Technische Universität Ilmenau

Fakultät für Informatik und Automatisierung

Studienjahresarbeit

Visualisierung von Unterwasserfahrzeugen, die durch externe Simulatoren gesteuert werden

CViewVR
Sea View Virtual Reality

---

vorgelegt von:	Andreas Voigt
eingereicht am:	18. 07. 2011
geboren am:	11. 05. 1980 in Komarno
Studiengang:	Ingenieur-Informatik
Studienrichtung:	Informatikkomponenten für Intelligente Systeme (IKIS) und Systemtechnik - Intelligente Steuerungen (ST)
Anfertigung im Fachgebiet:	Systemanalyse Fakultät für Informatik und Automatisierung
Verantwortlicher Professor:	Univ.-Prof. Dr.-Ing. habil. Christoph Ament
Wissenschaftliche Betreuer:	Dipl.-Ing. Thomas Glotzbach (TU Ilmenau) Dr.-Ing. Torsten Pfützenreuter (Fraunhofer AST)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Hintergrund

Unbemannte Fahrzeuge können in Umgebungen agieren, die für den Menschen gefährlich oder nur mit aufwendigen Schutzeinrichtungen erreichbar sind. Aus den Erfahrungen mit ferngesteuerten Systemen entwickelte sich schnell der Wunsch, autonome Fahrzeuge insbesondere für langwierige und ermüdende Routinetätigkeiten zu nutzen. Autonom bedeutet, daß die Fahrzeuge nicht ferngesteuert sind, sondern selbstständig agieren, um vorgegebene Aufgaben zu erfüllen. Die Orientierung innerhalb Umwelt geschieht in der Regel über Kameras, Sensoren für Lage-, Positions- und Abstandsbestimmung, sowie Sonar. Es gibt bereits praktische Untersuchungen, bei denen sich Teams von mehreren autonom agierenden Unterwasserfahrzeugen selbstständig koordinieren [URL 2].

Typische Einsatzgebiete für autonome Unterwasserfahrzeuge sind:

- Inspektion von Hafenbecken, Schleusen, Talsperren, Offshore-Windparks, Pipelines und sonstigen Unterwasserbauwerken,
- Inspektion von Schiffen und anderen beweglichen Objekten,
- Vermessung und Untersuchung des Meeresbodens (Topologie, Methan-Austrittsstellen, etc.),
- Durchführung von Messungen (z.B. Temperatur, Salinität, Strömungen, Schadstoffe),
- Suche nach vermissten Objekten (versunkene Schiffe, Flugzeug-Blackboxen, Leichen, etc.),
- Untersuchung des Verhaltes von Fischschwärmen,
- Minensuche und -beseitigung.

Projekte, die sich mit autonomen Unterwasserfahrzeugen beschäftigen, sind beispielsweise zu finden unter [URL 4] [URL 5] [URL 6] [URL 7] [URL 8] [URL 9] [URL 10] [URL 17].

1.2 Aufgabenstellung

Im Gegensatz zu menschlich ferngesteuerten Fahrzeugen ist für Autonome Unterwasserfahrzeuge (AUVs) ein intelligentes Führungssystem erforderlich, welches die vorgegebenen Aufgaben („Missionspläne“) erfüllt. Das Führungssystem muß Sensordatendaten auswerten, richtig interpretieren und entsprechend agieren. Durch die fehlende menschliche Kontrolle, ist ein hoher Grad an Robustheit des Fahrzeugführungssystems erforderlich, um auch in besonderen Situationen richtig zu reagieren und beispielsweise Kollisionen zu vermeiden.

Für die Entwicklung, Validierung und Optimierung von Führungssoftware ist nicht nur ein Dynamiksimulator hilfreich, sondern auch eine Softwareumgebung, die Fahrbewegungen darstellen und die Interaktion mit der Umwelt (z.B. Meeresboden, andere Fahrzeuge, Unterwasserbauwerke) widerspiegeln kann.

Grundlegendes Ziel der Studienjahresarbeit war, nach Möglichkeiten zu suchen, um eine *Visualisierung* von Unterwasserszenarien zu ermöglichen. Weiterhin war gefordert, daß sich die Visualisierung erweitern läßt. Dabei gab es folgende Schwerpunkte:

- Eine *Integration existierender 3D-Modelle* von Unterwasserfahrzeugen (z.B. den *Seewolf* von der Firma Atlas Elektronik[URL 3]) und die Erstellung von Szenarien (z.B. Hafenanlage) muß möglich sein.

- Fahrzeuge müssen sich über eine *Netzwerkschnittstelle* von externer Fahrzeugsimulationssoftware steuern lassen. Positionsangaben sollen einem *NED-Koordinatensystem* (englisch: North-East-Down) entsprechen, bei dem die X-Achse nach Norden, die Y-Achse nach Osten und die Z-Achse nach unten zeigt. Die räumliche Orientierung wird durch Eulerwinkel im Bogenmaß beschrieben. Im Initialzustand ($yaw = 0°$) sollen die Fahrzeuge nach Norden ausgerichtet sein. Abbildung 1.1 (S. 7) veranschaulicht die Vorgaben.

- Der Nutzer soll die Szene aus einem beliebigen *Blickwinkel* betrachten können (frei bewegliche Kamera, fahrzeuggekoppelte Kamera, Vogelperspektive).

- Eine *Erweiterbarkeit* für die Integration weiterer Module soll gegeben sein (z.B. Kollisionserkennung, Latitude-Longitude-Koordinaten (WGS 84), Sonarsimulation, Strömungssimulation).

- *Präsentationsmöglichkeiten* waren erwünscht (Speicherung und Wiedergabe von Fahrzeugbewegungen, Screenshoterstellung, Videoerstellung).

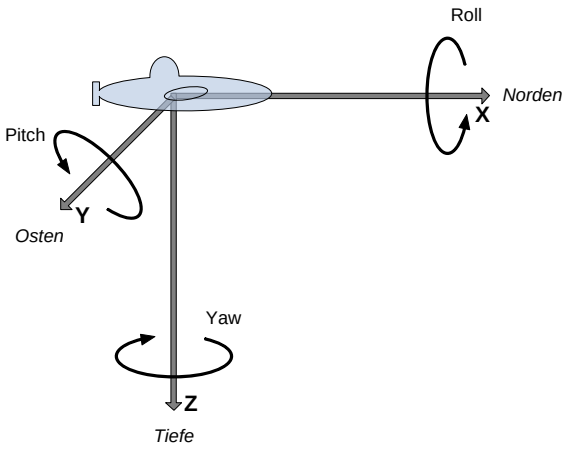


Abbildung 1.1: Räumliche Ausrichtung eines Objektes im NED-Koordinatensystem

1.3 Aufbau der Arbeit

Diese Arbeit besteht aus den drei Schwerpunkten *Grundlagen*, *Integration von 3D-Modellen* und *Entstandene Software CViewVR*.

In den *Grundlagen* werden die Recherchen bezüglich 3D-Visualisierung zusammengefasst, die Grafikbibliothek *Ogre* vorgestellt und die verwendeten Koordinatensysteme, sowie deren Rotationsvorschriften beschrieben.

Abschnitt *Integration von 3D-Modellen* beschreibt, wie die Ogre-Bibliothek Modelle verwaltet und wie solche eingebunden werden können, die nicht im Ogre-Format vorliegen. Ein Kapitel über das 3D-Modellierungsprogramm *Blender* beschreibt detailliert, wie Modelle „aufbereitet“ werden müssen und ein weiteres, wie diese in die Visualisierungssoftware CViewVR eingebunden werden.

Der dritte Schwerpunkt beleuchtet die *Entstandene Software CViewVR*. Es werden Funktionalität und Bedienung beschrieben, das Szenendateiformat, die vereinfachte Latitude-Longitude-Umrechnung und das auf TCP basierende Kommunikationsprotokoll ausführlich erklärt. Zudem werden drei Hilfsprogramme vorgestellt, die ebenfalls entwickelt wurden.

