

Projektarbeit

# Fahrimulator

Sandro Ropelato (ropelsan)  
Christof Würmli (wurmlchr)

23. Dezember 2011

**Studiengang**  
Systeminformatik (SI)

## Betreuende Dozenten

Prof. Dr. Peter Früh (frup)  
Prof. Martin Schlup (spma)





## **Zusammenfassung**

Die ETH Zürich betreibt diverse Forschungsprojekte im Bereich des Strassenverkehrs. Für diese Zwecke benötigt sie einen Fahrsimulator, in dem Verkehrssituationen möglichst realistisch nachgestellt werden können. Das Ziel dieses Projektes ist, in Zusammenarbeit mit der ETH, die Erstellung einer Fahrsimulationssoftware für den bestehenden Hardwareaufbau. Um die neue Software in das existierende Framework einzugliedern, ist eine Anbindung der Hardware über LabVIEW zwingend notwendig.

Unter Verwendung von OGRE, einer Open Source Grafikengine wurde ein funktionsfähiger Fahrsimulator entwickelt, der dem Benutzer die Möglichkeit gibt, ein Fahrzeug durch eine virtuelle Stadt oder eine Berglandschaft mit Tunnels zu navigieren. Dabei wird die Perspektive so gewählt, dass der Benutzer die Illusion erhält, tatsächlich im Fahrzeug zu sitzen. Um eine möglichst hohe Qualität der Forschungsergebnisse zu gewährleisten, wurde ein besonderes Augenmerk auf eine möglichst kurze Reaktionszeit des Systems auf Eingaben sowie ein intuitives und realistisches Fahrverhalten gelegt. Sämtliche relevanten Daten werden während der Simulation aufgezeichnet und stehen für weitere Auswertungen zur Verfügung.



## **Abstract**

ETH Zurich promotes various projects in the field of road traffic. For this purpose a driving simulator that recreates traffic situations in the most realistic way possible is required. This project was conducted in cooperation with ETH, department of Innovation and Technology Management and aimed at the development of a driving simulation software for the existing hardware installation. In order to integrate the new software in the present framework, it necessarily had to be linked to the hardware via LabView.

OGRE, an open source graphics engine, was used to develop an operational driving simulator which enables the user to navigate in a virtual city or in a mountainous landscape including tunnels. The perspective deliberately gives the user the illusion of sitting behind the driving wheel of a car. To ensure a high quality of the results, great importance was attached to maximally reduce the reaction time of the system executing tasks and to provide an intuitive and realistic driving behaviour. All the relevant data is continuously recorded during the simulation and therefore can be evaluated at a later date.



## **Erklärung betreffend das selbständige Verfassen einer Projektarbeit an der School of Engineering**

Mit der Abgabe dieser Projektarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppenmitglieder nicht als fremde Hilfe.)

Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Projektarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

**Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.**

Ort, Datum:

Unterschriften:

.....

.....

.....

.....

Das Original dieses Formulars ist bei der ZHAW-Version aller abgegebenen Projektarbeiten zu Beginn der Dokumentation nach dem Abstract bzw. dem Management Summary mit Original-Unterschriften und -Datum (keine Kopie) einzufügen.

## Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>4</b>
1.1 Ausgangslage . . . . .	4
1.2 Aufgabenstellung . . . . .	4
1.3 Grober Zeitplan . . . . .	5
<b>2 Aufbau des Systems</b>	<b>6</b>
2.1 Hardwareaufbau . . . . .	6
2.2 Systembeschreibung . . . . .	7
2.3 Anforderungen . . . . .	8
<b>3 Realisierung</b>	<b>10</b>
3.1 LabVIEW-Programm . . . . .	10
3.2 UDP-Listener . . . . .	10
3.3 Virtual Reality . . . . .	11
3.4 Hauptprogramm . . . . .	22
<b>4 Resultate und Tests</b>	<b>31</b>
4.1 Erreichtes . . . . .	31
4.2 Zeitliches Verhalten des Systems . . . . .	31
4.3 Testfälle . . . . .	31
<b>5 Nächste Schritte</b>	<b>32</b>
5.1 Offene Punkte . . . . .	32
5.2 Zusätzliche Funktionen . . . . .	32
5.3 Ausblick auf Bachelor Arbeit 2012 . . . . .	32
<b>6 Nachwort</b>	<b>33</b>
6.1 Rückblick . . . . .	33
6.2 Danksagung . . . . .	33
<b>A Aufgabenstellung</b>	<b>34</b>
<b>B Videoplayer</b>	<b>36</b>
B.1 Ziel . . . . .	36
B.2 Systembeschreibung . . . . .	36
B.3 Realisierung . . . . .	37
<b>C Das OGRE-Framework</b>	<b>42</b>
C.1 Was ist das OGRE-Framework . . . . .	42
C.2 Welche Features bringt OGRE mit? . . . . .	42
C.3 Weshalb wird OGRE eingesetzt? . . . . .	43
<b>D Betriebsanleitung</b>	<b>44</b>
D.1 VideoPlayer . . . . .	44
D.2 Fahrsimulator . . . . .	45

<b>E Screenshots</b>	<b>46</b>
<b>F Detaillierter Zeitplan</b>	<b>48</b>
<b>G Listings</b>	<b>50</b>
G.1 OGRE-Konfigurationsfiles . . . . .	50
G.2 LabVIEW-Programme . . . . .	51
<b>H Journal</b>	<b>53</b>
<b>Glossar</b>	<b>56</b>
<b>Abbildungsverzeichnis</b>	<b>57</b>



## 1. Einleitung

### 1.1. Ausgangslage

Im Bereich der Fahrsimulatoren gibt es eine Vielzahl von verschiedenen Lösungen. Einige davon bestehen aus Filmmaterial, das abgespielt wird und der Fahrer muss auf die Bremse drücken, sobald ein bestimmtes Ereignis eintritt. Andere bringen bereits eine virtuelle Welt mit, in der man sich mehr oder weniger frei bewegen bzw. frei fahren kann. Jedoch sind bei den meisten dieser Fahrsimulatoren feste Szenarien implementiert, die nicht geändert werden können. Die Möglichkeiten eines Fahrsimulators werden hauptsächlich durch die Leistungsfähigkeit des Rechners, auf dem er läuft, limitiert. Eine besondere Herausforderung ist die Simulation von interaktivem Verkehr auf den Strassen.

Dieses Projekt wird in Zusammenarbeit mit der ETH Zürich durchgeführt. Im Rahmen unterschiedlicher Forschungsthemen sollen Probanden im Fahrsimulator verschiedene Strassensituationen antreffen. Eine dieser Forschungen bezieht sich auf die Auswirkung von Medikamenten im Strassenverkehr. Hierbei soll die Aufmerksamkeit und Reaktionsfähigkeit des Probanden vor und nach der Einnahme von Medikamenten getestet werden. Eine andere Studie untersucht die Augenreaktion beim Einfahren in Tunnels. Wenn der Portalbereich eines Tunnels von der Sonne beschienen wird, ist dieser sehr hell, der Innenbereich hingegen ist dunkel. Der Autofahrer ist also einem starken Helligkeitsunterschied ausgesetzt und die Augen müssen sich auf die neue Situation einstellen. Um herauszufinden, wie diese Anpassungsphase die Wahrnehmung des Autofahrers beeinflusst, werden solche Versuche im Simulator durchgeführt.

Die Hardwarekomponenten des Fahrsimulators bestehen aus einem Rechner, einem Projektor, einer speziell bemalten Wand als Projektionsfläche und einem Cockpit mit Autositz und Sicherheitsgurt. Im Cockpit befinden sich Gas- und Brems- und Kupplungspedale, ein Steuerrad mit verschiedenen Knöpfen und ein Schaltknüppel. Auf dem Rechner läuft LabVIEW<sup>(1)</sup>, das als Schnittstelle zwischen Eingabehardware und Simulationssoftware dient.

### 1.2. Aufgabenstellung

#### 1.2.1. Formulierung

Um die ETH bei ihren Forschungen zu unterstützen wird im Rahmen dieser Projektarbeit ein Fahrsimulator für die bestehende Hardwareumgebung entwickelt. Durch Eingaben im Cockpit soll der Proband das Fahrzeug durch eine 3D-Umgebung bewegen können und dabei eine möglichst realistisch Illusion des Autofahrens vermitteln bekommen. Die Szenen sollen unter anderem durch frei verfügbare Modelle aus Google 3D Warehouse<sup>(2)</sup> aufgebaut werden. Alle Betriebszustände und Benutzereingaben registriert und aufgezeichnet werden um eine genaue Auswertung zu ermöglichen.

---

<sup>(1)</sup>Ein grafisches Programmiersystem vom Softwarehersteller National Instruments

<sup>(2)</sup>Eine Plattform, auf der Modelle aus Google SketchUp veröffentlicht und kostenlos heruntergeladen werden können.

<http://sketchup.google.com/3dwarehouse>

### **1.2.2. Aufteilung der Arbeit**

In einem ersten Schritt wird die Schnittstelle zwischen dem Cockpit und dem zu realisierenden Programm über LabVIEW implementiert. Zur Visualisierung wird ein Video-Player entwickelt, der bei Betätigung des Gas- und Bremspedals die Abspielgeschwindigkeit des Videos anpasst. Beim Videoplayer soll zudem die Speicherung von Daten in Log-Files implementiert werden. Danach wird mit der erstellten Schnittstelle der Fahrsimulator realisiert. Ist dieser implementiert und getestet, werden verschiedene Szenen erstellt.

### **1.3. Grober Zeitplan**

Die ersten zwei Wochen werden dazu verwendet, die Simulationsumgebung kennenzulernen. Danach werden weitere vier Wochen eingeplant um die Schnittstelle zwischen dem Cockpit und dem zu realisierendem Programm zu implementieren und sie mit einem Videoplayer zu testen. Für den Fahrsimulator und die zu erstellenden Szenen werden ca. vier Wochen eingeplant. Die letzten vier Wochen werden für die Dokumentation und eventuelle Ergänzungen benötigt. Ein detaillierterer Zeitplan ist im Anhang F zu finden.

## 2. Aufbau des Systems

### 2.1. Hardwareaufbau

Ein Steuerrad, drei Pedalen, ein Schaltknüppel und diverse Knöpfen dienen als Eingabegeräte für den Fahrsimulator. Diese sind über USB mit einem Computer verbunden. Als Ausgabe dient ein leistungsstarker Projektor, der das Bild auf eine hochreflektierende Projektionsfläche projiziert.



Abbildung 1: Hardwareaufbau mit Steuerrad und Pedalen

## 2.2. Systembeschreibung

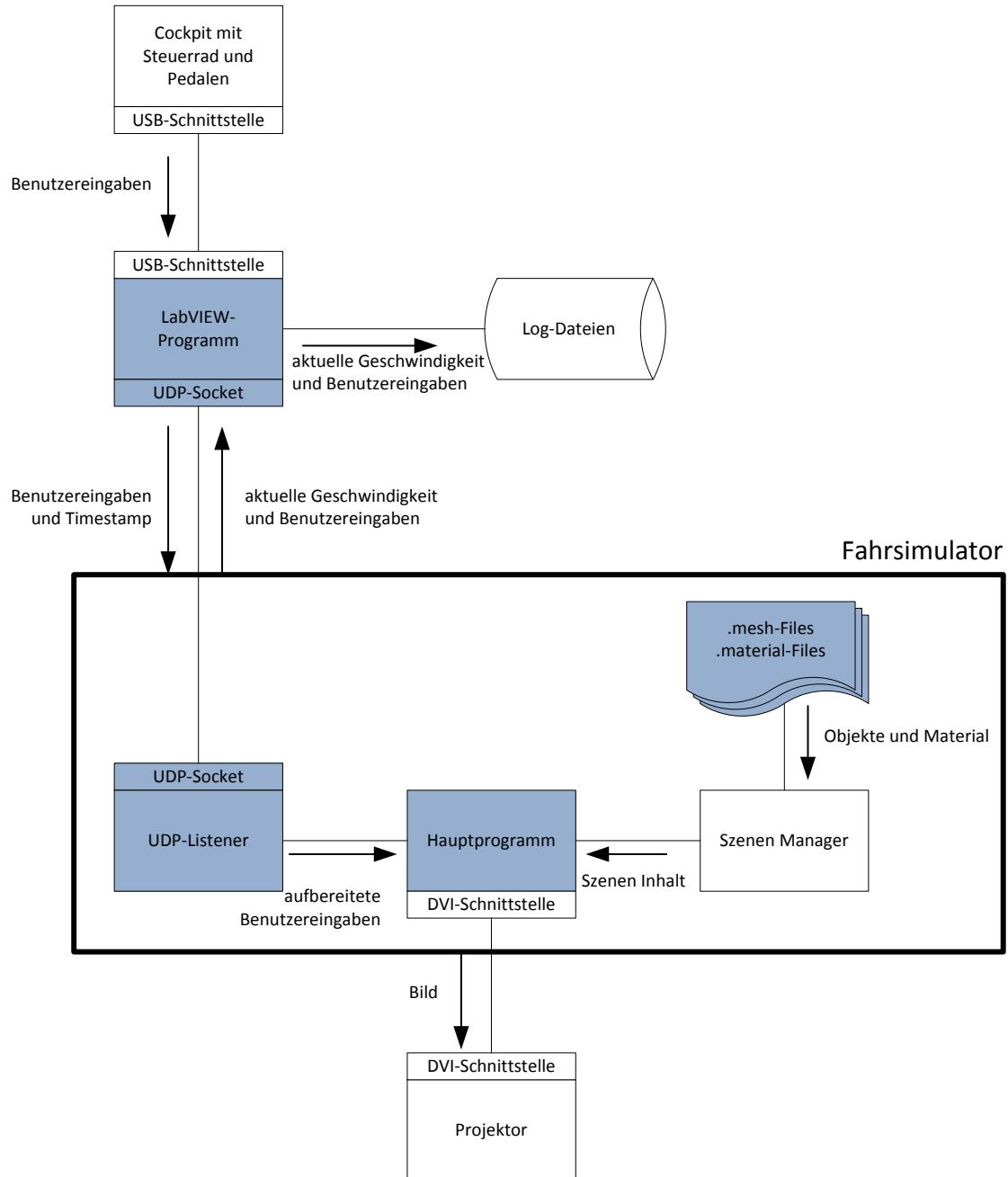


Abbildung 2: Systembeschreibung

Die blau markierten Komponenten in Abbildung 2 werden im Rahmen dieser Projektarbeit entwickelt. Alle übrigen sind bereits vorbestehend.  
 Benutzereingaben, die im Cockpit gemacht werden, werden von einem LabVIEW-Programm eingelesen. Dieses benötigt einen UDP-Socket, über den verschiedene Eingaben an die

Simulationssoftware weitergeleitet werden. Es handelt sich hierbei um Werte, die das Drehen des Steuerrads und den Druck auf Gas- oder Bremspedal quantifizieren. Zusätzlich wird ein zweiter UDP-Socket für das Empfangen diverser Informationen, die von unserem Programm gesendet werden, verwendet. Die empfangenen Daten werden vom LabVIEW-Programm in ein Log-File geschrieben.

Weiter muss in C++ ein UDP-Socket mit entsprechendem UDP-Listener implementieren werden, um die Benutzereingaben zu empfangen. Der UDP-Listener wird gleichzeitig dazu verwendet, die Geschwindigkeit des Fahrzeugs sowie Timestamps und weitere Daten an das LabVIEW-Programm zurück zu schicken. Damit können die Daten gespeichert und später ausgewertet werden.

Diese Aufteilung durch eine Netzwerkschnittstelle ermöglicht es, das System, wenn notwendig, zu dezentralisieren. Einfachheitshalber wird der UDP-Listener erst in einem Video-Beispiel implementiert und getestet (Siehe Anhang B). Nachfolgend wird er in das Programm des Fahrsimulators integriert.

Sind die Daten vom UDP-Listener empfangen und aufbereitet, werden sie im Hauptprogramm weiter verwendet. Während die Position des Steuerrades, des Gas- und Bremspedals vom UDP-Listener permanent an das Hauptprogramm übertragen werden, wertet dieses die Positionen aus und veranlasst die entsprechenden Aktionen in der geladenen Szene. Die Szene selbst wird von einem Szenen-Manager geladen. Die berechnete Szene wird schlussendlich in einem Fenster des Hauptprogramms angezeigt und über eine DVI-Schnittstelle an den Projektor übertragen.

