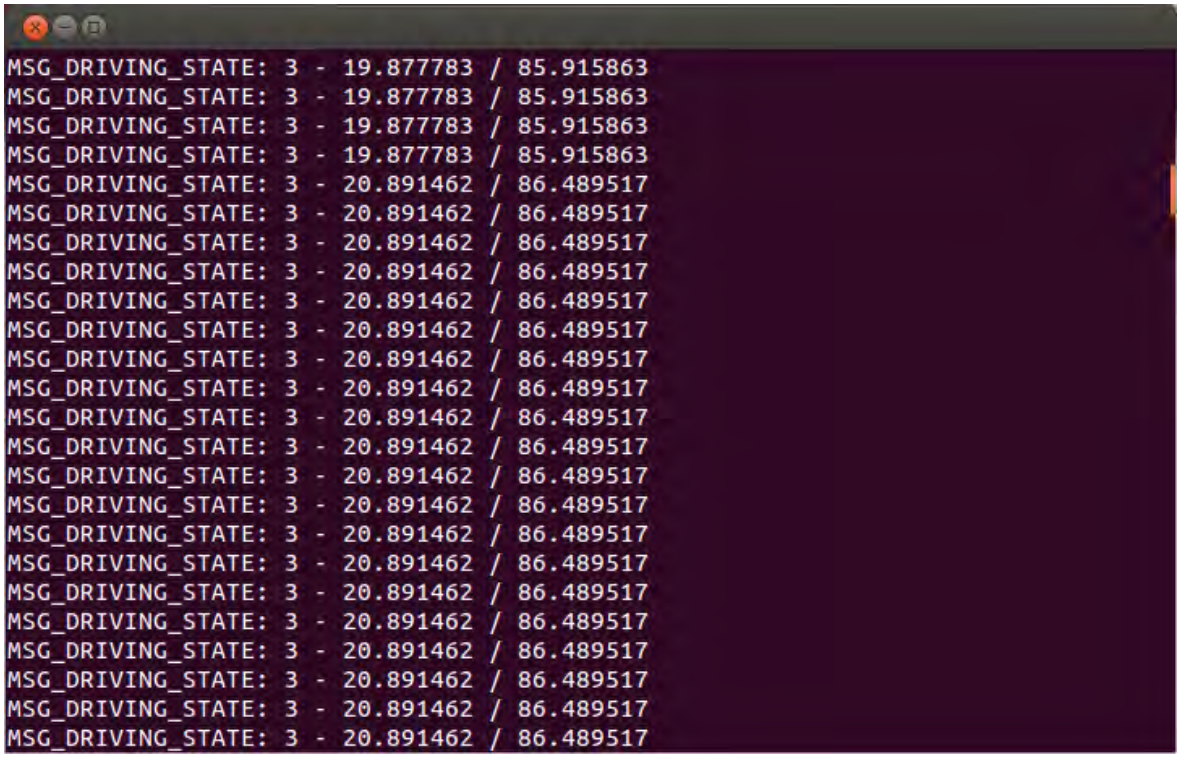


Figure 7.2.: Network Interface Test Program, displaying all received messages

Chapter 8.

Conclusion

Summarized, the following main points have been done within this thesis:

- terrain using satellite images
- roads highlighted on the terrain
- cars and obstacles using physics
- network interface to allow other simulators or appications to connect
- HUD for user informations
- stearing wheel support

There are tons of possible extensions for this driving simulator, and a few are now introduced. One would be the integration of height data. During the creation of this thesis, there was no suitable and complete source for receiving this data. OSM already has support for saving this information in its files, but it is yet hardly used. SRTM is faulty and inaccurate. If better information is available in the future, each TP has to be refined, adding much more vertices. These vertices can be adjusted in the height so that they match the interpolated height, received from the height-data. Other algorithms have to be introduced to allow a high frame rate, such as LOD[13] or frustrum culling.

The OSM files can also be received automatically, instead of doing it manually.

Another idea that came up for often used places is the creation of an additional database. This database could store information about 3D objects at given positions, like trees or traffic lights. This would improve the first person view and the physical interaction of the car with the surrounding world. In this context, much more 3D Objects can be integrated, designed by talented artists.

Another extension could be the improvement of the graphics engine used. Light sources may be added, for example to the car or to traffic and street lights. The sky can also be enhanced to

use a skybox or skysphere (box or sphere around the terrain that is filled with a sky texture) and maybe a lensflare effect to the sun.

More complex algorithms may simulate traffic and control other cars throughout the environment.