Das Kapitel *Anwendungsbeispiele & Screenshots* zeigt das Ergebnis dieser Arbeit anhand von prakischen Anwendungen der Visualisierungssoftware. Die darauf folgende *Zusammenfassung* gibt einen abschließenden Überblick.

Im Anhang befindet sich ein sogenanntes *Fehlerhandbuch*, welches typische Probleme beschreibt, die während der Entwicklung und Nutzung von CViewVR auftraten. Die enthaltenen Hinweise sollen zukünftigen Anwendern dabei helfen, derartige Probleme schnell zu lösen. Ein weiterer Abschnitt im Anhang beschreibt, wie Fahrzeugmodelle der Simulationssoftware *Matlab/Simulink* Daten an CViewVR senden können.

2 Grundlagen

2.1 Grafische Darstellung

Die grundlegende Aufgabe der Studienjahresarbeit war, eine Möglichkeit zur grafischen Darstellung von Unterwasserfahrzeugen und deren Umgebung zu ermöglichen.

Anfangs war eine direkte Integration in die Simulationssoftware *Matlab/Simulink* gewünscht. Dem Softwarepaket (Version R2006a) liegt u.a. der VRML-Editor *V-Realm Builder* bei, mit dessen Hilfe eine 3D-Visualisierung auf Basis des VRML Standards möglich ist. Es wurde festgestellt, daß diese Software nicht mit Matlab/Simulink kompatibel ist, obwohl sie auf den Matlab-Installations-CDs beiliegt. Zudem konnten die vorhandenen VRML-Modelle von Unterwasserfahrzeugen vom V-Realm Builder nicht geladen werden.

Ein anderer Ansatz war, Software zu verwenden, die für die Entwicklung von Computerspielen angeboten werden. Ein Vorteil dieser ist, daß sie grundlegende Elemente bereits beinhalten, beispielsweise Grafikausgabe, Nutzerinteraktion (Maus, Tastatur, Gamepad), 3D-Editoren, Textur-Editoren, Netzwerk-Kommunikation und Ton-Ausgabe. Besonders Level-Editoren wären für die Nachbildung von Unterwasserszenarien (z.B. Hafenanlagen) ein nützliches Werkzeug. Als Nachteil erwies sich, dass diese Entwicklungsumgebungen überwiegend proprietäre Datenformate verwenden. Dadurch ist ein Import und vor allem Export von Daten (z.B. erstellte Szenen) kaum möglich. Weiterhin stößt man sehr schnell an Grenzen, wenn Dinge benötigt werden, die nicht enthalten sind oder vorhandene Bestandteile nicht den Anforderungen entsprechen. Für die Zukunft sollte die Möglichkeit bestehen, die Visualisierungssoftware so zu erweitern, daß beispielsweise Sonarsensoren und Wasserstömungen simuliert werden und Kollisionen erkannt werden können. Artikel und Vergleiche von Spiele-Entwicklungsumgebungen[1] und Level-Editoren[2] sind zu finden in [2], [3], [4]. Einige in den Artikeln nicht genannte befinden sich unter [URL 11], [URL 12], [URL 13], [URL 14],

[1] 3D GameStudio, 3D Rad, 3Impact, Blitz3D, Cipher Engine, Dark Basic Pro, Q Engine, Torque Engine, Unity

[2] Far Cry Sandbox, DoomEdit, Half-Life 2 Hammer

[URL 15].
Tabelle 2.1 zeigt bereits vorhandene Softwareprodukte zur Unterwasserdarstellung. Diese wurden nicht verwendet, da sie kommerziell bzw. unverkäuflich sind. Das quelloffene Gazebo war zu Beginn der Arbeit in einem frühen Entwicklungsstatus.

Name	Beschreibung
Remotely Operated Vehicle Simulator	Kommerzielle Software zur Visualisierung von Unterwasserfahrzeugen inklusive Sonarsimulation [URL 18]
SubSim	Kommerzielles Simulationssystem für autonome Unterwasserfahrzeuge [URL 17]
Gazebo	Ein Simulator zur Visualisierung von Robotern in 3D-Umgebungen; enthält Methoden zur Physik und Sonarsimulation; frei verwendbare Software [URL 19]
AUV Framework	Eine Simulationsumgebung zur Darstellung von Unterwasserszenen und Steuerung autonomer Unterwasserfahrzeuge. Das AUV Framework wurde vom Fraunhofer IAIS im Auftrag der Bundeswehr entwickelt. [URL 20]

Tabelle 2.1: Simulationssoftware

Eine Alternative zu Entwicklungsumgebungen ist es, Softwarebibliotheken zu nutzen, die sich auf einen Kernbereich konzentrieren, beispielsweise Grafikausgabe. Diese können in den eigenen Quellcode integriert werden und sind flexibel verwendbar. Zudem besteht grundsätzlich die Möglichkeit, Bibliotheken mit verschiedenen Kernbereichen gemeinsam zu nutzen. Ein wesentlicher Nachteil besteht darin, daß es keine fertige Umgebung gibt (Objekteditoren, Oberfläche der Software mit Ein- und Ausgabemöglichkeiten, Ereignismanagement, etc.). Viele Dinge, die einem Spiele-Entwicklungsumgebungen abnehmen, müssen bei Verwendung von Bibliotheken eigens implementiert werden.

Dreidimensionale grafische Darstellungen basieren entweder auf dem *OpenGL* Standard oder der Programmierschnittstelle *Direct3D*[3]. Auch Grafikbibliotheken nutzen intern OpenGL oder Direct3D, erweitern diese jedoch um hilfreiche Funktionalitäten. Beispielsweise ist eine Ressourcenverwaltung für Objekte, Texturen[4], Materialien[5] und Objektanimationen sehr hilfreich. Viele mathematische Berechnungen sind ebenfalls

[3] *Direct3D* ist Bestandteil der Multimedia-Programmierschnittstelle *DirectX* des Herstellers Microsoft

[4] Als *Textur* bezeichnet man eine Grafik, die aus einer Grafikdatei geladen und auf einer Objektoberfläche in der 3D-Welt abgebildet wird.

[5] Als *Material* bezeichnet man Eigenschaften, die das Aussehen von Oberflächen definieren (Farbe, Reflexionen, Transparenz, etc.)

schon in einer Grafikbibliothek integriert, etwa sanft wirkende Bewegungen, verschiedene Schattenberechnungen oder Multi-Texturdarstellung mittels Alphakanal. Für Grafikbibiotheken gibt es in der Regel weitere Module, die sich integrieren lassen, z.B. zur Erzeugung von interaktiven Bedienelementen auf der Programmoberfläche.

Versuche mit der freien Grafikbibliothek *Irrlicht* [URL 16] zeigten, daß es grundsätzlich möglich ist, dreidimensionale Welten auf diese Weise darzustellen. Jedoch war die Dokumentation (Stand: Anfang 2007) unvollständig, wodurch es bei der Erstellung eines Softwareprototypen Probleme gab.

2.2 Verwendeter Softwareprototyp auf Basis der Grafikbibliothek Ogre

Ogre ist eine objektorientierte und quelloffene Grafikbibliothek [URL 30]. Sie besitzt einen großen Funktionsumfang, ist ausgereift, gut dokumentiert, wird beständig weiterentwickelt und u.a. für umfangreiche kommerzielle Spiele [URL 32] verwendet. Alle wichtigen Grundlagen zur Funktionsweise und Verwendung von Ogre wurden in einem Buch veröffentlicht [5].

Es existiert bereits eine unveröffentlichte Untersuchung im Fraunhofer Anwendungszentrum für Systemtechnik, bei der die Grafikbibliothek Ogre für die Darstellung von 3D-Szenen unter Wasser verwendet wurde. Als Ergebnis entstand ein Softwareprototyp, der auf den einführenden Ogre-Beispielen *Basic Tutorials* [URL 31] basiert. Er stellt eine einfache statische Szene dar, in der sich der Anwender mit der Kamera frei bewegen kann. Abbildung 2.1 zeigt den Prototyp.