## 2.3. Anforderungen

### 2.3.1. Funktionale Anforderungen

- Der Fahrsimulator ermöglicht es dem Probanden, vom Cockpit aus das Fahrzeug zu steuern. Der Proband blickt durch die Frontscheibe des Fahrzeugs und fährt auf der Strasse.
- Die aktuelle Geschwindigkeit des gesteuerten Fahrzeuges soll für den Probanden ersichtlich sein.
- Es sollen dem Fahrsimulator zwei Szenen zur Verfügung stehen. Die eine Szene stellt eine Stadt dar, die andere eine Berglandschaft mit Tunnels.
- Die Manipulationen des Benutzers und wichtige Parameter, wie z.B. Geschwindigkeit, werden in einer Datei fortlaufend abgespeichert.
- Alle ein- und ausgehenden Parameter des Systems werden in LabVIEW grafisch dargestellt.

### 2.3.2. Nicht funktionale Anforderungen

- Das System ist robust. Es muss auch bei Fehleingaben weiter laufen.
- Das Starten des Fahrsimulators wird einfach gehalten.
- Das Fahrzeughandling ist realitätsnah.

- Der Fahrsimulator bietet dem Probanden eine realistische Fahrsimulation inklusive Straßen, Gebäude und Verkehrsschilder.
- Die Reaktionszeit des Systems ist möglichst kurz. Die Verzögerungen sind mess- und kalkulierbar.
- Das System ist modular aufgebaut, um später einfach erweitert werden zu können.
- Das System funktioniert auf der existierender Hardware, und läuft auch wenn Teile des Fahrsimulators ausgetauscht werden.

### 3. Realisierung

Wie bereits in der Systembeschreibung erwähnt, wird der Fahrsimulator in zwei Hauptkomponenten unterteilt: den UDP-Listener und das Hauptprogramm. Da die Verbindung zwischen dem Cockpit und dem Fahrsimulator mit einer Netzwerkschnittstelle realisiert wird, ist es möglich, die Anbindung an das Cockpit und den Fahrsimulator physisch auf zwei unterschiedlichen Rechnern zu betreiben.

Dies kann notwendig werden, wenn mit fortschreitendem Ausbau der 3D-Umgebung die benötigte Rechenleistung ansteigt. Folgen ungenügender Leistung sind unregelmäßige Bewegungen (Lags) in der virtuellen Umgebung oder sogar der Absturz des gesamten Programms.

#### 3.1. LabVIEW-Programm

Eine LabVIEW-Umgebung existiert bereits auf dem Rechner, an den das Cockpit angeschlossen ist. Ein LabVIEW-Programm liest bereits die Eingaben im Cockpit ein. Da dieses Programm jedoch nicht genau den Anforderungen entspricht, wird es neu realisiert. Die Eingaben werden über einen UDP-Socket alle 10 ms gesendet. Zusätzlich wird ein zweiter UDP-Socket eingerichtet, um Packete, die vom Fahrsimulator zurück gesendet werden, zu empfangen. Die Daten der empfangenen Pakete werden von LabVIEW verarbeitet, dargestellt und in ein Log-File geschrieben.

Das detaillierte LabVIEW Programm ist im Anhang G.2 zu finden. Die LabVIEW-Schnittstelle ist im Anhang B.3.1 genauer ausgeführt.

#### 3.2. UDP-Listener

In einem eigenen Thread werden UDP-Pakete kontinuierlich gelesen und verarbeitet. Die Informationen werden zwischengespeichert, so dass sie bei jedem Renderdurchlauf zur Verfügung stehen. Dank der Häufigkeit, mit der Pakete versendet werden, stellt das Verlieren eines Pakets kein Problem dar, wodurch UDP sich durch den Geschwindigkeitsvorteil gegenüber TCP besser eignet.

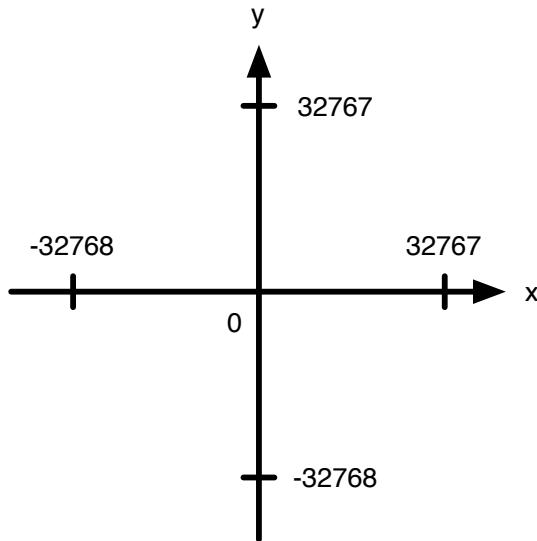


Abbildung 3: Koordinatensystem der Joystickeingabe

Die von LabVIEW empfangenen Daten sind in erster Linie Parameter, die den Ausschlag der Pedalen und den Einschlagwinkel des Steuerrades repräsentieren. Die beiden Pedale werden in einem Koordinatensystem wie in Abbildung 3 auf der y-Achse im Bereich von -32768 bis 32767 abgebildet. Dabei heisst ein negativer y-Wert, dass das Gaspedal gedrückt wird, ein positiver Wert steht für das Drücken der Bremse. Der x-Wert im Koordinatensystem quantifiziert den Einschlagswinkel des Steuerrades. Hier steht -32768 für das vollständig nach links eingeschlagene Steuerrad, 32767 für eine Drehung bis zum rechten Anschlag. Sollte es zu einem späteren Zeitpunkt nötig sein, Tastendrücke am Steuerrad zu übermitteln, kann das UDP-Paket einfach erweitert werden.

Nebst den Eingaben durch Steuerrad und Pedale wird von LabVIEW ein Timestamp gesendet. Der Timestamp wird von UDP-Listener empfangen und nach dem Abspeichern der Parameter wieder zurückgeschickt. Somit kann festgestellt werden, wieviel Verzögerung durch Kommunikation und Verarbeitung der Daten verursacht wird.

### 3.3. Virtual Reality

Eine erste Idee, um eine virtuelle Realität zu erzeugen, war der Einsatz von Google Street View oder Google Maps. Google Maps bietet die Möglichkeit, da die Satellitenbilder von der ganzen Erde gemacht wurden, überall auf der Welt zu fahren. Jedoch ist die Auflösung der Bilder nicht immer hoch genug, um damit eine Umgebung für den Fahrer generieren zu können. Erschwerend kommt hinzu, dass die Aufnahmen nur von oben gemacht wurden und somit keine Seitenansicht einer steilen Felswand oder eines Hauses zur Verfügung steht. Auch werden, da die Fotos aus der Vogelperspektive gemacht wurden, verschiedene Elemente, die sich über der Strasse befinden, zweidimensional angezeigt, wie zum Beispiel Bäume, Brücken oder Autos. Dies bedeutet, dass keine klare Strasse sichtbar ist. Somit würde der Einsatz von Google Maps zu keiner zufriedenstellenden Lösung führen.

Eine bessere Auflösung und Darstellung der Strassen bietet Google Street View. Hier wurde mit einem Wagen, der eine 360 Grad Kamera auf dem Dach trägt, den Strassen entlang gefahren und ca. alle 20 Meter ein Foto gemacht. An jedem Punkt, an dem ein Foto gemacht wurde, kann der Benutzer nun in alle Richtungen blicken. Es werden sogar grosse Gegenstände wie Wände oder Hausmauern erkannt und beim Drehen der Ansicht interpoliert. Diese Interpolation ist jedoch nicht immer korrekt und führt teilweise zu merkwürdigen Anzeigen. Zudem gibt es zwischen zwei Bildern einen zu grossen Unterschied, als dass eine fliessende Bewegung simuliert werden könnte. Eine Interpolation zweier Bilder ist denkbar, kostet aber viel Rechenleistung und liefert dennoch ein unbefriedigendes Resultat. Durch die Interpolation wird die Lücke zwischen den beiden Bildern zwar aufgefüllt, der Übergang ist aber derart verschwommen, dass keine Objekte erkannt werden können.



Abbildung 4: Verschwommene Interpolation von Google Street View Bildern

Zudem wurden meist nur auf einer Fahrspur Fotos gemacht. Wenn man nun in die andere Richtung fahren möchte, sieht es so, aus als würde man auf der falschen Strassenseite fahren. Somit kommen einem die anderen Verkehrsteilnehmer, die sich ebenfalls auf den Fotos befinden, entgegen. Auch dieser Lösungsansatz muss deshalb verworfen werden. Für den Fahrsimulator wird also eine eigene virtuelle dreidimensionale Umgebung benötigt, in der man sich frei bewegen kann.

Da in unserem Team bereits Kenntnisse über die Grafikengine OGRE<sup>(3)</sup> vorhanden sind, entscheiden wir und für ihren Einsatz in dieser Projektarbeit. Genaueres über OGRE kann im Anhang C nachgelesen werden.

Die virtuelle Umgebung, fortan auch Szene genannt, setzt sich aus verschiedenen Objekten zusammen. Die Struktur eines solchen Objektes wird durch ein so genanntes mesh-File beschrieben. In einer Szene kann das gleiche Objekt mehrmals an verschiedenen Positionen vorkommen. Das Objekt kann durch eine Skalierung einmal gross und einmal klein erscheinen. Oder man variiert sie durch Rotation. Zusätzlich wird mindestens ein

---

<sup>(3)</sup>Object-Oriented Graphics Rendering Engine

material-File benötigt, in dem sämtliche Texturen und Materialien, die von Objekten der Szene verwendet werden, definiert sind.

```
material city_ground
{
    technique
    {
        pass
        {
            diffuse 0.0 0.0 0.0 1.00
            ambient 0.75 0.75 0.75 1.00
        }
    }
}
```

Listing 1: Auszug aus dem material-File

Nach dem Keyword *material* wird der Name des Materials angegeben. Als *technique*-Block wird hier nur einer angegeben. Es können mehrere Blöcke angegeben werden, welche optional auch benannt werden können, um zu erkennen wofür die einzelnen Blöcke sind. Ausgeführt wird jedoch immer nur einer der Blöcke, wobei der erste Block der bevorzugte und der unterste Block die letzte Fallbaclklösung ist. Im *pass*-Block werden die Reflexionseigenschaften und die Farbe des Materials angegeben. Es können mehrere *pass*-Blöcke angegeben werden, die jeweils hintereinander ausgeführt werden und sich überlagern. Das *ambiant* als ambientes Licht bestimmt grundsätzlich die Farbe des Materials. Die Zahlen repräsentieren die RGB-Werte und können zwischen 0 und 1 liegen. Die Farbe des Materials in diesem Beispiel ist ein helles Grau. Die vierte Zahl repräsentiert den Alphawert der Farbe (0: unsichtbar, 1: vollständig sichtbar). Weiter zeigt das Listing 2 die definition einer Textur in einem material-File.

```

material city_street_h
{
    technique
    {
        pass
        {
            diffuse 0.80 0.80 0.80 1.00

            texture_unit
            {
                texture street_h.png 2d 4
                filtering anisotropic
            }
        }
    }
}

```

Listing 2: Beispiel aus dem material-File zur Beschreibung einer Textur

Für die Textur wird in diesem Beispiel keine ambiente Farbe definiert. Der diffuse Anteil bestimmt, wie stark das Material das Licht reflektiert. Da die Textur des Materials gut sichtbar sein soll, fällt dieser Wert relativ hoch aus. Im Block *texture\_unit* wird die Textur angegeben und konfiguriert. Die Grafik, die als Textur verwendet wird, wird nach dem Keyword *texture* angegeben. Das erste Argument nach der Grafikdatei, das *2d* gibt an, dass es sich hierbei um eine zweidimensionale Textur handelt. Die vier als drittes Argument gibt die Anzahl zu generierenden Mipmaps<sup>(4)</sup> an. Zusätzlich kann ein Filter für die Textur mit dem Keyword *filtering* angegeben werden. Der anisotropische Filter wertet die Qualität der Textur in der Distanz unter Berücksichtigung des Betrachtungswinkels auf.<sup>(5)</sup>

Es werden zwei unterschiedliche Szenen für den Fahrsimulator erstellt. Die eine Szene zeigt eine Stadt mit Häusern, Kreuzungen und Verkehrsschildern, die andere stellt eine Berglandschaft mit mehreren Tunnels dar. Die Szenen werden mit Cinema4D<sup>(6)</sup> erstellt.

<sup>(4)</sup>Gefilterte und verkleinerte Version einer Textur, um dem Aliasing-Effekt entgegenzuwirken

<sup>(5)</sup>Quelle: [http://en.wikipedia.org/wiki/Anisotropic\\_filtering](http://en.wikipedia.org/wiki/Anisotropic_filtering), Abruf: 18.12.2011

<sup>(6)</sup>Kommerzielles 3D-Modellierungstool vom Softwarehersteller MAXON

### 3.3.1. Erstellen der Stadtszene

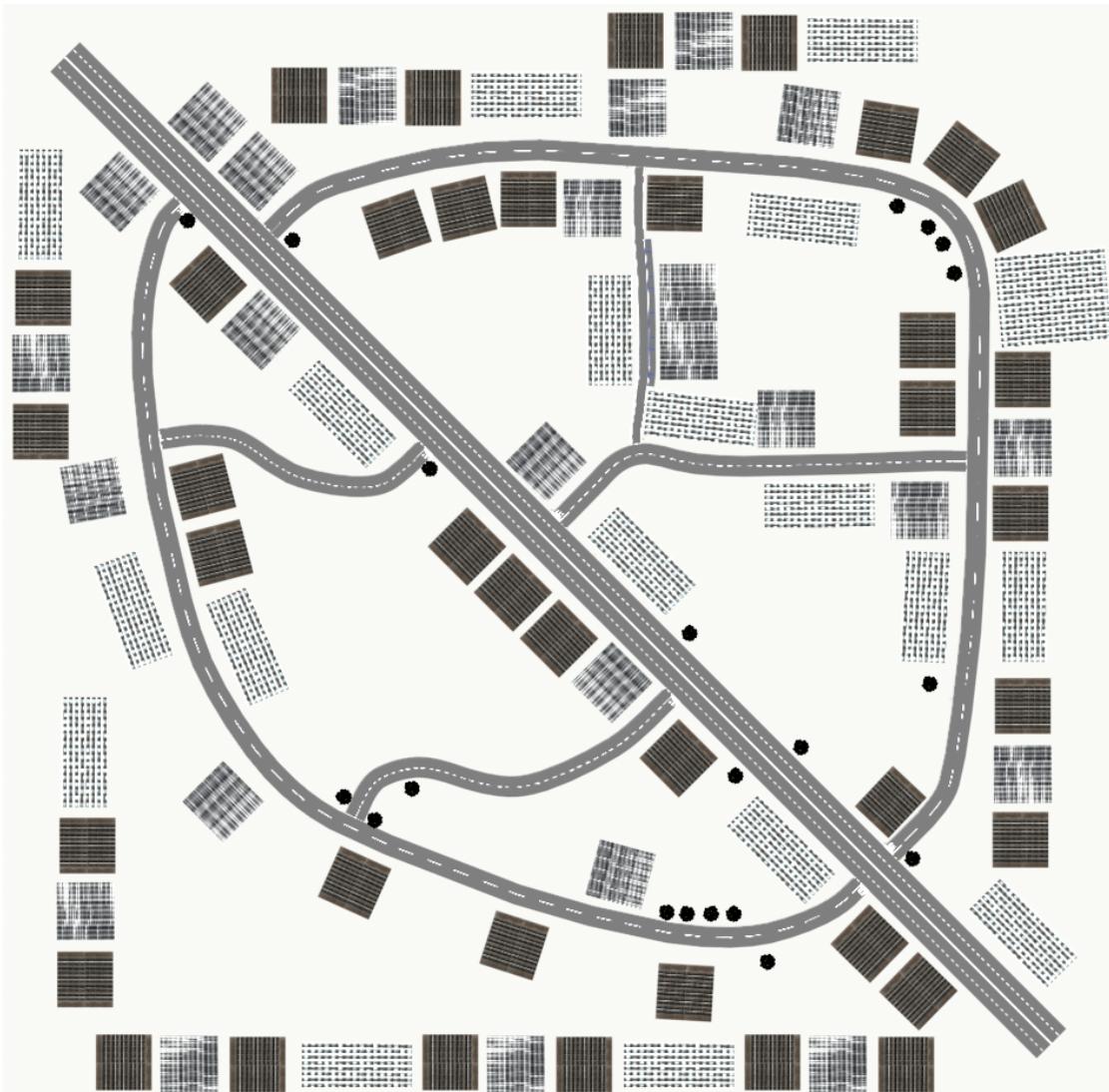


Abbildung 5: Karte der Stadtszene

Um die erstellte Stadtszene gut dokumentieren zu können, zeigt Abbildung 5 eine Übersichtskarte. Die grosse vierspurige Strasse, die sich von oben links nach unten rechts durch die gesamte Szene erstreckt, ist eine rechteckige Ebene. Die Ebene ist mit einer Textur versehen, die die vier Spuren mit einer Sicherheitslinie in der Mitte darstellt. Die Länge der Textur entspricht jedoch nicht der Länge der Strasse. Wie in Abbildung 6 zu sehen, enthält sie nur ein kurzes Stück der Strasse. Die Textur wird auf der Ebene immer wiederholt. Diese Technik wird bei allen Strassen angewendet.