Another scenario this simulator could be used for would be the observation of actual cars in realtime. A small app could be written for a smartphone which connects via the Internet to the network interface of the simulator in an office. The app then receives its position via GPS and sends it continuously. The computer then displays the car at the correct location.

Appendix A.

Architecture

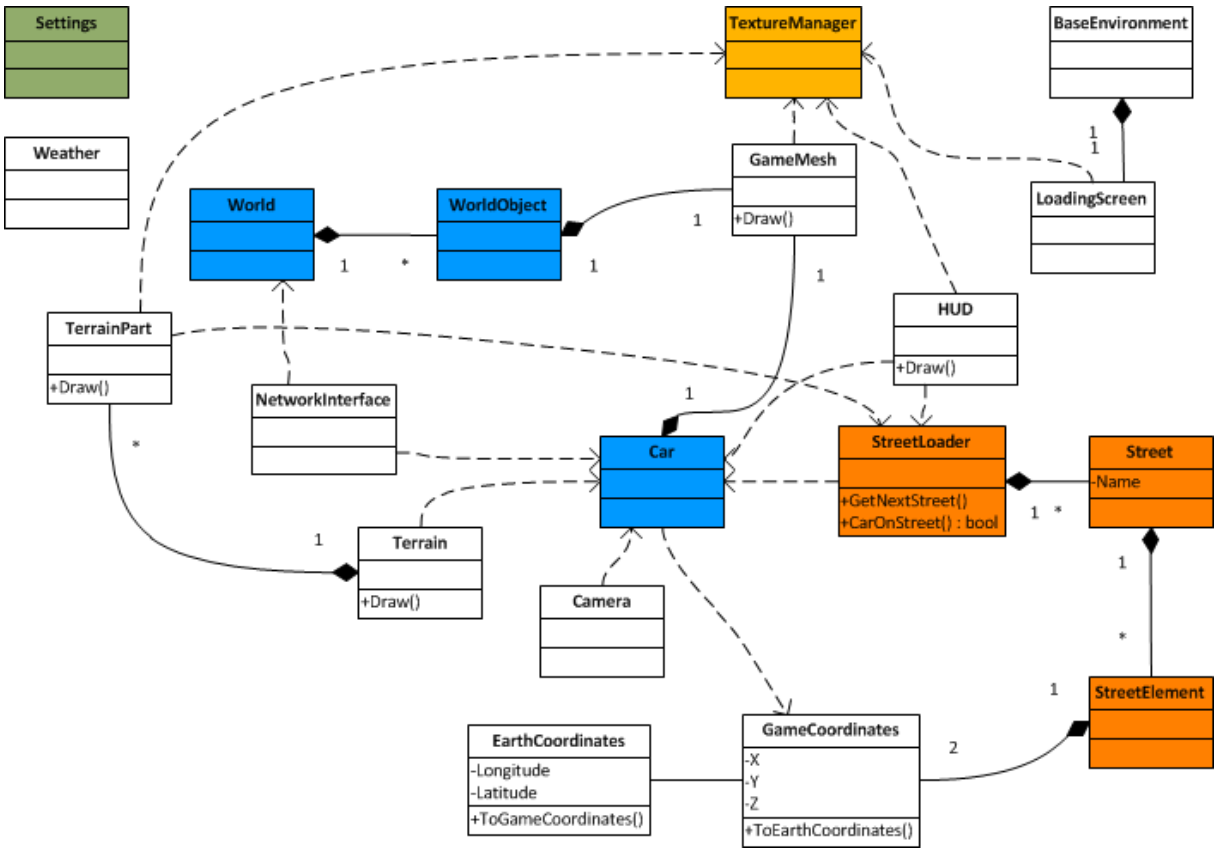


Figure A.1.: Software Architecture: **Blue**: using BulletPhysics; **Orange**: using OSM; **Yellow**: using libCURL; **Green**: using libXML2

Appendix B.

Messages

	Hello							
	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80
0	0x01							
1	version							
2	active	passive	positions	collisions	0	0	0	0
	...							

	Hello Response							
	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80
0	0x02							
1	version							
2	active	passive	positions	collisions	0	0	0	0
3	car ID at server side							
4	origin longitude							
	...							
15	origin latitude							
	...							

	Drive							
	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80
0	0x20							
1	version							
2	car ID							
3	accelerate							
	...							
14	stear							
	...							
25	break							
	...							