Er verwendet den Wrapper *Mogre* [URL 34], der es ermöglicht, Ogre unter der Programmiersprache C# zu verwenden. Vorteil dieser rein objektorientierten Programmiersprache ist, dass man damit recht einfach Bedienoberflächen für Anwendungen erzeugen und das .NET Framework verwenden kann. Da .NET eine Vielzahl an nützlichen Klassen enthält, ist eine schnelle Entwicklung von Anwendungen möglich.

Für diese Studienjahresarbeit wurde der Prototyp von Stephan Zschäck als Grundlage verwendet und erweitert, um zu untersuchen, wie gut die Ogre-Bibliothek für die im Abschnitt 1.2 genannten Anforderungen geeignet ist.

Eigenschaften des Prototyps

Der verwendete Prototyp besaß bereits folgende Funktionalität:

- Initialisierung von Ogre und Einbindung einer Kameraansicht,
- Anzeige eines Geländes, das mit einem Geländegenerator erstellt wurde,
- Anzeige einer Wasseroberfläche und eines Schiffes,
- Möglichkeit zum Laden und Speichern von Szenen,
- Bewegungsmöglichkeit der Kamera mit anpassbarer Geschwindigkeit,
- Umschaltmöglichkeit auf andere, fahrzeuggebundene Kameras,
- Aktivierung von Schatteneffekten.

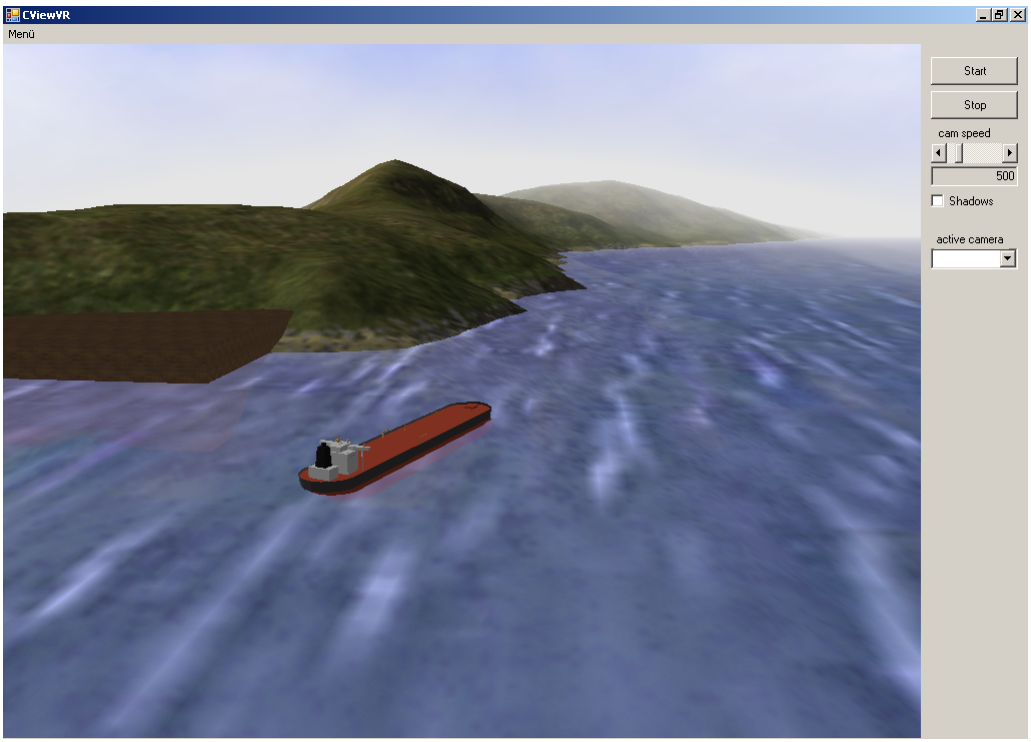


Abbildung 2.1: Grafische Oberfläche des Prototyps vom Fraunhofer AST

Szenen

Als Szene bezeichnet man eine dreidimensionale Umgebung in einer virtuellen Welt. Der verwendete Prototyp kann Szenen laden und speichern, die durch XML-basierte Dateien (mit der Dateiendung `*.cv2r`) beschrieben sind. Eine Szenendatei enthält in der Regel ein Gelände, Angaben zur Wasseroberfläche, sowie feste und bewegliche Objekte (z.B. Hafenmauer, Bäume, Schiffe, Unterwasserfahrzeuge). Neben der frei beweglichen Kamera können auch weitere, fahrzeuggebundene Kameras enthalten sein. Beim Speichern einer Szene wird die aktuelle Positionierung der frei beweglichen Kamera mitgespeichert, um beim späteren Laden die gleiche Ansicht zu erhalten.

Die Grafikbibliothek Ogre benutzt intern einen Szenengraph zur Beschreibung des Aufbaus einer Szene. Der Szenengraph ist eine Struktur, bei der die enthaltenen Elemente hierarchisch angeordnet sind. Für alle Objekte und deren Parameter, die in der Szenendatei definiert sind, werden Szenenknoten erstellt und diese in den Szenengraph von Ogre übertragen. Dazu zählen Position (bezüglich des Ogre-Koordinatensystems), Orientierung (Blickrichtung), anzuzeigendes 3D-Modell, etc. Aus dem Inhalt des Szenenmanagers berechnet Ogre die dreidimensionale Szene und daraus eine 2D-Projektion (angezeigtes Bild) bezüglich der Kameraeigenschaften.

Abbildung 2.2 (S. 13) zeigt den Szenengraph einer einfachen Beispielszene, die neben der frei beweglichen Kamera ein geladenes 3D-Modell und eine daran gekoppelte Kamera enthält. Das XML-Dokument (Szenendatei) dieser Beispielszene befindet sich im Anhang E (S. 102).

Details zum Aufbau einer Szenendatei befinden sich in Abschnitt 4.3 (S. 47). Dieser bezieht sich zwar auf die erweiterte Version, die während Studienjahresarbeit entstand, aber der grundsätzliche Dateiaufbau ist gleich.

Details zu den Koordinatensystemen befinden sich im folgenden Abschnitt. Die Einbindung von Gelände wird in 2.7 (S. 21) behandelt und die Integration von Objekten im Kapitel 3 (S. 24). Ausführliche Informationen zu Szenenknoten und zum Szenengraph befinden sich in [5].