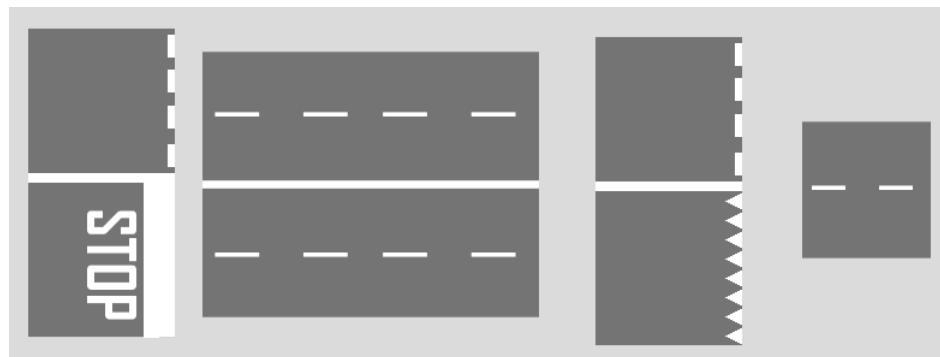


Abbildung 6: Beispiele von Strassentexturen

Die Straßen mit Kurven werden mit Splines<sup>(7)</sup> beschrieben. Eine Linie wird der Spline entlanggeführt um die Breite einer Straße zu definieren. Die Textur der Straße wird auf dieses, so zusammengefügte, Objekt gelegt. Der Abschluss solcher geschwungenen Straßen bilden wiederum rechteckige Ebenen. Diese erhalten Texturen mit Asphaltsymbolen, wie zum Beispiel einem *Stop* oder *kein Vortritt*, wie in Abbildung 6 gezeigt. Zusätzlich zu den Symbolen auf der Straße, werden noch Verkehrsschilderobjekte in die Szene geladen.



Abbildung 7: Beispiele eines Verkehrsssignals

Die Häuser der Stadt sind einfache Blöcke, die mit dem Bild einer Hausfassade texturiert werden. Insgesamt sind drei Gebäudetypen, wie in Abbildung 8 dargestellt, vorhanden. Diese drei Typen werden vervielfacht und in der Szene verteilt.

---

<sup>(7)</sup>Eine Linie die durch Interpolation von einzelnen Punkten definiert ist

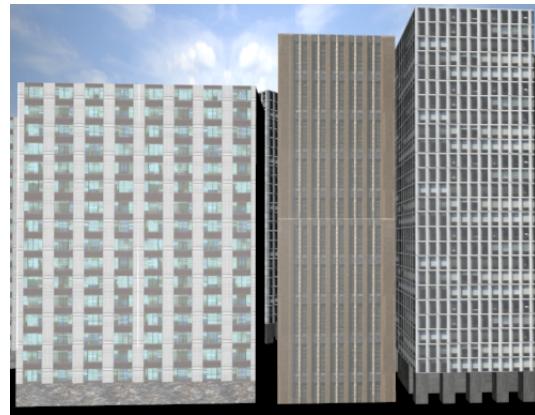


Abbildung 8: Drei verschiedene Gebäudetypen

Als nettes Extra wird das Hauptgebäude der ETH Zürich in die Szene eingefügt. Das Gebäude-Modell stammt von Google 3D Warehouse und wird mit Google SketchUp<sup>(8)</sup> bearbeitet. Dabei werden die Oberflächennormalen neu berechnet und die Bodenplatte, auf der das Gebäude steht, entfernt. Die Texturen werden manuell in das material-File übernommen. Nun wird das ETH-Modell in die Szene geladen, skaliert und an der richtigen Stelle platziert.

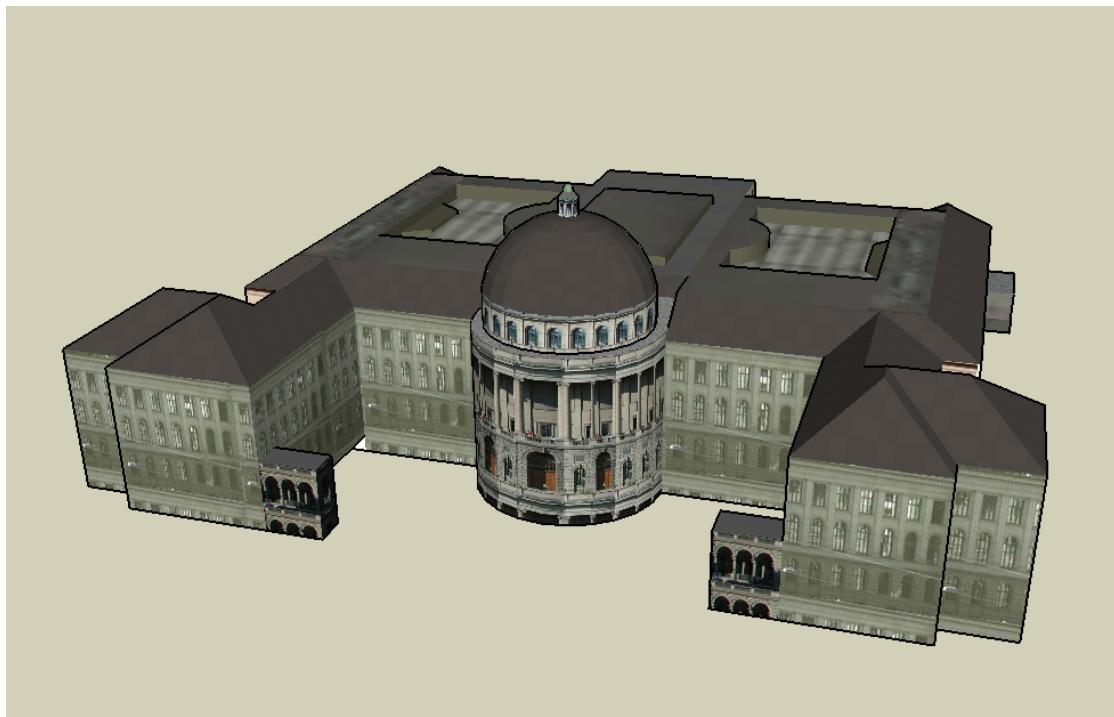


Abbildung 9: Das bearbeitete Modell der ETH Zürich

---

<sup>(8)</sup>Software zum Erstellen von Objekten. Sie wurde ursprünglich zur Modellierung von Gebäuden für Google Earth entwickelt, unterstützt aber auch andere bekannte 3D-Formate

Um der Szene ein bisschen Farbe zu verleihen sind noch Bäume, wie in Abbildung 10, in der Szene vorhanden. Ein Baum besteht aus einem Stamm mit einer Holztextur und einer Krone. Die Krone ist eine Kugel mit zufällig generierten Ausbeulungen. Nachdem eine Blättertextur darauf gelegt wurde, ist die Illusion eines Baumes vollständig.



Abbildung 10: Ein Baum-Modell

### 3.3.2. Erstellung der Berglandschaft mit Tunnels

Zur Generierung der Berglandschaft gibt es in Cinema4D ein eigens dafür vorgesehenes Objekt, das so genannte Landscape-Objekt. Durch die Konfiguration dieses Objekts kann die Gesamthöhe, sowie die Anzahl der Erhöhungen und Vertiefungen der Landschaft festgelegt werden. Mit einer Felstextur sieht dieses Objekt aus wie ein Berg.

Die Strasse in der Berglandschaft wird ähnlich der Strasse, in der Stadtszene gemacht. Zusätzlich wird der Spline ein Quader entlanggezogen, welcher mit einer boolschen Operation vom Berg subtrahiert wird (Siehe Abbildung 11). Dadurch entsteht eine Röhre im Berg, welche aussieht wie ein Tunnel. Die Innenseite wird mit einer Betonplattentextur versehen.

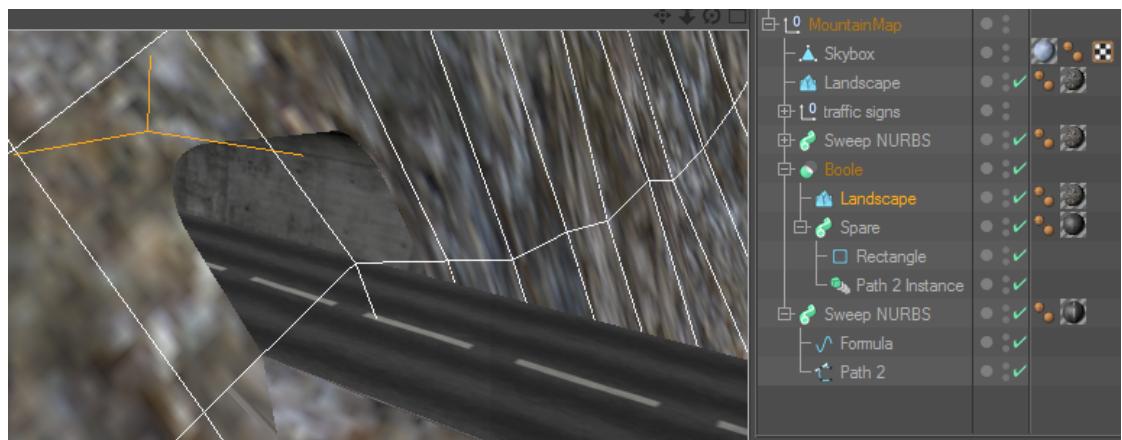


Abbildung 11: Ausschneiden des Tunnels mit Cinema4D

### 3.3.3. Fahrzeug

Zur Darstellung eines Autos wurde aus Google 3D Warehouse ein Mini Cooper heruntergeladen und in nach OGRE-XML<sup>(9)</sup> konvertiert. Wie in Abbildung 12 zu sehen, wird dieser aus der Perspektive einer dritten Person betrachtet. Beim Rückwärtsfahren wird die Kamera um das Fahrzeug gedreht, so dass man nach hinten sehen kann. Mit der Taste *V* auf der Tastatur kann die Ansicht in das Innere des Fahrzeugs gewechselt werden. Dort wird die Kamera dort positioniert, wo der Kopf des Fahrers in etwa sein sollte. In einem zweiten Schritt wird eine eigene Fahrerkabine erstellt. Dies war vor allem deshalb notwendig, um eine Geschwindigkeitsanzeige implementieren zu können.



Abbildung 12: Der Mini Cooper aus der 3rd-Person-Ansicht

---

<sup>(9)</sup>Von der OGRE-Community entworfenes Format zur Speicherung von 3D-Modellen. Es existieren Plugins für diverse 3D-Modellierungstools, mit denen Objekte in diesem Format gespeichert werden können

### 3.3.4. Fahrerkabine

In der Innenansicht wird die Kamera so positioniert, dass der Benutzer die Szene aus dem Blickwinkel des Fahrzeugführers wahrnimmt. Dazu wird das Fahrzeug ausgeblendet und ein Modell wird sichtbar. Dieses besteht aus drei getrennten Objekten:

- dem Cockpit, bestehend aus dem Armaturenbrett, der linken A-Säule<sup>(10)</sup> und dem Dach
- dem Tachozeiger, welcher relativ zum Cockpit positioniert und entsprechend der aktuellen Geschwindigkeit gedreht wird.
- dem Steuerrad, welches auch relativ zum Cockpit positioniert und entsprechend dem Lenkeinschlag rotiert wird.



Abbildung 13: Innenansicht des Mini mit Geschwindigkeitsanzeige

Um die Geschwindigkeit des Fahrzeugs herauszufinden, wird die Skalierung aller Objekte so gewählt, dass in Cinema4D mit Werten gearbeitet werden konnte, die den tatsächlichen Größen entsprechen. So sind die Strassenspuren drei Meter breit modelliert. Um

---

<sup>(10)</sup>Die vordersten beiden Säulen beim Auto, die das Dach tragen

die Geschwindigkeit zu ermitteln, wurde in Cinema4D eine Distanz abgemessen und diese mit verschiedenen konstanten Geschwindigkeiten abgefahren und die Zeit gemessen. Die interne Geschwindigkeit im Programm wird mit einem Faktor in die tatsächliche Geschwindigkeit umgerechnet. Diese kann dann auf dem Tachometer angezeigt werden.

Distanz (m)	Geschwindigkeit im Programm	Zeitmessung (s)						Reale Geschwindigkeit (km/h)	Skalierung
		1	2	3	4	5	Durchschnitt		
500	100	24.7	25	24.9	25	24.9	24.9	72.28915663	<b>0.722891566</b>
500	175	14.2	14.2	14.2	14.2	14.2	14.2	126.7605634	<b>0.724346076</b>

Abbildung 14: Berechnung der tatsächlichen Geschwindigkeit

### 3.4. Hauptprogramm

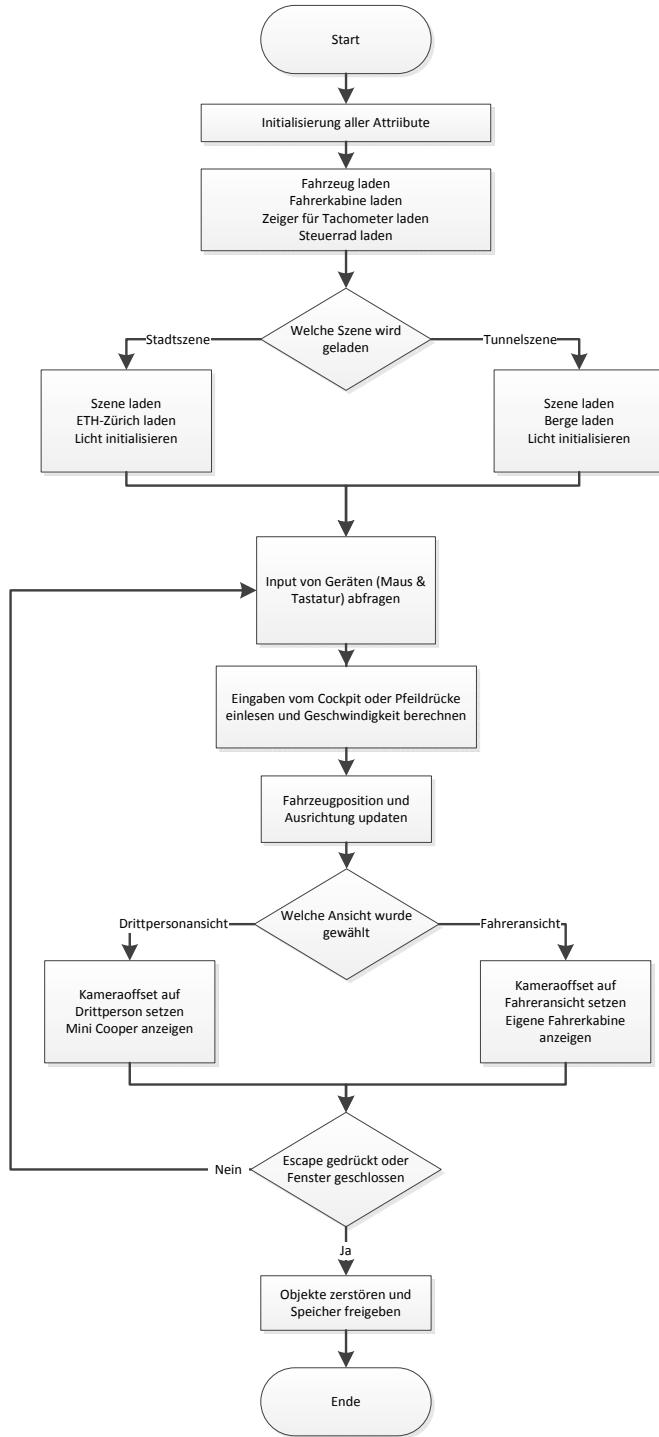


Abbildung 15: Ablauf des Hauptprogramms

### 3.4.1. Initialisierung

Beim Start der Applikation werden die Konfigurationsfiles (siehe Kapitel 3.4.2) geladen. Aus den darin enthaltenen Informationen wird das Grundgerüst von OGRE erstellt. Dazu gehören folgende Objekte:

#### **rootNode**

Der rootNode der 3D-Szene. Alle Objekte sind diesem angehängt.

#### **renderWindow**

Das RenderWindow ist das sichtbare Fenster der Applikation. In ihm wird der gesamte grafische Inhalt dargestellt.

#### **camera**

Wie der Name vermuten lässt, ist dies die Kamera, durch welche der Benutzer die Szene beobachtet.