	Driving State							
	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80
0	0x21							
1	version							
2	car ID							
3	Δx from origin							
	...							
14	Δz from origin							
	...							
25	rotation angle							
	...							
35	rotation x-axis							
	...							
45	rotation y-axis							
	...							
55	rotation z-axis							
	...							
65	velocity							
	...							

	Object Position							
	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80
0	0x34							
1	version							
2	object ID							
3	Δx from origin							
	...							
14	Δz from origin							
	...							
25	height							
	...							
36	rotation angle							
	...							
46	rotation x-axis							
	...							
56	rotation y-axis							
	...							
66	rotation z-axis							
	...							

	Collision Info							
	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80
0	0x40							
1	version							
3	Δx from origin							
	...							
13	Δy from origin							
	...							
23	Δz from origin							
	...							

Appendix C.

Shader

```
uniform sampler2D streetmap;

//position of car, relative to origin of this tile
float2 carposition;

//size of a single terrain-tile
float terrainpartsize;

//daytime
float daytime = 1.0;

//
//Street informations
//
//number of streetelements within this tile
int numstreets = 0;

//size of street in meter
float streetwidthhalf = 3;

//color of streethighlight
float4 streetcolor = float4(0.0705, 0.9450, 0.0901, 0.8);

//variables for making streets dynamic
float angle2;   //highlight
float angle3;   //outter visibility (opacity=0)
float angle4;   //inner visibility (opacity=1)

```

```
29 //
30 //Circle around car
31 //
32 //size of circle
33 float size = 400;
34 //opacity of circle
35 float alpha=0.5;
36 //angle of circle
37 float angle;
38
39 void main(uniform sampler2D texture,
40 float2 uv : TEXCOORD0,
41 out float4 outcolor : COLOR)
42 {
43
44         //quadratic distance to car
45         float distsq = pow(carposition[0] - (terrainpartsize * uv[0]), 2)
46         + pow(carposition[1] - (terrainpartsize *uv[1]), 2);
47
48         //correcture of texture border (512x512)
49         float2 t = uv;
50         if (uv[0] < (0.001953125)){ t[0] = 0.001953125;}
51         if (uv[1] < (0.001953125)){ t[1] = 0.001953125;}
52         if (uv[0] > (0.998046875)){ t[0] = 0.998046875;}
53         if (uv[1] > (0.998046875)){ t[1] = 0.998046875;}
54
55         //correcture of texture border (256x256)
56         float2 s = uv;
57
58         if (uv[0] < (0.003921568)){ s[0] = 0.003921568;}
59         if (uv[1] < (0.003921568)){ s[1] = 0.003921568;}
60         if (uv[0] > (0.996078431)){ s[0] = 0.996078431;}
61         if (uv[1] > (0.996078431)){ s[1] = 0.996078431;}
62
63         //correcture of texture border (128x128)
64         float2 q = uv;
65
66         if (uv[0] < (0.0078125)){ q[0] = 0.0078125;}
```

```
67          if (uv[1] < (0.0078125)){ q[1] = 0.0078125;}
68          if (uv[0] > (0.9921875)){ q[0] = 0.9921875;}
69          if (uv[1] > (0.9921875)){ q[1] = 0.9921875;}
70
71          outcolor = tex2D(texture, t);
72
73          //daytime (night=darker)
74          outcolor *= daytime;
75
76          //Draw Streets
77          if (numstreets > 0)
78          {
79                  float streetdistance = tex2D(streetmap, q).r / 25.5;
80
81                  if ((streetdistance * terrainpartsize) < streetwidthhalf)
82                  {
83
84                          float distancenormiert = (streetdistance *
85                                  terrainpartsize) / streetwidthhalf;
86
87                          float alpha1 = ((500 + 100*sin(angle3))
88                                  - sqrt(distsq)) / ((500 + 100*sin(angle3))
89                                  - (100 + 20*cos(angle4)));
90                          if (alpha1>1.0) alpha1 = 1.0;
91                          if (alpha1<0.0) alpha1 = 0.0;
92
93                          float alpha2 = 1-cos(1.570796*distancenormiert);
94                          float alpha = alpha1 * alpha2 * streetcolor.a;
95
96
97                          //lighten up street
98                          float li = cos(angle2);
99                          if (li<0.0) li = 0.0;
100                         outcolor *= 1.0 + (alpha1 * (0.4 * pow(li,5)));
101
102                         //draw
103                         outcolor = ((1.0 - alpha) * outcolor)
104                                 + (alpha * streetcolor);
```

```
105                     }
106             }
107
108             //Draw Circle around Car
109             if (distsq < size)
110             {
111                     float alpha2 = (atan2((carposition[0] -
112                             (terrainpartsize * uv[0])), (carposition[1]
113                             - (terrainpartsize *uv[1]))) - angle);
114
115                     if (alpha2 < 0)
116                             alpha2 += 6.28318530717958647692528676655 9;
117
118                     alpha = alpha*abs(pow(cos(alpha2/2.0),3));
119
120
121                     outcolor.r = (alpha * distsq/size) + ((1.0 -
122                             (alpha*distsq/size)) * outcolor.r);
123                     outcolor.g = ((1.0 - (alpha*distsq/size)) * outcolor.g);
124                     outcolor.b =((1.0 - (alpha*distsq/size)) * outcolor.b);
125             }
126     }
```