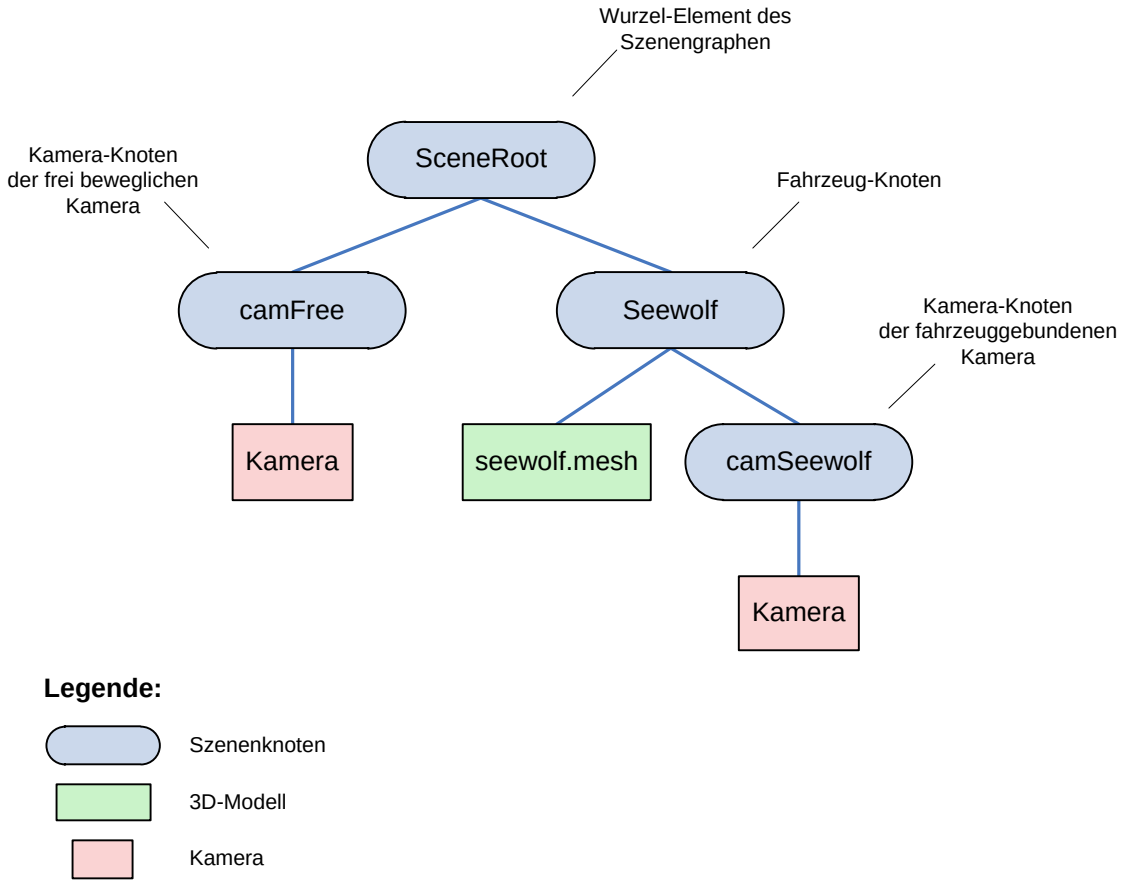


Abbildung 2.2: Szenengraph einer einfachen Beispielszene

2.3 Koordinatensysteme

Die Grafikbibliothek Ogre verwendet ein kartesisches Koordinatensystem, bei dem die Y-Achse nach oben zeigt. Eine Definition von Himmelsrichtungen gibt es nicht. Bei Einbindung von Gelände (Details in Kapitel 2.7, S. 21) befindet sich dieses im ersten Oktant des Koordinatensystems, wodurch horizontale Geländekoordinaten immer ein positives Vorzeichen besitzen. Der Koordinatenursprung liegt in der südwestlichen Ecke auf dem Niveau des tiefsten Punktes des Geländes.

Als Grundlage für die Steuerung von Fahrzeugen durch externe Simulatoren war ein kartesisches NED-Koordinatensystem zu benutzen, dessen Achsen in Richtung Norden, Osten und nach unten zeigen. Der Ursprung sollte sich auf Höhe der Wasseroberfläche befinden.

Für die Verwendung des NED-Koordinatensystems in Ogre mußte eine Transformation verwendet werden. Der Ursprung des NED-Koordinatensystems wurde in die selbe Ecke der Geländekarte gelegt wie das Ogre-Koordinatensystem. Dadurch besitzt nur die Höhenachse einen Translationsunterschied. Bezüglich der räumlichen Ausrichtung wurde für das Ogre-Koordinatensystem definiert, daß die X-Achse nach Norden, die Z-Achse nach Osten zeigt. Die Y-Achse zeigt nach oben. Weiterhin wurde definiert, daß eine Längeneinheit im Ogre-Koordinatensystem einem Meter entspricht.

Die Umrechnung zwischen Ogre- und NED-Koordinatensystem erfolgt durch Formel 2.1, wobei Δw den Abstand zwischen Wasserhöhe und tiefstem Punkt des Geländes angibt. Ein Vergleich der Koordinatensysteme veranschaulicht Abbildung 2.3 (S. 15).

$$\begin{pmatrix} x_{Og} \\ y_{Og} \\ z_{Og} \end{pmatrix} = \begin{pmatrix} x_{NED} \\ -z_{NED} + \Delta w \\ y_{NED} \end{pmatrix} \tag{2.1}$$

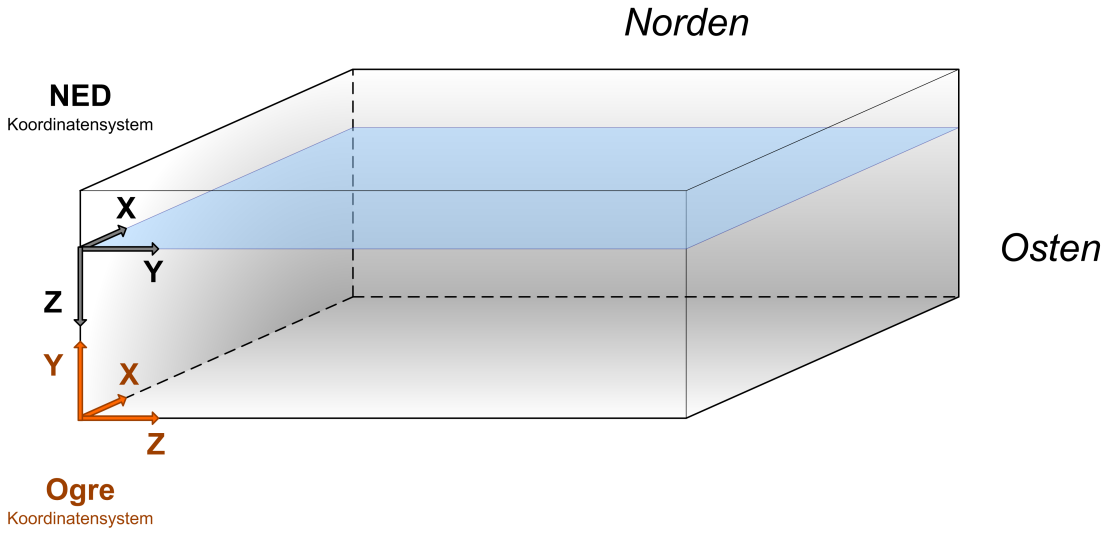


Abbildung 2.3: Vergleich der Koordinatensysteme

2.4 Latitude-Longitude-Koordinaten

Da es Fahrzeugsimulatoren gibt, die Fahrzeugkoordinaten in Form von Latitude-Longitude-Koordinaten ausgeben, ist es sinnvoll, eine entsprechende Koordinatentransformation in die Visualisierungsumgebung zu integrieren. Für globale Positionierungen werden üblicherweise Polarkoordinaten verwendet, die sich auf einen Ellipsoid beziehen, der die Form der Erde näherungsweise darstellt. Weit verbreitet ist die Verwendung des standardisierten *WGS 84* Ellipsoiden, der u.a. in der Luftfahrt verwendet wird. Zur Transformation ist es sinnvoll, die Polarkoordinaten erst in das kartesische Koordinatensystem *Earth Centered Earth Fixed* (ECEF) umzurechnen und anschließend in ein kartesisches *North-East-Down* Koordinatensystem (NED) mit lokalem Bezugspunkt. Die gewölbte Erdoberfläche ist dann auf einer flachen Ebene abgebildet, was räumliche Verzerrungen zur Folge hat. Umrechnungen mit hoher Genauigkeitsanforderung sind aufwendig. Zudem gibt es unterschiedlich berechnete Referenzkarten, die Ausschnitte der Erdoberfläche planar abbilden. Ein gut verständlicher Artikel über die Form der Erde, Kartenbezugssysteme, Geoide und Ellipsoide ist nachzulesen unter [URL 51]. Ausführliche wissenschaftliche Informationen zur Geologie findet man beim U.S. Geological Survey unter [URL 52]. Frei verwendbare Softwarebibliotheken und Quellcodes zur Umrechnung sind u.a. bei [URL 53], [URL 54] und [URL 55] zu finden.

Für diese Studienjahresarbeit wäre eine Implementierung dieser Zusammenhänge zu umfangreich gewesen. Statt dessen wurde eine vereinfachte Latitude-Longitude umgesetzt. Mehr dazu steht in Kapitel 4.6 (S. 58).