#### **viewPort**

Der ViewPort ist mit dem Kameraobjekt verknüpft und zeigt die Bildprojektion der 3D-Szene an.

```
// create root node
#ifndef _DEBUG
rootNode = new Ogre::Root("Resources/plugins_d.cfg",
    "Resources/graphics_d.cfg", "log_d.txt");
#else
rootNode = new Ogre::Root("Resources/plugins.cfg",
    "Resources/graphics.cfg", "log.txt");
#endif

// load config file
Ogre::ConfigFile configFile;
#ifndef _DEBUG
configFile.load("Resources/resources_d.cfg");
#else
configFile.load("Resources/resources.cfg");
#endif

// load resource files
Ogre::ConfigFile::SectionIterator it = configFile.getSectionIterator();
while(it.hasMoreElements())
{
    Ogre::String sectionName = it.peekNextKey();
    Ogre::ConfigFile::SettingsMultiMap* settings = it.getNext();
    Ogre::ConfigFile::SettingsMultiMap::iterator i;
    for(i = settings->begin(); i != settings->end(); ++i)
    {
        Ogre::ResourceGroupManager::getSingleton().
            addResourceLocation(i->second, i->first, sectionName);
    }
}

// try to initialize window with settings from config file
if(!(rootNode->restoreConfig()))
{
```

```

throw Ogre::Exception(Ogre::Exception::ERR_INVALID_STATE, "Could
    not read graphics configuration", "MainApplication::go");
}

renderWindow = rootNode->initialise(true, "Driving Simulator V1");

// initialize resources
Ogre::TextureManager::getSingleton().setDefaultNumMipmaps(5);
Ogre::ResourceGroupManager::getSingleton().
    initialiseAllResourceGroups();

// create scene manager
sceneManager = rootNode->createSceneManager("DefaultSceneManager");

// create camera
camera = sceneManager->createCamera("PlayerCam");
camera->setAutoAspectRatio(true);
camera->setNearClipDistance(1);

// add viewport
Ogre::Viewport* viewPort = renderWindow->addViewport(camera);
viewPort->setBackgroundColour(Ogre::ColourValue(0, 0, 0));

```

Listing 3: Laden der Konfigurationsdateien

### 3.4.2. Konfigurationsfiles

Die Applikation verwendet folgende Dateien zur Konfiguration beim Start:

#### **plugins.cfg**

OGRE lädt einige Komponenten zur Laufzeit, die zur Ausführung des Programms benötigt werden. Welche Komponenten geladen werden, wird im File plugins.cfg definiert. Inhalt siehe Anhang G.1.

#### **graphics.cfg**

In diesem File sind die Grafikeinstellungen definiert. Inhalt siehe Anhang G.1.

#### **resources.cfg**

Hier werden alle Pfade angegeben, in welchen sich benötigte Ressourcen (z.B. mesh-Files, Bilder, etc.) befinden. Inhalt siehe Angang G.1.

### 3.4.3. Laden des Autos

```

void DrivingSimulatorV1::createCar()
{
    // load car
    Ogre::Entity* car = sceneManager->createEntity("MiniCooper.mesh");
    carNode = sceneManager->getRootSceneNode()->createChildSceneNode();
    carNode->attachObject(car);
    carNode->scale(4, 4, 4);

    // load Cockpit
}

```

```

Ogre::Entity* cockpit =
    sceneManager->createEntity ("MiniCockpit.mesh");
cockpitNode =
    sceneManager->getRootSceneNode()->createChildSceneNode();
cockpitNode->attachObject (cockpit);
cockpitNode->scale (0.05, 0.05, 0.05);

Ogre::Entity* pointer =
    sceneManager->createEntity ("MiniCockpitPointer.mesh");
pointerNode =
    sceneManager->getRootSceneNode()->createChildSceneNode();
pointerNode->attachObject (pointer);
pointerNode->scale (0.02, 0.02, 0.02);

Ogre::Entity* steeringWheel =
    sceneManager->createEntity ("MiniCockpitSteeringWheel.mesh");
steeringWheelNode =
    sceneManager->getRootSceneNode()->createChildSceneNode();
steeringWheelNode->attachObject (steeringWheel);
steeringWheelNode->scale (0.03, 0.03, 0.03);

// setup camera
camera->setFOVy (Ogre::Degree(70));
}

```

Listing 4: Laden des Autos

Hier werden die drei Modelle *MiniCooper.mesh*, *MiniCockpitPointer.mesh* und *MiniSteeringWheel.mesh* durch den Szenenmanager geladen und jeweils einem eigenen Node<sup>(11)</sup> angehängt und entsprechend skaliert, so dass sie in die Szene passen.

### 3.4.4. Laden des Szene

```

void DrivingSimulatorV1 :: createScene1 () // city
{
    // create world node
    Ogre::SceneNode* worldNode =
        sceneManager->getRootSceneNode()->createChildSceneNode();
    Ogre::Entity* cityWorld =
        sceneManager->createEntity ("CityWorld.mesh");
    worldNode->scale (0.05, 0.05, 0.05);
    worldNode->attachObject (cityWorld);

    // create ETH Node
    Ogre::SceneNode* ethNode =
        sceneManager->getRootSceneNode()->createChildSceneNode();
    Ogre::Entity* eth = sceneManager->createEntity ("ETH.mesh");
    ethNode->attachObject (eth);
    ethNode->scale (1.3, 1.3, 1.3);
    ethNode->setPosition (428, 0, 235);
    ethNode->yaw (Ogre::Degree(210));

    // create ambient light
}

```

<sup>(11)</sup>Element des Szenengraphen, das eine Transformation und Modelle bzw. weitere Nodes enthält

```

sceneManager->setAmbientLight (Ogre::ColourValue (0.7, 0.7, 0.7));

// create sun light
Ogre::Light* sunLight = sceneManager->createLight ();
sunLight->setType (Ogre::Light::LT_DIRECTIONAL);
sunLight->setDirection (Ogre::Vector3 (-0.5, -0.5, 0.5));
sunLight->setDiffuseColour (Ogre::ColourValue (1, 1, 1));
sunLight->setSpecularColour (Ogre::ColourValue (0.7, 0.7, 0.7));

// set car to initial position and orientation
carNode->setPosition (584, 0, 121);
carNode->setOrientation (Ogre::Quaternion (Ogre::Degree (-4.5),
    Ogre::Vector3::UNIT_Y));
}

```

Listing 5: Laden der Szene

Ähnlich zur Funktion *createCar* (Abschnitt 4) werden hier zunächst Objekte geladen. Diesmal die beiden Meshes *CityWorld.mesh* und *ETH.mesh*. Auch sie werden jeweils eigenen Nodes angehängt und entsprechend positioniert, ausgerichtet und skaliert. Weiter wird in dieser Funktion die Belichtung der Szene definiert. Einmal wird mit *setAmbientLight* das Umgebungslicht gesetzt, danach wird eine neue direktionale Lichtquelle erstellt, die in unserem Programm das Sonnenlicht liefert. Zuletzt wird das Auto an die korrekte Startposition in der Szene gesetzt und ausgerichtet.

Das Laden der Tunnelszene geschieht in der Funktion *createScene2*. Details dazu im Listing *DrivingSimulatorV1.cpp* auf der beiliegenden CD.

### 3.4.5. Callback-Methode

Die so genannte Callback-Methode<sup>(12)</sup> wird durch einen internen Timer des OGRE-Frameworks aufgerufen. Dies geschieht je nach Vsync<sup>(13)</sup>-Konfiguration entweder mit der Bildwiederholungsrate der Bildschirms z.B. 60 mal pro Sekunde oder so oft wie möglich. Über die Variable *timeSinceLastFrame* ist genau bestimmt, wie lange der letzte Renderdurchlauf zurück liegt. Die Aufgabe dieser Methode ist alles das zu tun, was vor dem Zeichnen eines neuen Bildes getan werden muss. Dazu gehört folgendes:

### Tastatur- und Mauseingaben abfragen

```

// capture input devices
if (inputManager)
{
    keyboard->capture ();
    mouse->capture ();
}

```

Listing 6: Abfragen der Eingabegeräte

<sup>(12)</sup>Methode, die von einem externen Programmteil aufgerufen wird

<sup>(13)</sup>Vertical Sync - bedeutet, dass mit dem Neuberechnen eines Bildes gewartet wird, bis der gesamte Bildschirminhalt gezeichnet wurde. Dadurch wird die Verzerrung des Bildes durch schnelle Bewegungen verhindert

### Geschwindigkeit reduzieren

```
// decrease speed (air resistance, etc...)
if(speed > 0)
    speed *= Ogre::Math::Pow(0.92, evt.timeSinceLastFrame);
else
    speed *= Ogre::Math::Pow(0.7, evt.timeSinceLastFrame);
```

Listing 7: Geschwindigkeit reduzieren

Gibt man kein Gas, so verlangsamt sich das Fahrzeug durch das Einwirken diverser Kräfte (z.B. Luft- und Rollwiderstand, Reibungskräfte innerhalb von Motor und Getriebe, etc.). Dies wird durch das Multiplizieren mit einem Wert kleiner 1 simuliert. Da nicht immer bekannt ist, wie oft diese Methode pro Sekunde ausgeführt wird, potenziert man diesen Wert mit der Anzahl vergangener Sekunden seit dem letzten Renderdurchlauf. So ist definiert, dass sich die Geschwindigkeit des Autos pro Sekunde um den Faktor 0.92 bzw. 0.7 verringert.

### Beschleunigen oder Bremsen

```
// udp input
if(UdpListener::throttle >= 0)
    speed += UdpListener::throttle * 9 * evt.timeSinceLastFrame;
else
    speed += UdpListener::throttle * 30 * evt.timeSinceLastFrame;

// keyboard input
if(keyboard->isKeyDown(OIS::KC_UP))
    speed += 9 * evt.timeSinceLastFrame;
if(keyboard->isKeyDown(OIS::KC_DOWN))
    speed -= 30 * evt.timeSinceLastFrame;
```

Listing 8: Beschleunigen oder Bremsen

Abhängig vom Ausschlag des Gas- bzw. Bremspedals wird das Fahrzeug beschleunigt oder abgebremst. Auch hier wird die *timeSinceLastFrame*-Variable verwendet um das zeitliche Verhalten genau zu definieren. Das Drücken der *Pfeil-oben*-Taste entspricht dem vollständig durchgedrückten Gaspedal. Analog dazu löst das Drücken der *Pfeil-unten*-Taste eine Vollbremsung aus.

### Fahrzeug lenken

```
// calculate steer intensity
Ogre::Real normalizedSpeed = Ogre::Math::Abs(speed / 180);
Ogre::Real steerIntensity = 100 / (100 *
    (Ogre::Math::Pow(normalizedSpeed, 2)) + 1) *
    Ogre::Math::Pow(normalizedSpeed, 1.5);

// rotate car by udp input
```

```

carNode->yaw( Ogre::Degree( UdpListener::steer * -5 *
    evt.timeSinceLastFrame * speed));
cameraRotationOffset += UdpListener::steer * 90 *
    evt.timeSinceLastFrame;

// rotate car by keyboard input
if( keyboard->isKeyDown(OIS::KC_LEFT))
    keyboardSteer += 270 * evt.timeSinceLastFrame;
if( keyboard->isKeyDown(OIS::KC_RIGHT))
    keyboardSteer -= 270 * evt.timeSinceLastFrame;

```

Listing 9: Fahrzeug lenken

Wenn das Fahrzeug in eine Richtung gelenkt werden soll, findet eine Rotation um die Y-Achse statt. Die Geschwindigkeit, mit der rotiert wird (hier *steerIntensity*), ist von der Geschwindigkeit des Fahrzeugs abhängig und wird hier von einer nicht-linearen Funktion beschrieben. Damit soll das Untersteuern<sup>(14)</sup> des Fahrzeugs bei hohen Geschwindigkeiten simuliert werden.

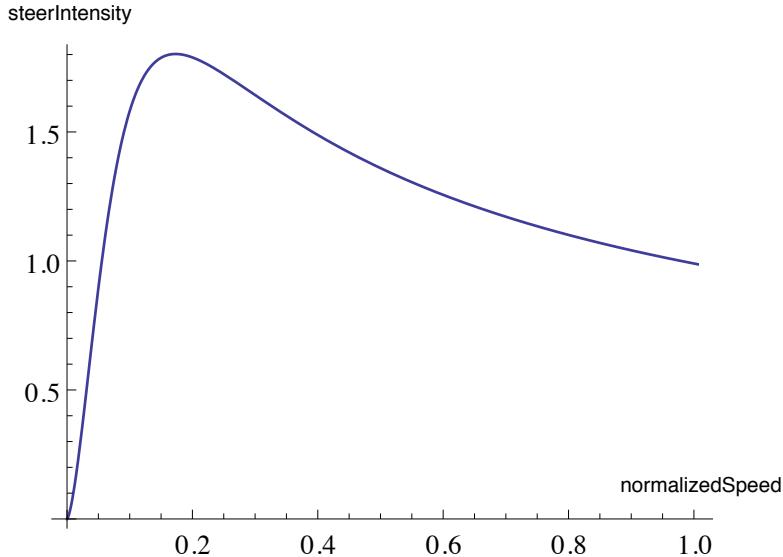


Abbildung 16: Steuerintensität in Abhängigkeit der Geschwindigkeit

### Fahrzeug bewegen

```

// update car position
Ogre::Real xMove = Ogre::Math::Sin(carNode->getOrientation().getYaw())
    * speed * evt.timeSinceLastFrame;
Ogre::Real zMove = Ogre::Math::Cos(carNode->getOrientation().getYaw())
    * speed * evt.timeSinceLastFrame;
carNode->translate(xMove, 0, zMove);

```

Listing 10: Fahrzeug bewegen

<sup>(14)</sup>Rutschen des Fahrzeugs über die Vorderachse in der Kurve

Die Koordinaten des Autos werden entsprechend seiner Ausrichtung und Geschwindigkeit neu berechnet.

### Kamera positionieren

```
// position camera
Ogre::Vector3 cameraOffset(1.3, 4.0, 0.7);
camera->setOrientation(carNode->getOrientation() *
    Ogre::Quaternion(Ogre::Degree(180), Ogre::Vector3::UNIT_Y));
camera->setPosition(carNode->getPosition() + carNode->getOrientation() *
    * cameraOffset);
```

Listing 11: Positionierung der 1st-Person-Kamera

In der 1st-Person<sup>(15)</sup>-Ansicht wird die Kamera an die Stelle, an der sich der Kopf des Fahrers befindet, verschoben.

```
if(gear != REVERSE)
    cameraRotationOffset *= Ogre::Math::Pow(0.1,
        evt.timeSinceLastFrame);
else
    cameraRotationOffset = cameraRotationOffset * Ogre::Math::Pow(0.1,
        evt.timeSinceLastFrame) + 180 * (1 - Ogre::Math::Pow(0.1,
        evt.timeSinceLastFrame));

Ogre::Radian camAngle = carNode->getOrientation().getYaw() +
    Ogre::Degree(cameraRotationOffset);
Ogre::Real camXOffset = -Ogre::Math::Sin(camAngle) * 25;
Ogre::Real camYOffset = 8;
Ogre::Real camZOffset = -Ogre::Math::Cos(camAngle) * 25;

camera->setPosition(carNode->getPosition() + Ogre::Vector3(camXOffset,
    camYOffset, camZOffset));
camera->lookAt(carNode->getPosition());

carNode->setVisible(true);
cockpitNode->setVisible(false);
pointerNode->setVisible(false);
steeringWheelNode->setVisible(false);
```

Listing 12: Positionierung der 3rd-Person-Kamera

Im 3rd-Person<sup>(16)</sup>-Modus wird die Kamera um eine bestimmte Distanz nach hinten verschoben und zum Mittelpunkt des Fahrzeugs ausgerichtet.

---

<sup>(15)</sup>Ich-Perspektive

<sup>(16)</sup>Kameraeinstellung, bei der die Kamera aus dem Blickwinkel einer dritten Person (Beobachter) filmt

### 3.4.6. Sequenzdiagramm

In folgendem Diagramm wird zusammenfassend gezeigt, wie die einzelnen Komponenten zeitlich zusammenarbeiten.