Appendix D.

Distance Point to Road

Calculating the Distance from a Point to a line is an essential task within this driving simulator. The following formulas are taken from the book Graphic Gems II [14].

$$a_2 = (P_y - A_y)(B_x - A_x) - (P_x - A_x)(B_y - A_y)$$

$$d_1 = \sqrt{\frac{a_2^2}{(B_x - A_x)^2 + (B_y - A_y)^2}}$$

$$t = (P_x - A_x)(B_x - A_x) + (P_y - A_y)(B_y - A_y)$$

if(t < 0)

$d_2 = |AP|$

else

{

$t = (B_x - P_x)(B_x - A_x) + (B_y - P_y)(B_y - A_y)$

if(t < 0)

$d_2 = |BP|$

else

$d_2 = d_1$

}

Appendix E.

Keymap

Key	function
W	accelerate
S	break
A	stear left
D	stear right
Q	toggle camera
+	zoom camera in
-	zoom camera out
H	enable/disable HUD
T	enable/disable shader
I	reset car to origin
K	add beacon object
L	clear all objects
ESC	exit

List of Figures

List of Acronyms

API	Application Programming Interface
GPU	Graphics Processing Unit
HDD	Hard Disc Drive
HUD	Head Up Display
HTTP	Hypertext Transfer Protocol
LOD	Level of Detail
OS	Operating System
OSM	Open Street Map
SRTM	Shuttle Radar Topography Mission
SDL	Simple Directmedia Layer
STL	Standard Template Library
TP	Terrain Part
POI	Point Of Interest
URL	Uniform Resource Locator
XML	Extensible Markup Language

Bibliography

[1] S. Diewald, A. Möller, L. Roalter, and M. Kranz, "DriveAssist - A V2X-Based Driver Assistance System for Android," in *Mensch & Computer Workshopband* (H. Reiterer and O. Deussen, eds.), pp. 373–380, Oldenbourg Verlag, 2012.

[2] S. Diewald, T. Leinmüller, B. Atanassow, L.-P. Breyer, and M. Kranz, "Mobile Device Integration and Interaction with V2X Communication," in *19th World Congress on Intelligent Transport Systems (ITS)*, Oct. 2012.

[3] Penn Wu, Pedro Manrique, "Is java relevant in the game industry?," *The Journal of Computing Sciences in Colleges, vol 27*, 2011.

[4] James Cremer, Joseph Kearney, and Yiannis Papelis, "Driving simulation: Challenges for vr technology," 1996.

[5] D. A. Stall, "The national advanced driving simulator: Potential applications to its and ahs research," 1996.

[6] Mordechai Haklay, Patrick Weber, "Openstreetmap: User-generated street maps," *IEEE*, 2008.

[7] Farr, T. G., et al., "The shuttle radar topography mission," *Rev. Geophys*, vol. 45, 2007.

[8] E. Pazera, *Focus on SDL*. Cincinnati: Premier Press, 2003.

[9] Chris Seddon, *OpenGL Game Development*. Plano, Texas: Wordware Publishing, Inc., 2005.

[10] P. Osborne, *The Mercator Projections*. EDINBURGH, 2008.

[11] *Beginning C++ Game Programming*. Boston: Thomson Course Technology, 2004.

[12] William R. Mark, R. Steven Glanville, Kurt Akeley, Mark J. Kilgard, *Cg: A system for programming graphics hardware in a C-like language*. ACM, 2003.

[13] David Luebke, Martin Reddy, Jonathan D. Cohen, *Level of Detail for 3D Graphics*. San Francisco: Morgan Kaufmann Publishers, 2003.

[14] James Arvo, *Graphic Gems II*. Ithaca, New York: Academic Press, Inc., 1995.