2.5 3D-Modelle und Rotationsvorschriften

Für die Visualisierung im Rahmen der Studienjahresarbeit waren die 3D-Modelle so auszurichten, daß die Vorderseite – falls vorhanden – nach Norden ausgerichtet ist, wenn alle Rotationswinkel gleich Null sind. Für die räumliche Ausrichtung in eine beliebige Richtung waren Eulerwinkel bezüglich des NED-Koordinatensystems zu benutzen. Dabei war zu beachten, die einzelnen Rotationen (*yaw*, *pitch*, *roll*) nach einer bestimmten Konvention anzuwenden. Diese entspricht der Luftfahrtnorm DIN 9300 und kann mathematisch als Rotation um die Achsen Z, Y', X" bezüglich des NED-Koordinatensystems beschrieben werden. Die erste Rotation (yaw) ist der Gierwinkel (bzw. Steuerkurs) und beschreibt die Ausrichtung entsprechend der Himmelsrichtung. Hierbei entspricht 0° der Richtung Norden; bei $yaw = 90°$ zeigt das Fahrzeug Richtung Osten. Die zweite Rotation (Pitch) bezieht sich auf das körperfeste Koordinatensystem des Fahrzeuges und beschreibt das Nicken. Bei positiver Rotation hebt sich der Bug. Bei der dritten Rotation (Roll) wird das Fahrzeug um seine körperfeste Längsachse rotiert und vollzieht bei positiver Drehung eine seitliche Kippbewegung nach rechts.

Abbildung 2.4 (S. 17) zeigt ein nach Norden ausgerichtetes Fahrzeug und kennzeichnet die Rotationswinkel entsprechend der Koordinatensysteme NED und Ogre. Die bevorzugten Wertebereiche sind in (2.2) zu sehen.

$$\begin{array}{lll} -180° < yaw \leq 180° & -90° \leq pitch \leq 90° & -180° \leq roll \leq 180° \\ -\pi < yaw \leq \pi & -\frac{\pi}{2} \leq pitch \leq \frac{\pi}{2} & -\pi \leq roll \leq \pi \end{array} \tag{2.2}$$

Aufgrund der unterschiedlichen Systemausrichtung sind die oben genannten Rotationsvorgaben *nicht* direkt auf das Ogre-Koordinatensystem anwendbar. Auch die Bedeutung von *Pitch* und *Roll* ist im Ogre-Koordinatensystem eine andere, da sich die Begriffe auf einen anderen Initialzustand beziehen.
In (2.3) ist die Übertragung der Rotationen vom NED-Koordinatensystem in das Ogre-Koordinatensystem zu sehen.

$$\begin{array}{rcl} yaw_{NED} & \Longrightarrow & -\,yaw_{Og} \\ pitch_{NED} & \Longrightarrow & roll_{Og} \\ roll_{NED} & \Longrightarrow & pitch_{Og} \end{array} \tag{2.3}$$

Rotationskonvention: $Y_{Og}, Z'_{Og}, X''_{Og}$

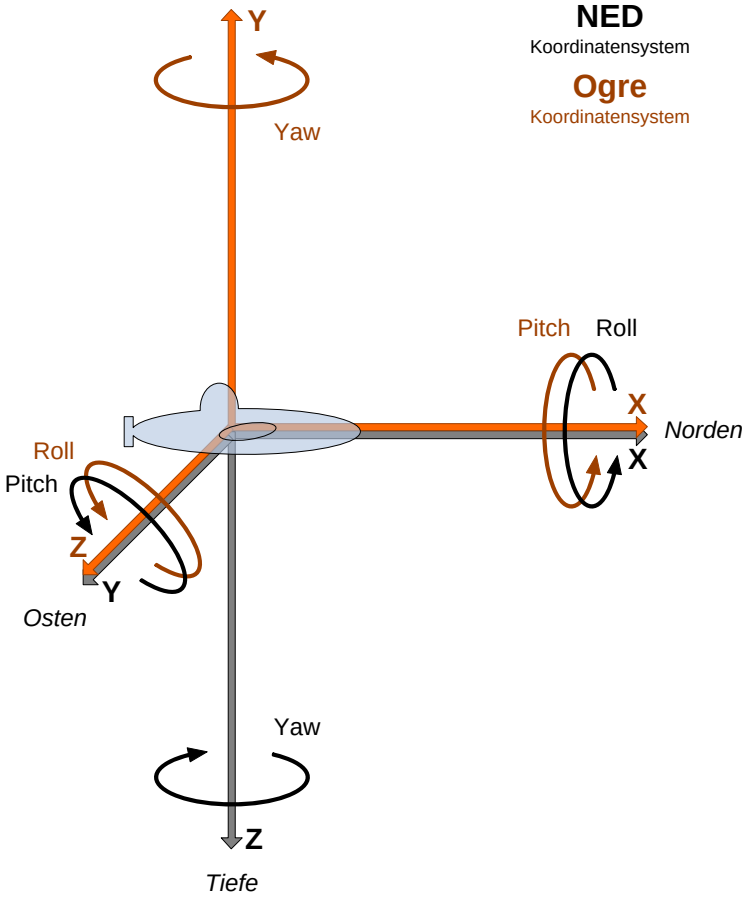


Abbildung 2.4: Rotationswinkel (NED, Ogre) für Fahrzeuge

Abgesehen von verschiedenen Konventionen zur Rotation können bei Verwendung von Eulerwinkeln Probleme mit *Singularitäten* auftreten, die den unerwünschten *Gimbal Lock*[6] verursachen. Intern verwendet die Ogre-Bibliothek *Quaternionen* zur Beschreibung der räumlichen Lage über 4 reelle Parameter. Durch die Verwendung von Quaternionen tritt das Problem des Gimbal Lock nicht auf, Berechnungen werden vereinfacht und bei vielen kaskadierten Rotationen sind die Rundungsfehler deutlich geringer als bei der Verwendung von Eulerwinkeln. Die vorhandenen Fahrzeugsimulatoren im Fachgebiet Systemanalyse der TU Ilmenau und im Fraunhofer AST können Lagebeschreibungen nur in Form von Eulerwinkeln ausgeben. Daher war für die Visualisierungssoftware keine Datenübertragung von Quaternionsangaben vorgesehen. Dennoch wäre es möglich, die entstandene Software CViewVR so zu erweitern, daß Fahrzeuge optional über Quaternionsparameter gesteuert werden. Verschiedene Konzepte zur Erweiterung sind in Kapitel 4.4.2 (S. 54) beschrieben.

Seitens der Visualisierungssoftware kann kein Gimbal Lock entstehen, da die vom Fahrzeugsimulator empfangenen Eulerwinkel absolute Werte sind, die sich auf die Grundausrichtung des Fahrzeuges beziehen. Anders wäre es, wenn relative Rotationswerte übertragen werden würden, welche sich auf die vorherige Fahrzeugorientierung beziehen.

Die Winkel *yaw*, *pitch* und *roll* werden mit griechischen Buchstaben bezeichnet. (Dies entspricht DIN 13312, einer Norm bezüglich der Navigation, in der Begriffe, Abkürzungen, Formelzeichen und graphische Symbole festgelegt sind.) Tabelle 2.2 zeigt eine Übersicht.

Bezeichnung	Symbole		Winkelname	Bedeutung
yaw	Ψ	ψ	Gierwinkel	Himmelsrichtung
pitch	Θ	θ	Nickwinkel	Bug heben/senken
roll	Φ	ϕ	Wankwinkel	Seitliche Drehung

Tabelle 2.2: Winkelbezeichnungen nach DIN 13312

[6] Der *Gimbal Lock* ist ein geometrisches Problem, welches in Verbindung mit Eulerwinkeln auftreten kann. In einem kritischen Bereich liegen zwei Rotationsachsen in einer Ebene und es erfolgt eine Art Blockierung, wodurch das zu rotierende Objekt nicht die gewollte räumliche Ausrichtung erreichen kann. Ein anschauliches Video dazu befindet sich unter: `www.youtube.com/watch?v=rrUCBOlJdt4`

2.6 Schnittstelle zu externen Simulatoren

Fahrzeuge, die in der Visualisierung dargestellt sind, sollen sich durch Fahrzeugsimulatoren steuern lassen. Solche gibt es bereits als eigenständige Software oder eingebettet in Simulationsumgebungen wie Matlab/Simulink. Sie enthalten in der Regel mathematisch/physikalisch beschriebene Modelle von Fahrzeugen und Umgebungen, die deren reales Verhalten nachbilden. Fahrzeugbewegungen entstehen durch interaktive Steuereingaben von Nutzern oder durch eine automatische Steuerlogik, die versucht, bestimmte Aufgaben (sogenannte Missionspläne) zu erfüllen. Ebenso können simulierte Umwelteinflüsse wie Strömungen eine Rolle spielen.