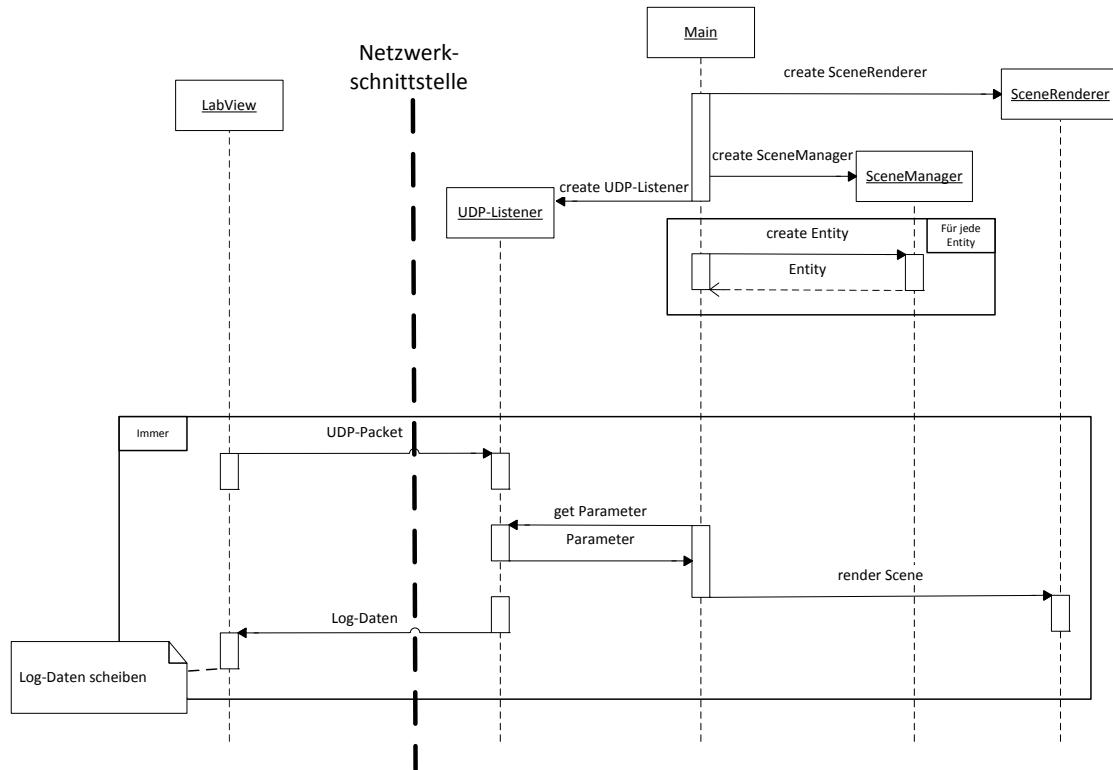


Abbildung 17: Sequenzdiagramm des Fahrersimulators

## 4. Resultate und Tests

### 4.1. Erreiches

Die Schnittstelle zwischen dem Cockpit und dem Fahrsimulator wurde vollständig in LabVIEW realisiert. Um diese zu testen wurde ein Videoplayer implementiert. Der Videoplayer reagiert auf Eingaben im Cockpit. Beim Drücken des Gaspedals wird das Video schneller, bis maximal doppelt so schnell, abgespielt. Bei einem Druck auf das Bremspedal wird das Video langsamer abgespielt. Liegt die Geschwindigkeit unter 15 km/h, wird das Video pausiert.

Im Fahrsimulator kann das Fahrzeug durch Eingaben im Cockpit bewegt werden. Es stehen zwei unterschiedliche Szenen zur Verfügung. Eine Szene stellt eine Stadt dar, in welcher mehrere Straßen, die sich kreuzen, vorhanden sind. Die Kreuzungen sind durch Asphalt-Symbole und Verkehrsschilder gekennzeichnet. Um die Stadtszene lebendiger zu machen, sind verschiedene Gebäude und Bäume eingefügt worden. Zudem wurde das ETH Zürich Gebäude in die Szene eingefügt. Dies zeigt exemplarisch, wie einfach Objekte aus dem Google 3D Warehouse eingefügt werden können. Eine zweite Szene ist ein Rundkurs in einer Berglandschaft. Auf diesem Rundkurs gibt es mehrere Tunnels, die durch die Berglandschaft führen. Die Daten beider Systeme, die des Videoplayers sowie die des Fahrsimulators, werden erfolgreich an das LabVIEW-Programm zurückgeschickt und in ein Log-File geschrieben. Diese Daten können in eine spätere Auswertung mit einbezogen werden.

Somit wurden alle Anforderungen an das Projekt erfüllt. Das Resultat betrachten wir als zufriedenstellend.

### 4.2. Zeitliches Verhalten des Systems

Es wurden für den Videoplayer und den Fahrsimulator Zeitmessungen durchgeführt. Bei beiden Tests war das System auf einer einzigen Maschine installiert. Das Einlesen des Cockpits wurde physisch nicht vom System getrennt. Es gibt also keine Verzögerung durch das Netzwerk. Trotz dieser Tatsache war das Resultat überraschend, denn die Verzögerung liegt im Normalfall unter einer Millisekunde.

### 4.3. Testfälle

Es ist im Allgemeinen schwierig eine grafische Applikation wie ein Fahrsimulator durch automatische Tests zu überprüfen. Deshalb wurden die meisten Tests manuell durchgeführt. Dies geschah jeweils mindestens jeden zweiten Montag Nachmittag während vier bis fünf Stunden. In dieser Zeit wurden alle Neuerungen, die während der Woche entwickelt wurden, auf dem Fahrsimulator installiert und eingehend getestet.

Häufige Fehlerursachen waren versäumte Initialisierungen von Variablen, welche auf verschiedenen Betriebssystemen unterschiedlich gehandhabt werden. Da der Fahrsimulator auf unterschiedlichen Betriebssystemen getestet wurde, konnten diese Fehlerquellen weitgehend eliminiert werden.

Wir, das Entwicklerteam, sind durch das häufige Testen zur Überzeugung gekommen, dass die Software den Anforderungen an Stabilität und Zuverlässigkeit gerecht wird.

## 5. Nächste Schritte

### 5.1. Offene Punkte

Trotz einem gutem Endergebnis der Arbeit, konnten einige Punkte aus Zeitgründen nicht realisiert werden. Einer dieser Punkte ist die Implementierung von interaktivem Verkehr mit anderen Autos, Passanten oder Velofahrern. Zusätzlich hätte dann auch eine Kollisionserkennung implementiert werden müssen, um Zusammenstöße mit ruhenden oder bewegten Objekten zu erkennen. Um der Illusion der realen Welt etwas näher zu kommen, wäre eine Implementation eines eigenen Shaders<sup>(17)</sup> von grossem Vorteil. Um das Starten des Fahrsimulators zu erleichtern wäre ein GUI<sup>(18)</sup> eine gute Lösung.

### 5.2. Zusätzliche Funktionen

Um den Fahrsimulator von manuell erstellten Szenen unabhängig zu machen, könnten die Szenen aus Informationen von Google Maps automatisch generiert werden. Auf Google Maps sind die Informationen über Lage, Art und Richtung der Strasse verfügbar. Ein Algorithmus könnte mit diesen Informationen Strassen für jedes beliebige Gebiet generieren.

Eine zusätzliche Funktion wäre ein erweitertes GUI, mit dem Manipulationen am Fahrzeug oder der Szene vorgenommen werden können. Zum Beispiel könnte die Leistung des Fahrzeuges Verkehrsichte verändert werden.

Eine Möglichkeit, die Illusion des Fahrens zu verbessern, wäre der Einsatz von Motorengeräuschen und anderen Soundeffekten, wie zum Beispiel quitschenden Reifen bei starkem Bremsen.

### 5.3. Ausblick auf Bachelor Arbeit 2012

Da die Zusammenarbeit mit der ETH sehr erfolgreich war und das Projekt zufriedenstellend verlief, wurde von den zuständigen Dozenten die Weiterführung der Projektarbeit als Bachelor Arbeit beschlossen. Wir begrüssen diese Entscheidung sehr und freuen uns, ein interessantes Projekt weiterführen zu dürfen.

---

<sup>(17)</sup>Programm, welches auf der GPU läuft und der Berechnung von Licht-, Schatten- und weiteren Effekten von Materialien dient

<sup>(18)</sup>Abkürzung für Graphical User Interface (deutsch: Grafische Benutzeroberfläche)

## 6. Nachwort

### 6.1. Rückblick

Die Projektarbeit Fahrsimulator konnte erfolgreich und termingerecht abgeschlossen werden. Sämtliche Anforderungen der Aufgabenstellung wurden erfüllt. Das Fahrzeug kann durch Eingaben im Cockpit navigiert werden und es stehen zwei unterschiedliche Szenen zur Verfügung. Das System läuft soweit robust.

Mehr Zeit als ursprünglich geplant, benötigten wir vor allem für die Erstellung der beiden Szenen. Obwohl wir mit Cinema4D ein sehr gutes Tool zu Hand hatten, mussten Materialien und Texturen manuell erstellt werden. Dies führte zu Fehlern, deren Behebung wiederum viel Zeit in Anspruch nahm.

Rückblickend sind wir froh, dass wir uns zu Beginn gegen einen Einsatz von Google Maps und Google Street View entschieden haben. Durch die Modellierung einer eigenen 3D Umgebung konnten so gute Resultate erzielt werden, wie sie mit den beiden anderen Tools niemals möglich gewesen wären.

### 6.2. Danksagung

An dieser Stelle möchten wir unseren beiden Projektbetreuern Prof. Dr. Peter Früh und Prof. Martin Schlup für ihre Betreuung, Unterstützung und Anregungen während der gesamten Projektarbeit danken. Ebenso möchten wir uns beim Team an der ETH Zürich, Ying-Yin Huang und Prof. Dr. Marino Menozzi für eine erfolgreiche und angenehme Zusammenarbeit bedanken. Des weiteren danken wir Daniela Zehnder und Kathrin Achtach für die Überprüfung der Rechtschreibung und Grammatik unserer Arbeit.

## A. Aufgabenstellung

### Aufgabenstellung der Projektarbeit PA11\_frup\_2

Herbstsemester 2011

**Students:** Sandro Ropelato ([ropelsan@students.zhaw.ch](mailto:ropelsan@students.zhaw.ch)), Christof Würmli ([wurmlchr@students.zhaw.ch](mailto:wurmlchr@students.zhaw.ch))

**Industriepartner:** Frau Rudy Ying-Yin Huang ([yingyinhuang@ethz.ch](mailto:yingyinhuang@ethz.ch), 044 63 22823)

ETH Zürich - MTEC - TIM - Research - Ergonomie der Informationsmedien  
Scheuchzerstrasse 7, 8092 Zürich

**Betreuer:** Dr. Peter Früh ([frup@zhaw.ch](mailto:frup@zhaw.ch)), Martin Schlup ([spma@zhaw.ch](mailto:spma@zhaw.ch))

#### Titel der Arbeit

### Fahrsimulator mit realistischer virtuellen Umgebung

#### Ausgangslage

Im Rahmen einer grösseren Studie, soll das Verhalten diverser Autofahrer unter bestimmten reproduzierbaren Bedingungen, wie verschiedene Geschwindigkeiten oder Strassenverhältnisse, untersucht werden. Dazu steht an der ETH Zürich ein Fahrsimulator zur Verfügung, für den eine interaktiv steuerbare virtuelle Umgebung entwickelt werden soll. Die benötigten Strassenszenen sollen durch "Google Street View" (Google Earth) geliefert und durch diverse in die Landschaft eingefügte Objekte wie Fahrzeuge, Fussgänger oder Signalisationsschilder ergänzt werden, z.B. mit Hilfe der Google SketchUp-Software. Die Steuerelemente bestehen aus den üblichen Bedienelementen eines PWs (Lenkrad, Gas- und Bremspedale, usw.) welche über LabVIEW den Szenenablauf in "Echtzeit" steuern sollen. Die Nahtstellen zwischen den Bedienelementen und LabVIEW sind bereits vorhanden. Mit LabVIEW sollen auch die Betriebszustände und -abläufe des Simulators, sowie die eingegebenen Steuerbefehle registriert werden.

#### Zielsetzungen

1. Erzeugen einer virtuellen Umgebung (virtual reality) für den Fahrsimulator mit
  - o interaktiv steuerbaren Strassenlandschaft basierend auf Google Street View
  - o Synthetisieren und Einfügen von diversen Objekten in die „VR-Landschaft“, z.B. mittels Google SketchUp
2. Registrieren der Betriebszustände und -abläufe des Simulators, sowie der eingegebenen Steuerbefehle mit LabVIEW.

## Beschreibung der Arbeit

- Situationsanalyse: Aufstellen der Anforderungen an die Computer-Graphik, Beschreibung der Nahtstellen und Abläufe, Auflistung der Steuersignale und Registrierdaten
- Realisierungsvorschläge, Übersicht über mögliche Verfahren
- Erarbeiten eines Lösungs- und Vorgehenskonzepts (die Reihenfolge der einzelnen Entwicklungsschritte ist frei wählbar)
- Realisierung der gewählten Lösung
- Analyse, Tests und Dokumentation
- gegebenenfalls Verbesserungen, Vorschläge

Weitere Einzelheiten werden in regelmässigen Besprechungen festgelegt.

Aufwand: entsprechend 2x6 ECTS, 300 bis 360 Mann-Stunden

## Material

Ex: Instrument, Messgerät, Steuerung, Antrieb, usw.

- Software: LabVIEW, Google Tools
- PC
- Fahrsimulator

## Bericht und weitere Dokumente

Im Rahmen der Arbeit sollen in einem Bericht folgende Punkte dokumentiert werden:

- Pflichtenheft und Spezifikation der VR
- Beschrieb der untersuchten Lösungsvarianten und Begründung der Wahl der Realisierungsvariante
- Detailbeschreibung der Implementierung mit Testergebnissen
- Bedienanleitung und Kurzbeschrieb der Programme
- offene Punkte und Ausblick

## Termine

Kick-off Meeting: ab 19. Sept. 2011, nach Absprache in der KW 38

Besprechung: in der Regel einmal wöchentlich, nach Absprache

Präsentation der Arbeit: in der KW50 oder 51

Abgabe des Berichtes: 23. Dez. 2011

## B. Videoplayer

### B.1. Ziel

In einem ersten Schritt, vor der Realisierung des Fahrsimulator, wird ein Videoplayer realisiert. Das Ziel dabei ist der Aufbau und Test der Anbindung der Eingabehardware an unser Programm. Bei der Betätigung des Gaspedals im Cockpit soll das aufgenommene Video schneller abgespielt werden. Bei der Betätigung des Bremspedals dementsprechend langsamer. Der Videoplayer soll so aufgebaut werden, dass Komponenten davon auch im Fahrsimulator wiederverwendet werden können. Die ETH Zürich besitzt bereits Videos, die sich gut dafür eignen. Um Experimente mit diesen Videos durchführen zu können, müssen Eingaben, die der Proband im Cockpit macht, mit der aktuellen Position des Videos abgespeichert werden. Dies dient zur späteren Auswertung der Experimente.

### B.2. Systembeschreibung

Um einzelne Komponenten des Videoplayer wiederverwenden zu können, wird das System möglichst gleich wie das des Fahrsimulators aufgebaut. Dieser Aufbau wird anhand der Abbildung 18 illustriert. Blau markiert sind neu entwickelte Komponenten des Videoplayers. Die Eingaben im Cockpit werden von einem LabVIEW-Programm über eine USB-Schnittstelle eingelesen. Die eingelesenen Parameter werden, zusammen mit einem Timestamp, von LabVIEW in ein UDP-Paket verpackt und über einen UDP-Socket gesendet. Das Paket wird von einem UDP-Listener empfangen und gelesen. Dieser speichert die Daten ab und hält sie für das Hauptprogramm bereit. Das Hauptprogramm liest die gespeicherten Parameter aus und interpretiert diese. Der MPlayer<sup>(19)</sup>, der für das Abspielen der Videos verwendet wird, lässt sich durch Konsoleingaben über die Standard-In-Pipe steuern. So wird ihm vom Videoplayer mitgeteilt, wie schnell das aktuelle Video abgespielt werden soll. Über die Standard-Out-Pipe gibt MPlayer die aktuelle Abspielposition zurück. Das Hauptprogramm wertet diese aus, ergänzt sie mit den Benutzereingaben und leitet sie über den UDP-Listener an LabVIEW weiter. Dort werden die Daten über einen zweiten UDP-Socket empfangen und für eine spätere Auswertung in ein Log-File gespeichert.