Um die Fahrzeugbewegungen in der Visualisierungssoftware darzustellen, muß eine Kommunikation zwischen Simulator und Visualisierung stattfinden. Da die Kommunikation auch rechnerübergreifend stattfinden soll, ist es sinnvoll, ein programmiersprachen- und plattformunabhängiges Netzwerkprotokoll zu verwenden. Weit verbreitet und zuverlässig sind die Protokolle TCP und UDP. Viele Programmiersprachen enthalten Funktionen oder Klassen, die die Verwendung dieser Protokolle einfach machen. Auch für Matlab/Simulink gibt es entsprechende Module.

Beide Protokolle ermöglichen eine gleichzeitige Anbindung von mehreren Simulatoren. Abbildung 2.5 zeigt das Prinzip, wie Fahrzeugsimulatoren mit der Visualisierung kommunizieren. Die Simulatoren übertragen in regelmäßigen Intervallen Zustandsdaten wie Position und räumliche Orientierung. Ein optionaler Rückkanal ermöglicht Feedbacks an die Simulatoren; beispielsweise Szeneninformationen, Fehlermeldungen, Sensordaten und Kollisionsereignisse.

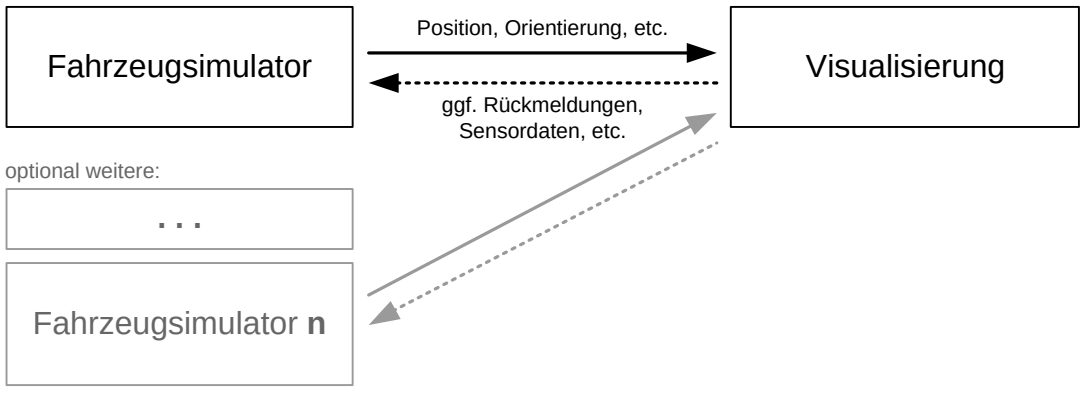


Abbildung 2.5: Kommunikationsprinzip

TCP vs. UDP

Ungünstig bei TCP ist der Sendepuffer, durch den kleine Datenmengen nicht sofort gesendet werden. Statt dessen werden sie erst zwischengespeichert und dann gemeinsam in einem festen Zeitintervall übertragen (alle 0,2 Sekunden). Das führt zu einem starken Ruckeln von bewegten Objekten. Diese grundsätzlich sinnvolle Protokolleigenschaft läßt sich abschalten, indem man die Socketoption *nodelay* auf *true* setzt. Danach funktioniert die Übertragung wie gewünscht. Bei hoher Netzlast wird die Datenrate durch das TCP-Protokoll stark gedrosselt. Das alternative Protokoll UDP ist für Echtzeitanwendungen konzipiert und verhält sich anders. Alle Pakete werden unverzögert weitergeleitet. Es wird in Kauf genommen, daß bei Netzüberlastung ein Teil der Pakete verworfen, also beim Transport vernichtet werden. Im Gegensatz zu TCP ist das UDP-Protokoll nicht verbindungsorientiert. Es gibt keinen Rückkanal und die richtige Reihenfolge der Datenpakete ist nicht garantiert.

Das Problem der TCP-Geschwindigkeitsdrosselung spielt bei der Übertragung im lokalen Netzwerk keine Rolle, da die nötige Datenrate deutlich geringer ist als die Übertragungsgeschwindigkeit heutiger Netzwerkhardware. Bei Verbindungen über das Internet sind Störungen möglich (Verzögerungen, weniger Durchsatz).

Rechenbeispiel: Es werden 20 Objekte gesteuert, die alle 10 ms ein Zustandsaktualisierung bekommen. Das ergibt (inklusive 10% Protokolloverhead) eine Datenrate von ca. 0,2 MByte/s. Eine 1000 MBit Netzwerkverbindung bietet eine Übertragungsgeschwindigkeit von ca. 12 MByte/s.

Ein möchlicher Nachteil von UDP ist, daß die Reihenfolge der empfangenen Datenpakete nicht sichergestellt ist. Ältere Pakete kann man zwar verwerfen, aber dazu müsste zur Reihenfolgeermittlung eine Art Zeitstempel integriert werden. Zusätzlich müßte in Abhängigkeit dazu eine Senderidentifikation stattfinden, da bei Verwendung mehrerer Clients davon auszugehen ist, daß deren Zeitbasis nicht synchron ist.

Grundsätzlich lassen sich für die Verbindung zwischen Simulator und Visualisierung beide Protokolle verwenden. Abgesehen von der Geschwindigkeitsdrosselung bei hoher Netzlast und der (abschaltbaren) Sendeverzögerung erschien das TCP-Protokoll als geeigneter. Da es verbindungsorientiert ist, wird der Verarbeitungsaufwand geringer. Der Rückkanal ermöglicht die Übertragung von Statusmeldungen (z.B. Warnungen, Infos) und bei zukünftiger Erweiterung der Visualisierungssoftware können simulierte Sensordaten (z.B. Sonar, Kollisionen) zurück an den Simulator gesendet werden. Daher wurde festgelegt, für die Visualisierung das TCP-Protokoll zu verwenden. TCP-Verbindungen haben eine etwas größere Latenz (Verzögerungszeit) bei der Übertragung als UDP, doch Verzögerungen von etwa vier bis zwölf Millisekunden in lokalen Netzwerken sind für

die Visualisierung von Fahrzeugbewegungen unbedeutend. Eine spätere Erweiterung der Software für das UDP-Protokoll ist grundsätzlich möglich.

Alternativ wäre die Verwendung von CORBA (common object request broker architecture) möglich. Bei CORBA wird Software von Drittanbietern verwendet, die Protokolle und Dienste zur Kommunikation anbietet. Dies soll eine Interaktion von Software – unabhängig von Hardware, Betriebssystem und Programmiersprache – ermöglichen. Für den Einsatzzweck in der Visualisierungssoftware war kein wesentlicher Vorteil gegenüber TCP zu erkennen.

Details zur Implementierung des auf TCP basierenden Übertragungsprotokolls befindet sich in Kapitel 4.4 (S. 49).

Aktualisierungsintervall

Für eine flüssige Bewegungsdarstellung müssen die Fahrzeugzustände in sehr kurzen Intervallen übertragen werden. Bei zu langen Intervallen wirken Bewegungen ruckelig. Laut Angaben der Fachzeitschrift *c't* sind Bewegungen am Bildschirm ab 25 Aktualisierungen pro Sekunde (fps) akzeptabel und wirken ab 40 fps flüssig [1]. Bei Fahrzeugen und anderen gesteuerten Objekten, die sich nur langsam bewegen (relativ zur Kameraperspektive), fallen längere Zeitintervalle weniger auf. Bei unnötig kurzen Übertragungsintervallen werden nutzlos mehr CPU- und Netzwerkressourcen beansprucht.

Empfehlung: 50 bis 100 Aktualisierungen pro Sekunde, was einem Intervall von 10 bis 20 Millisekunden entspricht.

2.7 Gelände einbinden

Gelände kann in Form von Rastergrafiken importiert werden. Ein Grauwertbild repräsentiert die Höheninformation. Hohes Gelände wird durch helle Grauwerte dargestellt und tiefliegendes Gelände durch dunkle Pixelwerte. Der in Ogre integrierte Terrain Scene Manager [URL 56] generiert daraus ein 3D-Modell. Aus einer weiteren Grafikdatei wird eine Geländetextur geladen und senkrecht von oben auf das 3D-Geländemodell projeziert. Abbildung 2.6 zeigt die Grafiken eines Geländes, das durch die Software L3DT [URL 58] erstellt wurde.

Betrachtet man Gelände aus einer kurzen Distanz, so wirkt es unscharf und unrealistisch. Für eine detailliertere Darstellung müßte die Auflösung um ein Vielfaches größer sein. Das ist in der Praxis schlecht machbar. Statt dessen kann zusätzlich eine pseudozufällige Struktur fein aufgelöst über die Geländetextur gelegt werden. Dadurch werden dem Gelände viele kleine abgedunkelte Stellen hinzugefügt, was einen deutlich

realistischeren Eindruck macht. Abbildung 2.7 zeigt einen Vergleich von Gelände mit und ohne Überlagerung durch die sogenannte Detailtextur, welche in Abbildung 2.8 zu sehen ist. Grundsätzlich können bei Ogre mittels der Alphakanaltechnik mehrere Texturen (z.B. Gras, Erde, Schotter) teiltransparent überlagert werden, um Gelände oder andere Objektoberflächen zu gestalten. Darauf geht diese Arbeit jedoch nicht ein.

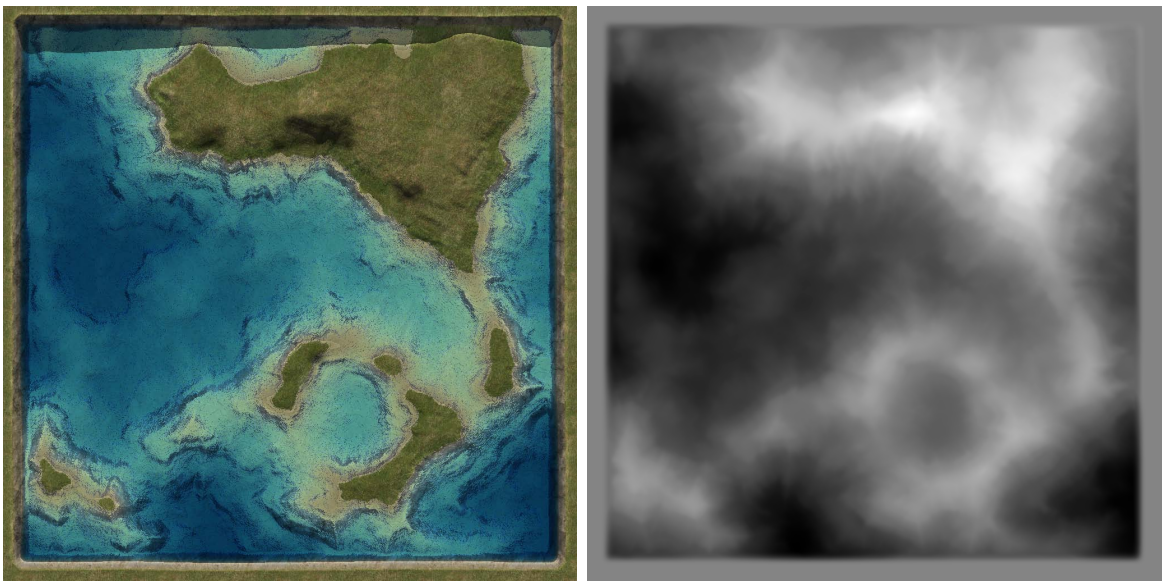


Abbildung 2.6: Gelände-Textur und Höhenkarte

Bei der Erstellung von Geländetextur- und Höhengrafiken ist zu beachten, daß sich die Himmelsrichtung **Norden auf der rechten Seite** befindet, statt wie allgemein üblich auf der oberen Seite. Grund dafür ist die unterschiedliche Ausrichtung vom Ogre- und NED-Koordinatensystem.

Geländeparameter werden in speziellen Konfigurationsdateien hinterlegt. Beim verwendeten Prototyp des Fraunhofer AST stehen die Parameter in Datei `terrain1.cfg`.

Mehr Details zum *Terrain Scene Manager* ist zu finden unter [URL 56]. Alternativ stehen weitere Szenenmanager (u.a. für Gelände) zur Verfügung [URL 57]. Gelände kann durch die Software *Large 3D Terrain Generator (L3DT)* [URL 58] generiert werden. Eine Übersicht weiterer Programme zur Erstellung von Gelände befindet sich in [URL 60].

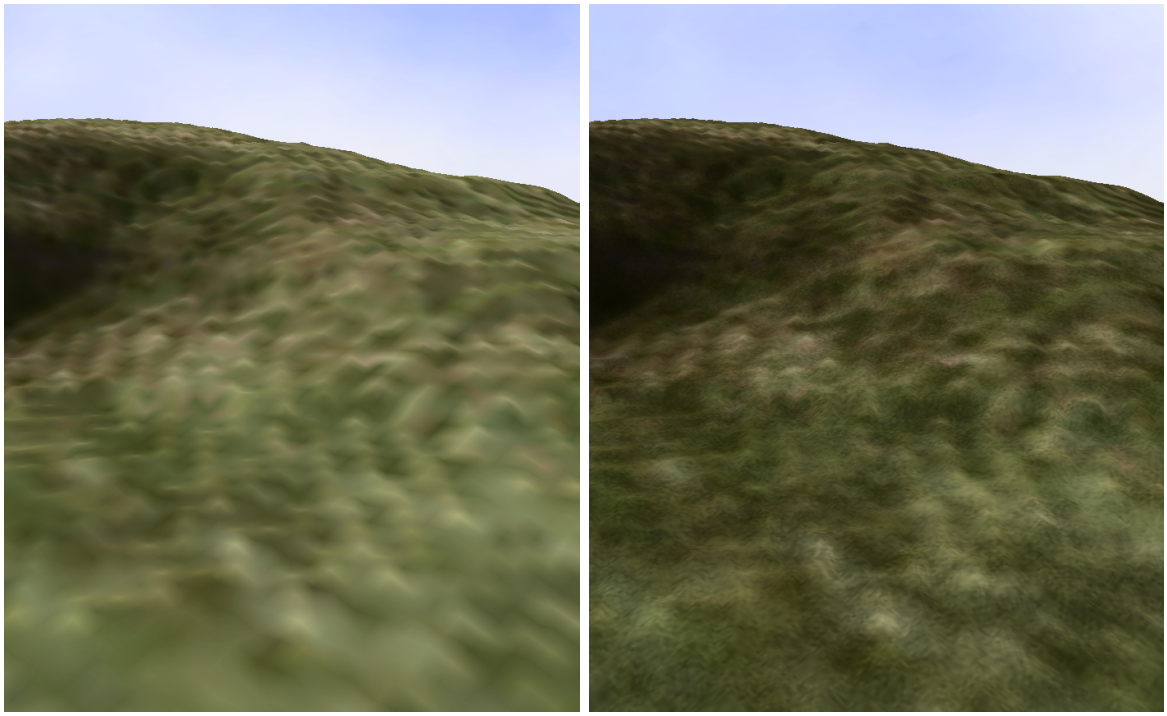


Abbildung 2.7: Gelände ohne/mit Detail-Textur

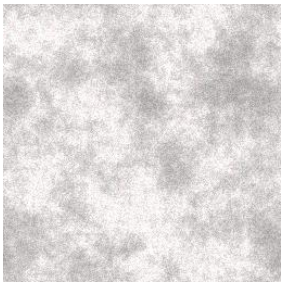


Abbildung 2.8: Detail-Textur

3 Integration von 3D-Modellen

Im Fachgebiet Systemanalyse der TU Ilmenau wird an Unterwasserfahrzeugen und deren teils autonomen Steuerung geforscht. 3D-Modelle einiger Fahrzeuge lagen im Dateiformat VRML vor und sollten in die Visualisierungssoftware integriert werden. Auch die Integrationsmöglichkeit anderer Modellformate sollte untersucht werden.