---

<sup>(19)</sup>Mächtiger Opensource-Mediaspieler. Offizielle Website: <http://www.mplayerhq.hu>

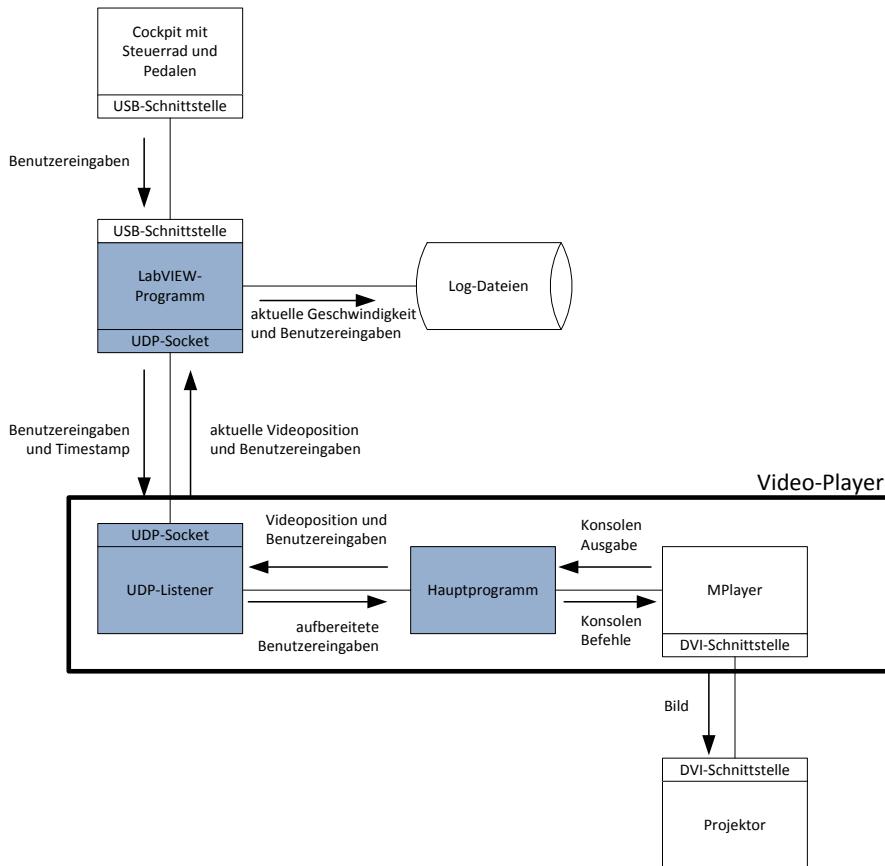


Abbildung 18: Systembeschreibung Videoplayer

### B.3. Realisierung

Bei der Realisierung des Videoplayers wird der Fokus vor allem auf die Entwicklung der LabVIEW-Schnittstelle zu unserem Programm gelegt. Darum wird dieser Schritt nachfolgend ausführlich erklärt. Diese Schnittstelle wird auch für den Fahrsimulator selbst verwendet.

#### B.3.1. LabVIEW-Schnittstelle

Die Realisierung der LabVIEW Schnittstelle wird in zwei Teile unterteilt. Der erste Teil dient dazu, die Benutzereingaben einzulesen und an den UDP-Listener zu senden. Der zweite Teil befasst sich mit dem Empfangen von Daten, die vom UDP-Listener gesendet werden und deren Speicherung in ein Log-File.

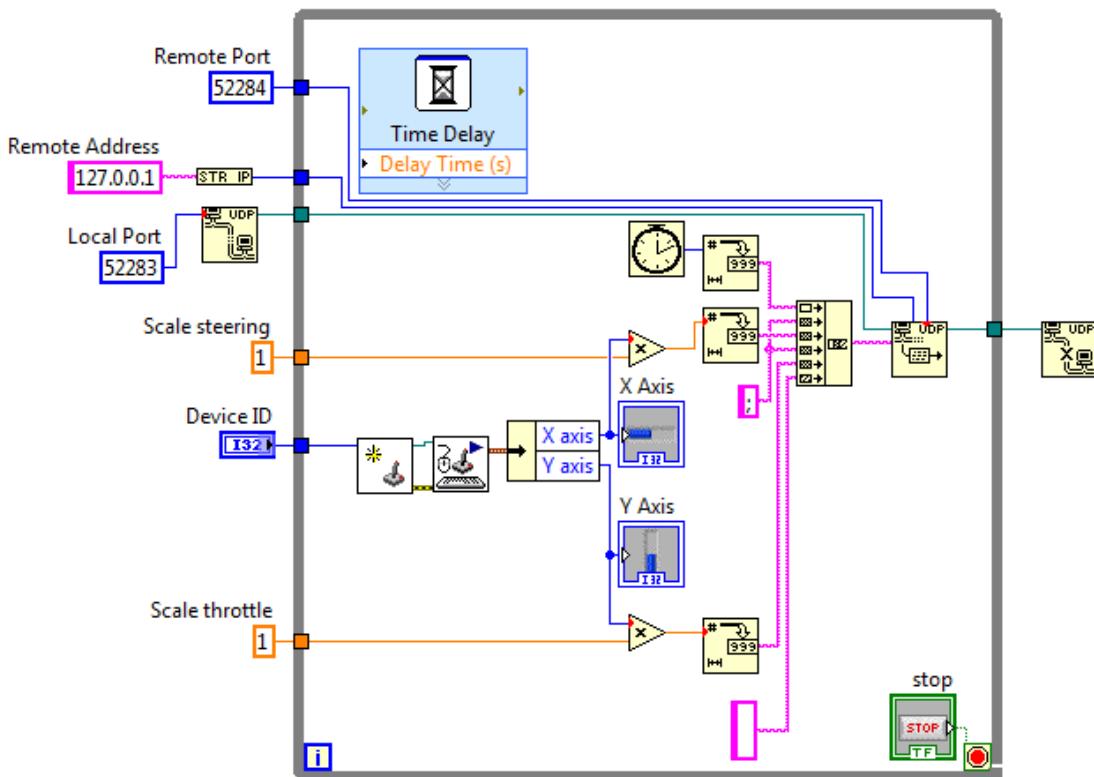


Abbildung 19: Einlesen und Versenden der Daten mit LabVIEW

Die Abbildung 19 illustriert den Aufbau des LabVIEW-Programms, mit dem Eingaben vom Cockpit eingelesen und über einen UDP-Socket versendet werden. Das Verhalten des Programms kann über diverse Parameter geändert werden.

Die *Remote Address* gibt an, an wen die UDP-Pakete gesendet werden. Da sich der UDP-Listener zur Zeit auf dem selben Rechner wie dieses Programm befindet, wählen wir hierfür die Localhost-Adresse 127.0.0.1. Für den *Remote Port* wählen wir einen freien Port, hier 52284. Zusammen mit dem *Local Port*, der uns eine Verbindungs-ID generiert, ist unsere Konfiguration für den UDP-Socket vollständig. Um die Komponenten im Cockpit ansprechen zu können, muss lediglich die richtige *Device ID* gewählt werden. Die anderen Komponenten, die sich in der grauen Box befinden, werden in einer Schleife 100 mal pro Sekunde ausgeführt. Dieser Wert wird mit dem Element *Time Delay* konfiguriert.

Aus dem Cockpit werden die Werte der x- und y-Achse des Joysticks eingelesen. Auf der y-Achse werden die Werte der beiden Pedalen abgebildet. Ein negativer Wert quantifiziert hierbei das Drücken des Gaspedals, ein positiver Wert das Drücken des Bremspedals. Die Intensität beider Pedalen wird im Positiven durch 32767 und im Negativen durch 32768 ganzzahlige Werte abgestuft. Ein voll gedrücktes Gaspedal entspricht also dem Wert -32768 auf der y-Achse und ein voll gedrücktes Bremspedal entspricht dem Wert 32767. Die x-Achse quantifiziert analog dazu den Einschlagswinkel des Steuerrades. Hierbei ist, wenn das Steuerrad sich in der neutralen Position befindet, der x-Wert null. Komplett nach rechts eingeschlagen beträgt der x-Wert 32767 und -32768 bei einem Einschlag nach links. Der x-Wert wird ebenfalls ausgelesen, ist aber im Video-Player nicht

relevant. Die beiden Werte werden je mit einem separaten Faktor multipliziert, um eventuell eine Anpassung derjenigen vorzunehmen. Danach werden sie zusammen mit dem aktuellen Timestamp durch Semikolons voneinander getrennt, in das UDP-Paket gepackt und dann versendet. Bei Beendigung des Programms wird die Schleife verlassen und der UDP-Socket wieder geschlossen.

Die Abbildung 20 illustriert den Aufbau des LabVIEW-Programms, mit dem die Daten vom Videoplayer empfangen und in ein Log-File gespeichert werden.

Es wird ein UDP-Socket geöffnet um ankommende Pakete zu empfangen. Der *Local port* mit dem UDP-Verbindungskästchen generiert wieder eine Connection-ID, die für das Öffnen des UPD-Ports benötigt wird. Durch die Angaben von *Maximum datagram size* und *Timeout* wird der UPD-Socket so konfiguriert, dass er nur Pakete kleiner als 4096 Bytes empfängt und höchstens 500 ms auf das Eintreffen eines Pakets wartet. Ist die Übertragung des Pakets fehlerhaft oder dauert sie zu lange, wird abgebrochen und auf das nächste Paket gewartet. Der hierbei von LabVIEW erzeugte Fehler wird ignoriert.

Bei einer erfolgreichen Übertragung liegen die Daten als eine Zeichenkette vor. Die einzelnen Werte in der Zeichenkette sind mit Semikolons voneinander getrennt. Diese Zeichenkette wird jetzt weiterverarbeitet. Die beiden Filmstreifenfenster bewirken, dass die Aktionen im zweiten Fenster erst ausgeführt werden, wenn die im ersten Fenster erzeugten Daten vollständig vorliegen. Dies ist vor allem für den Timestamp, der wieder von der Uhr kommt, wichtig, da dieser erst erstellt werden darf, wenn das Packet gelesen wurde. Die Werte werden voneinander getrennt und die Semikolons gelöscht. Sie repräsentieren nun verschiedenste Eigenschaften des Systems. So ist zum Beispiel der dritte Wert die aktuelle Geschwindigkeit des Fahrzeuges. Dieser Wert wird benötigt um auf der Benutzeroberfläche die Geschwindigkeit in einem Tacho anzuzeigen. Zudem wird auch der vierte Wert benötigt um die aktuelle Abspielposition des Videos auszugeben. Alle Werte, inklusive der aktuellen Position von Pedalen und Steuerrad, werden nun, wieder getrennt durch Semikolons, zu einer Zeichenkette zusammengefügt. Diese wird dann laufend in einem File gespeichert. Die erste Zeile des Files ist ein Header und beschreibt die Felder. Er wird in der Abbildung 20 unten links zusammengestellt. Nachdem die Schleife beendet wurde, werden UDP-Socket und File geschlossen.

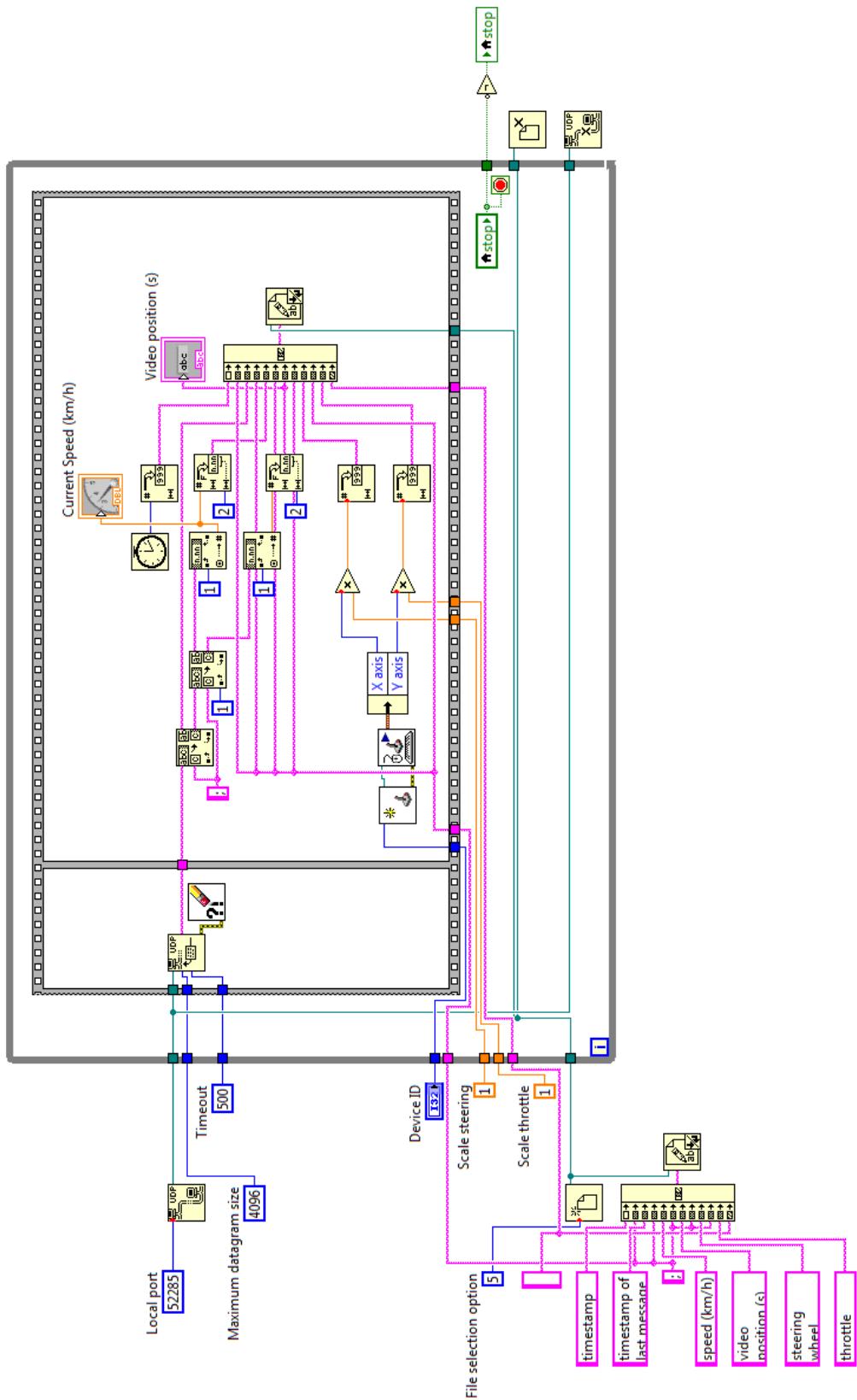


Abbildung 20: LabVIEW: Daten empfangen

### B.3.2. UDP-Listener

Der UDP-Listener empfängt die Pakete, die von LabVIEW gesendet werden. Diese werden ausgepackt und die übermittelten Werte abgespeichert. Der UDP-Listener wird im Kapitel 3.2 genau dokumentiert.

### B.3.3. Hauptprogramm

Das Hauptprogramm startet den MPlayer und den UDP-Listener. Der MPlayer wird, um ein Video abzuspielen, in einem eigenen Prozess gestartet und kann durch Aufrufe über die Standard-In-Pipe, wie im Listing 13 beschrieben, im Slavemode gestartet werden. Wie der MPlayer im Videoplayer aufgerufen wird, kann dem Quellcode auf der beiliegenden DVD entnommen werden.

```
mplayer .exe -slave -hardframdrop -osdlevel 0 BeispielVideo.mp4
```

Listing 13: Starten des MPlayers in einem eigenen Prozess

Der Parameter *hardframdrop* dient der Erhöhung der Framerate. Es erlaubt dem Player bei hohen Abspielgeschwindigkeiten einzelne Bilder auszulassen um so eine höhere Framerate zu erzielen. Der *osdlevel* Parameter bestimmt wieviel Information auf dem Bildschirm angezeigt wird. Der Level null unterdrückt alle Ausgaben.

In diesem Slavemode nimmt der MPlayer Befehle die über die Standard-In-Pipe entgegen und schreibt seine Ausgabe in die Standard-Out-Pipe. Dazu werden die beiden Pipes des MPlayer-Prozesses im Hauptprogramm verfügbar gemacht und erlauben so eine Kommunikation der beiden Programme.

```
int sendMessage(const char* message)
{
    DWORD dwWritten;
    if (!WriteFile(stdInWr, message, strlen(message), &dwWritten, NULL))
    {
        fprintf(stderr, "Could not write to pipe.\n");
        return 1;
    }
    return 0;
}
```

Listing 14: Funktion zum Senden von Nachrichten an MPlayer

Im Listing 14 quantifiziert *speed* die Abspielgeschwindigkeit in der das Video abgespielt werden soll.

## C. Das OGRE-Framework

### C.1. Was ist das OGRE-Framework

OGRE (Object-Oriented Graphics Rendering Engine) ist eine Grafikengine zur Echtzeitdarstellung von dreidimensionalen Szenen. Sie ist in C++ geschrieben und ihre Verwendung unterliegt der MIT-Lizenz<sup>(20)</sup>. Durch den modularen Aufbau und die Unterstützung auf verschiedenen Plattformen erweist sich OGRE als sehr flexibel und mächtig. OGRE selbst benutzt die Grafikbibliotheken OpenGL<sup>(21)</sup> und DirectX<sup>(22)</sup> zum hardwarebeschleunigten Rendern der Szenen und setzt automatisierte Optimierungsalgorithmen zur Geschwindigkeitsgewinnung ein. Mittlerweile existiert eine grosse und aktive Community, welche das OGRE-Framework ständig weiter entwickelt und verbessert.



Abbildung 21: OGRE Logo

### C.2. Welche Features bringt OGRE mit?

- Anbindung an OpenGL und DirectX.
- Verfügbarkeit von SDKs für Windows, Linux, Mac OS X und iOS<sup>(23)</sup>.
- Unterstützung für Shader.
- Organisation von Objekten basiert auf Szenengraph<sup>(24)</sup>.
- Möglichkeit zum Importieren von statischen und animierten 3D-Objekten.
- Sehr gute und detaillierte Dokumentation.

Quelle: [ogre3d.org](http://www.ogre3d.org)<sup>(25)</sup>

<sup>(20)</sup>Eine vom Massachusetts Institute of Technology entworfene Lizenz zur Benutzung von Software. (Quelle: [http://en.wikipedia.org/wiki/MIT\\_License](http://en.wikipedia.org/wiki/MIT_License), Abruf: 03.12.2011)

<sup>(21)</sup>Opensource Grafikbibliothek

<sup>(22)</sup>Microsofts Grafikbibliothek und Konkurrenz zu OpenGL

<sup>(23)</sup>Betriebssystem für Apples iPhone, iPad und iPod touch

<sup>(24)</sup>Datenstruktur zur hierarchischen Organisation von Objekten

<sup>(25)</sup><http://www.ogre3d.org/about/features>, Abruf: 03.12.2011

### C.3. Weshalb wird OGRE eingesetzt?

Im Vergleich zur direkten Verwendung von OpenGL oder DirectX bringt der Einsatz einer Grafikengine wie OGRE eine Vielzahl von Vorteilen mit sich:

#### **Abstraktion**

OpenGL und DirectX sind zwei komplett verschiedene Bibliotheken. Hat man sich für eine von beiden entschieden, so bedeutet das Umsteigen auf eine andere unter Umständen das Neuschreiben des kompletten Codes. OGRE abstrahiert die Verwendung dieser Grafikbibliotheken und ermöglicht den selben Code entweder mit OpenGL oder DirectX laufen zu lassen.

#### **Geschwindigkeit**

Ohne Optimierung kann das Darstellen einer 3-dimensionalen Welt sehr langsam werden und den Rechner bzw. die Grafikkarte stark auslasten. Mit ausgefeilten Optimierungsalgorithmen wird die Szene zur Laufzeit so angepasst, dass nur Objekte dargestellt werden, die sich im Sichtbereich befinden.

#### **Portierbarkeit**

OGRE kümmert sich um viele Aspekte der grafischen Programmierung und vereinheitlicht z.B. das Erstellen eines Grafikkontexts auf verschiedenen Plattformen. So ist es möglich, das selbe Programm für Windows, Mac OS X oder Linux zu kompilieren.

## D. Betriebsanleitung

Hier wird in einer kurzen Schritt-für-Schritt-Anleitung erklärt, wie die Applikationen *VideoPlayer* und *DrivingSimulator* zu bedienen sind.

### D.1. VideoPlayer

Der VideoPlayer benötigt eine funktionierende Installation des Programms MPlayer. MPlayer kann auf der Herstellerseite (<http://www.mplayerhq.hu>) kostenlos heruntergeladen werden.

#### 1. LabVIEW-Programm starten

Öffnen Sie die Datei *VideoPlayer.vi* mit LabVIEW und starten Sie das Programm mit einem Klick auf die Schaltfläche Run. Sie werden nach einem Dateipfad gefragt. Geben Sie hier den Pfad zu einer Datei an, in der Sie den Output vom LabVIEW-Programm gespeichert haben wollen. **Achtung: Existierende Dateien werden überschrieben!**

#### 2. VideoPlayer starten

Starten Sie *VideoPlayer2.exe* entweder direkt von der Kommandozeile mit folgenden Parametern in der Reihenfolge, wie sie hier aufgelistet sind:

- Pfad zu mplayer.exe
- Pfad zur Videodatei
- Startzeit der gewünschten Videosequenz (in Sekunden)
- Endzeit der gewünschten Videosequenz (in Sekunden)
- Referenzgeschwindigkeit, d.h. Geschwindigkeit mit der das Video aufgenommen wurde (in km/h)
- UDP Input Port
- UDP Output Port
- Adresse des Remote Computers ("127.0.0.1" wenn das LabVIEW-Programm auf demselben Computer läuft)

oder benutzen Sie die Datei *VideoPlayer2.bat*, welche bereits mit den Standardwerten konfiguriert ist und ein beigelegtes Beispielvideo abspielt.

#### 3. Los gehts

Setzen Sie sich nun ans Steuer und kontrollieren Sie die Geschwindigkeit des Fahrzeugs durch drücken des Gas- bzw. Bremspedals. Die aktuelle Geschwindigkeit und Position im Video wird im LabVIEW-Programm angezeigt. Alle Informationen werden, solange das Programm läuft, in die, in Schritt 1 angegebene, Datei geschrieben.

## D.2. Fahrsimulator

Der Fahrsimulator benötigt DirectX 9 um zu laufen. Falls Sie dies noch nicht installiert haben, können Sie es von der offiziellen Microsoft-Seite<sup>(26)</sup> herunterladen:

### 1. LabVIEW-Programm starten

Öffnen Sie die Datei DrivingSimulator.vi mit LabVIEW und starten Sie das Programm mit einem Klick auf die Schaltfläche Run. Sie werden nach einem Dateipfad gefragt. Geben Sie hier den Pfad zu einer Datei an, in der Sie den Output vom LabVIEW-Programm gespeichert haben wollen. **Achtung: Existierende Dateien werden überschrieben!**

### 2. Grafikeinstellungen überprüfen

Öffnen Sie die Datei *Resources/graphics.cfg*. Darin befinden sich einige Attribute für die Grafikkonfiguration. Vergewissern Sie sich, dass Ihre Grafikkarte die dort spezifizierten Grafikmodi unterstützt und ändern Sie diese gegebenenfalls ab. Die wichtigsten Optionen sind hier aufgeführt:

- Video Mode: Bildschirmauflösung und Farbtiefe in folgendem Format: <horizontale Auflösung> x <vertikale Auflösung> @ <Farbtiefe>-bit colour  
Beispiel: 1024 x 768 @ 32-bit colour
- FSAA: Anzahl Filterdurchgänge für Antialiasing (0, 2, 4, 8, etc.)
- Full Screen: Yes oder No
- VSync: Bildschirmsynchronisation aktivieren (Yes oder No)

### 3. Fahrsimulator starten

Wechseln Sie zuerst in das Verzeichnis, in dem sich die Datei *DrivingSimulatorV1.exe* befindet und starten Sie diese entweder direkt von der Kommandozeile mit folgenden Parametern in der Reihenfolge, wie sie hier aufgelistet sind:

- UDP Input Port
- UDP Output Port
- Adresse des Remote Computers ("127.0.0.1" wenn das LabVIEW-Programm auf demselben Computer läuft)
- Rückwärtsgang (1: aktiviert, 0: deaktiviert)
- Steuerrad (1: sichtbar, 0: unsichtbar)
- Level (1: Stadtszene, 2: Berglandschaft)

oder benutzen Sie die Dateien *DrivingSimulator\_City.bat* oder *DrivingSimulator\_Mountains.bat*, um eine entsprechende Szene mit Standardeinstellungen zu laden.

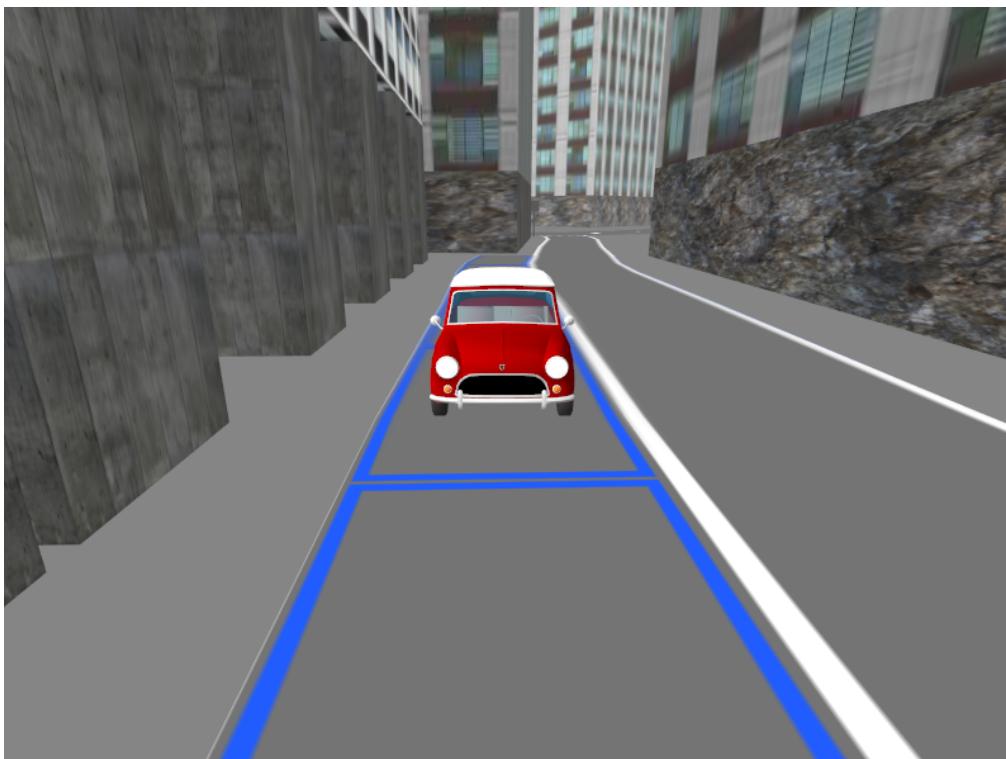
### 4. Los gehts

Setzen Sie sich nun ans Steuer und fahren Sie durch die virtuelle Welt. Das Fahrzeug kann über Eingaben im Cockpit oder durch die Pfeiltasten gesteuert werden. Durch Drücken der Taste V auf der Tastatur, wird die Kameraansicht zwischen 1st- und 3rd-Person umgestellt.

---

<sup>(26)</sup><http://www.microsoft.com/download/en/details.aspx?id=8109>

## E. Screenshots





## F. Detaillierter Zeitplan

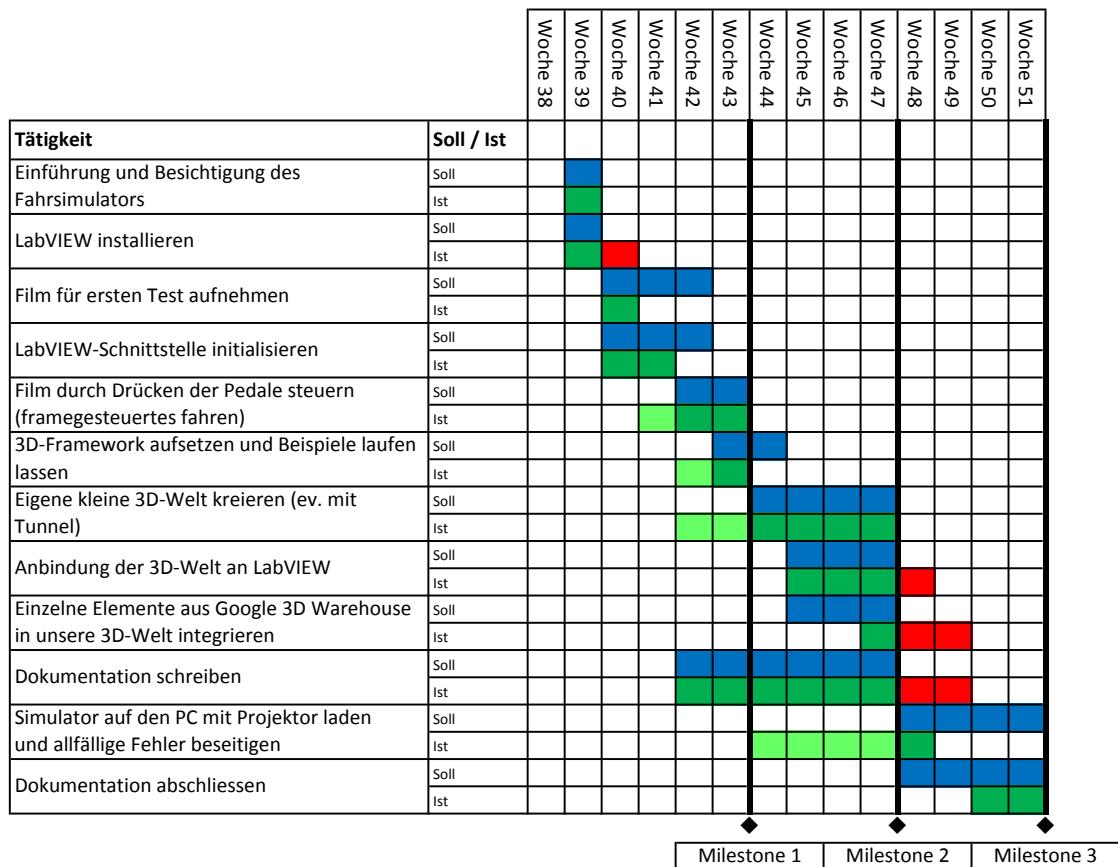


Abbildung 22: Detaillierter Zeitplan

### Einführung und Besichtigung des Fahrsimulators

Der Aufbau des Fahrsimulators und die Softwareumgebung wird besichtigt. Da das Projekt für die ETH durchgeführt wird, werden die Verantwortlichen vorgestellt und mit ihnen Ziele und Anforderungen an die Arbeit besprochen.

### LabVIEW installieren

Das LabVIEW Programm wird auf unseren privaten Rechnern installiert. Wir machen uns damit vertraut können es für die Arbeit verwenden.

### Film für ersten Test aufnehmen

Um einen ersten Test mit einem Video-Player machen zu können wird ein geeignetes Video benötigt. Das Video zeigt Aufnahmen von einer Autofahrt und beinhaltet vorzugsweise eine Tunneleinfahrt.

### **LabVIEW-Schnittstelle initialisieren**

Im LabVIEW wird eine Schnittstelle eingerichtet um, die Werte des Cockpits in den Fahrsimulator zu übertragen. Entsprechend wird ein Gegenstück entwickelt um die Werte, die von LabVIEW gesendet werden, zu empfangen.

### **Film durch drücken der Pedale steuern (framegesteuertes fahren)**

Realisation eines Video Players, der einen Film abspielt. Die Abspielgeschwindigkeit wird durch das Drücken des Gas- oder Bremspedals im Cockpit beeinflusst. Beim drücken des Gaspedals wird der Film schneller und beim Bremspedal langsamer.

### **3D-Framework aufsetzen und Beispiele laufen lassen**

Für die Entwicklung des Fahrsimulators wird ein 3D-Framework mit dem Namen OGRE verwendet. Dies soll in diesem Arbeitsschritt aufgesetzt, konfiguriert und getestet werden. Nach erfolgreichem Testen wird eine kurze einfache Szenen, die nur eine gerade Strasse enthält, erstellt. Die Szene wird aus dem Blickwinkel des Fahrers oder aus der Vogelperspektive gesehen. Zu Testzwecken soll das Fahrzeug mit den Pfeiltasten der Tastatur gesteuert werden.

### **Eigene kleine 3D-Welt kreieren (ev. mit Tunnel)**

Es wird eine eigene 3D-Welt kreiert. Diese Szene kann bereits ein Tunnel enthalten.

### **Anbindung der 3D-Welt an LabVIEW**

In diesem Arbeitsschritt werden die LabVIEW Schnittstelle und die 3D-Welt zusammengefügt. Danach ist es möglich, das Fahrzeug in der Simulation durch Manipulation im Cockpit zu steuern.

### **Einzelne Elemente aus Google 3D Warehouse in unsere 3D-Welt integrieren**

Es werden Objekte aus dem Google 3D Warehouse in die 3D-Welt integriert, um die Umgebung noch realistischer zu gestalten.

### **Dokumentation schreiben**

Mit der Dokumentation wird in der zweiten Hälfte der zur Verfügung stehenden Zeit begonnen. Die Besprechungen mit dem ETH Team oder den internen Betreuern werden von Beginn an protokolliert.

### **Simulator auf den PC mit Projektor laden und allfällige Fehler beseitigen**

Der getestete Fahrsimulator wird definitiv auf den Rechner geladen und getestet. Kleinere Fehler und Unschönheiten werden noch beseitigt.

### **Dokumentation abschliessen**

Die Dokumentation wird von mehreren Personen gegengelesen, korrigiert und die definitive Version dann ausgedruckt und abgegeben.

## G. Listings

### G.1. OGRE-Konfigurationsfiles

```
#Define plugin folder
PluginFolder=.

#Define plugins
Plugin=RenderSystem_Direct3D9
Plugin=RenderSystem_GL
Plugin=Plugin_ParticleFX
Plugin=Plugin_BSPSceneManager
Plugin=Plugin_CgProgramManager
Plugin=Plugin_PCZSceneManager
Plugin=Plugin_OctreeZone
Plugin=Plugin_OctreeSceneManager
```

listings/plugins.cfg

```
Render System=Direct3D9 Rendering Subsystem

[ Direct3D9 Rendering Subsystem ]
Allow NVPerfHUD>No
FSAA=4
Floating-point mode=Fastest
Full Screen>No
Resource Creation Policy>Create on all devices
VSync=Yes
VSync Interval=1
Video Mode=1024 x 768 @ 32-bit colour
sRGB Gamma Conversion>No
```

listings/graphics.cfg

```
[ Models ]
FileSystem=Resources/Models/MiniCooper
FileSystem=Resources/Models/Mountains
FileSystem=Resources/Models/City
FileSystem=Resources/Models/ShadowTest
FileSystem=Resources/Models/PhysicsTest
FileSystem=Resources/Models/ETH
FileSystem=Resources/Models/MiniCockpit
FileSystem=Resources/Models/SimpleGeometry
```

listings/resources.cfg

## G.2. LabVIEW-Programme

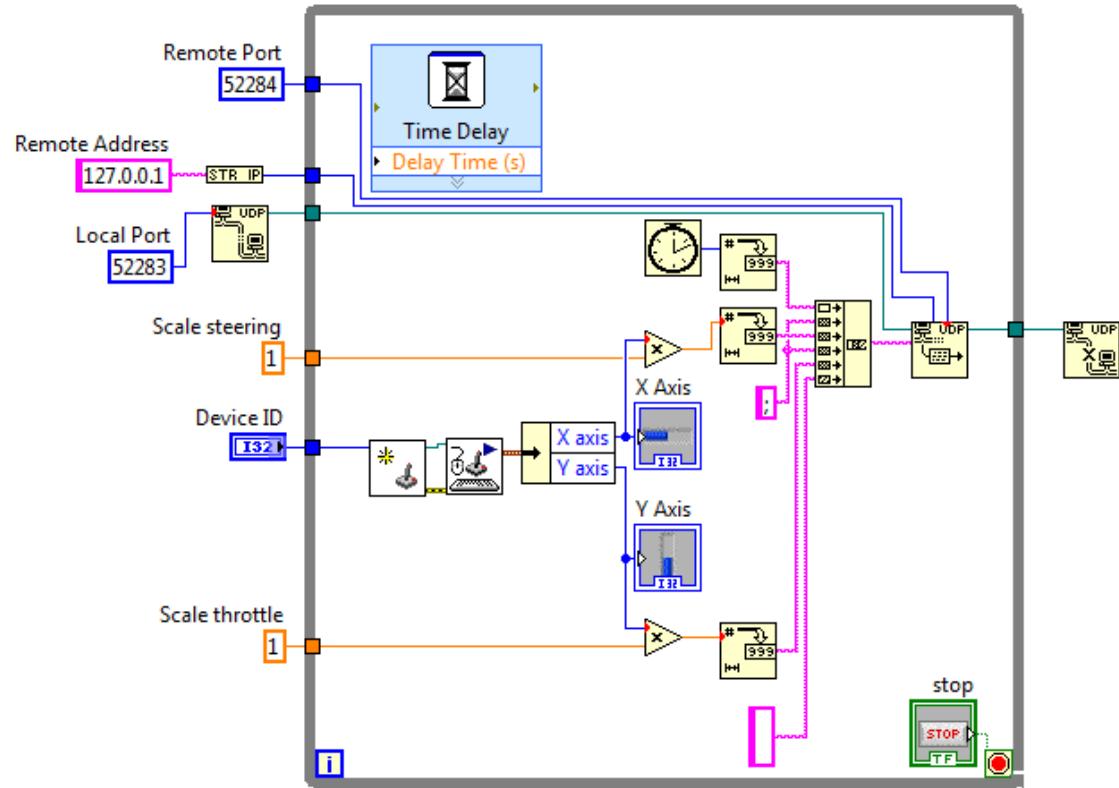


Abbildung 23: LabVIEW: Daten werden gesendet

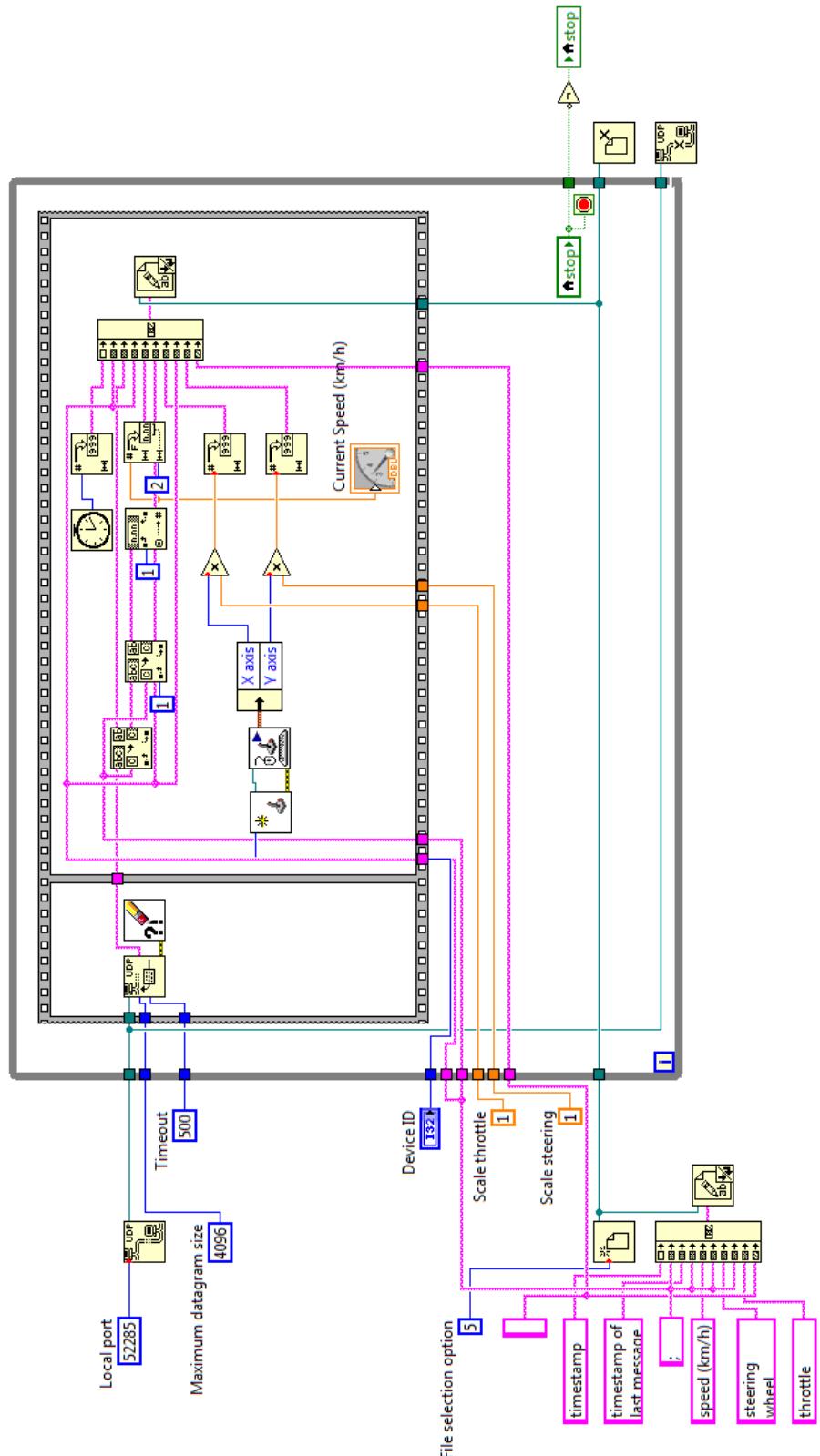


Abbildung 24: LabVIEW: Daten werden empfangen

## H. Journal

### 26.09.2011

The driving simulator with steering wheel and pedals is connected to the computer. There is a LabVIEW interface which reads the input from the devices. In a first step, Rudy would like to have a driving controlled only by the pedals. -> modified frame rate  
After this she'd like to have a driving simulator with focus on tunnel entrances and exits.

### 03.10.2011

The driving simulator may be a fantasy environment or it could be a copy of a real environment.

Google Street View is no possibility to generate a 3D world because there are too many distractions in it. For example the other cars, people and traffic jams. The street view does not clearly distinguish between street and surfaces of other objects. Also sometimes there are only pictures available on the wrong side of the street. Another argument not to use Google Street View is that there are too large distances between the single pictures, so the rendering is not fluent.

We would like to use the information from Google Earth to build a city like Zurich. We could use the street location information to build our own streets and then try to render already existing 3D models from Google 3D Warehouse into our virtual world.

There is a difficulty about Google Earth when it comes to creating scenes that take place in a tunnel since there is no height information available. A possibility could be to implement it manually or to ignore spots which cause such problems. If we implement it manually we should define a fixed area where no problems occur.

We decided to use UDP sockets to extract data out of LabVIEW into our program and an external video player application to control the video.

In a first step we use LabVIEW to control an external application which plays a video with a configurable framerate.

### 10.10.2011

The journal and the timetable have been set up and seem to be okay. There has to be an English version of the timetable so Rudy can follow the progress made in the project. We have agreed on creating our own 3D world and extend it with some buildings from Google 3D Warehouse. There are already tons of finished 3D models of different buildings. We have to build the streets by ourselves because the streets in Google Earth are not as good as we need them. We will also create a tunnel in our own 3D world, which is impossible when rendering a scene with material from Google Street View.

We have presented the video we controlled with LabVIEW and the OGRE framework we would like to use.

### 17.10.2011

We have to calculate the delay time of the user interaction. A timestamp would be very helpful to study the delays. That's important for Rudy's further studies.

The video is now controlled by the pedals, and played in MPlayer. There is a batch file to

start the application with different videos. We switched our repository to github because there it's easier to maintain.

### **07.11.2011**

It should be possible to get this work further as our Bachelor thesis.

We have brought up some ideas to build streets and cities dynamically with little pieces of street tales.

We also could create a city (or at least a map) by ourselves. It is only useful to use Google Street View or Google Maps if it bring a remarkable time advantage over doing the scenes manually.

Rudy told us some scenarios which she'd like to have for her studies. We will try to create the most of them but we have decided that if there are some elements in it which are animated or have to be controlled from outside, for example another car, we will move it to the Bachelor thesis.

We agreed on building a tunnel scene but won't be able to create a controllable second car moving through that scene.

### **14.11.2011**

Today we agree on a set of features that have to be included in the software we will deliver as the result of our project. Namely, these are:

- Logging of the timestamp in the VidePlayer application
- Integration of a cockpit view with speedometer
- Small city map

After having completed these tasks we will focus on updating the documentation and making a deliverable version of the software.

Tasks to be included in the Bachelor thesis will be discussed in the meeting of the 28th November.

### **21.11.2011**

We have got different ideas, we would like to implement in the Bachelor thesis. Important about these ideas is a relevance for the experiment they do with the driving simulator. Rudy also gave us a lot of points we could implement in a further work. These are things like a bigger scene simulating an area in Zurich around the airport including the Bubenhofen tunnel.

### **05.12.2011**

A simple goal of the Bachelor thesis has been set up and we named different possible tasks for it.

We discussed the structure we had set up for the PA documentation and got some good hints from Mr. Frueh and Mr. Schlup. There has been a short interview with Prof.

Menozzi on a Swiss television programme about his studies. Our driving simulator has been mentioned and shown on TV<sup>(27)</sup>.

### **12.12.2011**

We verified the definitive structure of the PA documentation. We settled the closing date for the documentation on the 23.12.2011. We are on the right way and absolutely on time with our work.

---

<sup>(27)</sup><http://www.videoportal.sf.tv/video?id=9410c439-0c70-4ee2-895c-f9339ec2809d>

## Glossar

**1st-Person** Ich-Perspektive.

**3rd-Person** Kameraeinstellung, bei der die Kamera aus dem Blickwinkel einer dritten Person (Beobachter) filmt.

**A-Säule** Die vordersten beiden Säulen beim Auto, die das Dach tragen.

**Callback-Methode** Methode, die von einem externen Programmteil aufgerufen wird.

**Cinema4D** Kommerzielles 3D-Modellierungstool vom Softwarehersteller MAXON.

**DirectX** Microsofts Grafikbibliothek und Konkurrenz zu OpenGL.

**Google 3D Warehouse** Eine Plattform, auf der Modelle aus Google SketchUp veröffentlicht und kostenlos heruntergeladen werden können.  
<http://sketchup.google.com/3dwarehouse>.

**Google SketchUp** Software zum Erstellen von Objekten. Sie wurde ursprünglich zur Modellierung von Gebäuden für Google Earth entwickelt, unterstützt aber auch andere bekannte 3D-Formate.

**GUI** Abkürzung für Graphical User Interface (deutsch: Grafische Benutzeroberfläche).

**LabVIEW** Ein grafisches Programmiersystem vom Softwarehersteller National Instruments.

**Mipmap** Gefilterte und verkleinerte Version einer Textur, um dem Aliasing-Effekt entgegenzuwirken.

**MIT-Lizenz** Eine vom Massachusetts Institute of Technology entworfene Lizenz zur Benutzung von Software. (Quelle: [http://en.wikipedia.org/wiki/MIT\\_License](http://en.wikipedia.org/wiki/MIT_License), Abruf: 03.12.2011).

**MPlayer** Mächtiger Opensource-Mediaplayer. Offizielle Website: <http://www.mplayerhq.hu>.

**Node** Element des Szenengraphen, das eine Transformation und Modelle bzw. weitere Nodes enthält.

**OGRE** Object-Oriented Graphics Rendering Engine.

**OGRE-XML** Von der OGRE-Community entworfenes Format zur Speicherung von 3D-Modellen. Es existieren Plugins für diverse 3D-Modellierungstools, mit denen Objekte in diesem Format gespeichert werden können.

**OpenGL** Opensource Grafikbibliothek.

**Shader** Programm, welches auf der GPU läuft und der Berechnung von Licht-, Schatten- und weiteren Effekten von Materialien dient.

**Spline** Eine Linie die durch Interpolation von einzelnen Punkten definiert ist.

**Szenengraph** Datenstruktur zur hierarchischen Organisation von Objekten.

**Untersteuern** Rutschen des Fahrzeugs über die Vorderachse in der Kurve.

**Vsync** Vertical Sync - bedeutet, dass mit dem Neuberechnen eines Bildes gewartet wird, bis der gesamte Bildschirminhalt gezeichnet wurde. Dadurch wird die Verzerrung des Bildes durch schnelle Bewegungen verhindert.

## **Abbildungsverzeichnis**

Abbildung 1:	Hardwareaufbau mit Steuerrad und Pedalen . . . . .	6
Abbildung 2:	Systembeschreibung . . . . .	7
Abbildung 3:	Koordinatensystem der Joystickeingabe . . . . .	11
Abbildung 4:	Verschwommene Interpolation von Google Street View Bildern . . . . .	12
Abbildung 5:	Karte der StadtSzene . . . . .	15
Abbildung 6:	Beispiele von Strassentexturen . . . . .	16
Abbildung 7:	Beispiele eines Verkehrssignals . . . . .	16
Abbildung 8:	Drei verschidene Gebäudetypen . . . . .	17
Abbildung 9:	Das bearbeitete Modell der ETH Zürich . . . . .	17
Abbildung 10:	Ein Baum-Modell . . . . .	18
Abbildung 11:	Ausschneiden des Tunnels mit Cinema4D . . . . .	18
Abbildung 12:	Der Mini Cooper aus der 3rd-Person-Ansicht . . . . .	19
Abbildung 13:	Innenansicht des Mini mit Geschwindigkeitsanzeige . . . . .	20
Abbildung 14:	Berechnung der tatsächlichen Geschwindigkeit . . . . .	21
Abbildung 15:	Ablauf des Hauptprogramms . . . . .	22
Abbildung 16:	Steuerintensität in Abhängigkeit der Geschwindigkeit . . . . .	28
Abbildung 17:	Sequenzdiagramm des Fahrsimulators . . . . .	30
Abbildung 18:	Systembeschreibung Videoplayer . . . . .	37
Abbildung 19:	Einlesen und Versenden der Daten mit LabVIEW . . . . .	38
Abbildung 20:	LabVIEW: Daten empfangen . . . . .	40
Abbildung 21:	OGRE Logo . . . . .	42
Abbildung 22:	Detailierter Zeitplan . . . . .	48
Abbildung 23:	LabVIEW: Daten werden gesendet . . . . .	51
Abbildung 24:	LabVIEW: Daten werden empfangen . . . . .	52