3.1 Modelle in Ogre

Die zur Visualisierung verwendete Grafikbibliothek Ogre kann ausschließlich 3D-Modelle laden, die im eigenen Format *Ogre Mesh* hinterlegt sind. Um Modelle aus anderen Dateiformaten einzubinden, werden für einige verbreitete 3D-Modellierungsprogramme Export-Module von den Ogre-Entwicklern und Dritten zur Verfügung gestellt [URL 38]. Durch diese können 3D-Modelle im Ogre-Format gespeichert werden.

Das Modellformat Ogre Mesh gibt es in XML- und Binärform mit den Dateiendungen `*.mesh.xml` und `*.mesh`. Die XML-Variante enthält die Modellinformation in einer standardisierten Datenbeschreibungssprache und kann von anderen Programmen und Export-Plugins vergleichsweise einfach erstellt oder ausgelesen werden. Ogre selbst kann Modelle nur in der proprietären Binärform laden. Zur Erstellung der Binärform diehnt der *OgreXMLconverter* [URL 61]. Dieser kann Binärdateien auch wieder in XML-Dateien konvertieren. Ogre Mesh-Dateien enthalten die dreidimensionale Modellstruktur, aber keine Materialeigenschaften zur Definition vom Aussehen der Objektoberflächen (z.B. Farbe, Reflexionseigenschaften, Texturen).

Materialeigenschaften werden in separaten Dateien mit der Endung `*.material` gespeichert. Die Dateinamen sind unabhängig von den Modellnamen und können beliebig gewählt werden. Eine Materialdatei kann auch Materialdefinitionen für mehrere Modelle beinhalten. Die Zuordnung der Definitionen erfolgt über eindeutige Materialnamen. Weiterhin können bei Materialdefinitionen auch Texturen eingebunden werden. Texturen sind Grafiken, die auf Modelloberflächen projiziert werden, um sie realistischer wirken zu lassen. Die Grafiken sind in separaten Dateien gespeichert und stellen beispielsweise Mauerwerk, Rasen, Holz, Gestein, Korrosionen, diverse Strukturen oder

auch Gesichter von menschlichen Modellen dar.

Ogre enthält einen Ressourcenmanager, der alle vorhandenen „Ressourcen“ (u.a. Mesh-, Material-, Texturdateien) einliest und deren Inhalte verwaltet. Für den Anwendungsentwickler ist es dadurch sehr einfach, Modelle in eine Szene einzubinden. Der Ressourcenmanager kümmert sich automatisch um die Zuordnung der Materialdefinitionen zu den entsprechenden Modellen. Abbildung 3.1 zeigt das Prinzip, wie 3D-Modelle und deren Oberflächendarstellung (Material und Texturen) in Ogre verwaltet wird.

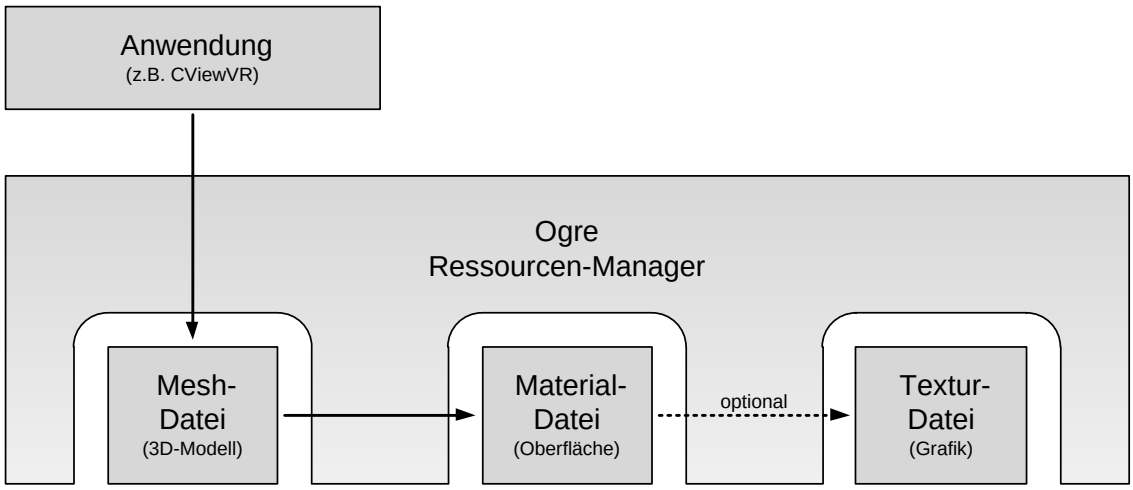


Abbildung 3.1: Verwaltung von Modellen, Materialien und Texturen in Ogre

3.2 VRML

Es gibt viele Formate zur Speicherung von 3D-Objekten. Die meisten sind proprietär und werden nur von wenigen Programmen unterstützt. VRML ist eine verbreitete, standardisierte Beschreibungssprache, mit deren Hilfe man 3D-Modelle von Objekten inklusive Texturen und Animationen beschreiben kann. Objekte im VRML-Format der Version 1.0 werden von vielen Programmen unterstützt. Der erweiterte Standard VRML 2.0 (alias VRML 97) wird den Recherchen zufolge kaum von freier Software unterstützt. Einige kostenlose Programme konnten die zur Verfügung gestellten Fahrzeugmodelle zwar öffnen, aber nur fehlerhaft darstellen. Möglicherweise wird statt dessen der neuere Standard X3D besser unterstützt. Für Ogre gibt es einen quelloffenen VRML-Konverter[URL 39], der laut Hilfedatei VRML 97-Dateien in das Ogre-Format überführen kann, was jedoch bei den verwendeten Modellen nicht funktionierte. Die Positionsangaben der einzelnen Modellbestandteile ging verloren und nach dem Import existierte nur noch ein „wilder Teilehaufen“. Die letzte Aktualisierung des Importers war im Januar 2004.

3.3 Arbeitsablauf allgemein

Zur Speicherung von Modellen im Ogre Mesh-Format stehen verschiedene Exporter zur Verfügung [URL 38]. Von den unterstützten 3D-Modellierungsprogrammen ist *Blender* jedoch das einzige kostenlose. Es handelt sich dabei um eine ausgereifte, quelloffene und vielseitige Software. Sie kann bereits vorhandene Modelle aus verschiedenen Formaten laden. Darunter auch VRML 1.0, jedoch nicht VMRL 97.

Dennoch wurde ein Weg gefunden, die vorhandenen Fahrzeugmodelle in die Visualisierungssoftware zu integrieren. Das Programm *AccuTrans3D* [URL 27] ermöglicht eine Konvertierung zwischen diversen 3D-Formaten. Es steht als Testversion kostenlos zur Verfügung und kann ohne Einschränkung unbegrenzt lange genutzt werden (wobei laut Lizenz nur 30 Tage Testzeit erlaubt sind). Konvertiert man Modelle mit *AccuTrans3D* in das Format `*.3ds` (3D Studio Max), so können diese dann von *Blender* geladen werden.

Aufgrund der Vorgaben für das später verwendete NED-Koordinatensystem müssen in Blender einige Modellanpassungen gemacht werden, ebenso bei bestimmten Modelleigenschaften:

- Die räumliche Orientierung des Fahrzeugmodells muß korrekt am Koordinatensystem ausgerichtet werden. (siehe Abbildung 3.2 auf Seite 28)
- Für eine korrekte Darstellung von Fahrzeugrotationen muß sich der Ursprung des modellbezogenen Koordinatensystems in der Fahrzeugmitte befinden. Ist das nicht der Fall, so muß das Fahrzeugmodell entsprechend verschoben werden.
- Eine Überprüfung der Maße (Länge, Breite, Höhe) ist empfehlenswert. Gegebenfalls muß der Maßstab durch Skalierung angepaßt werden.
- Redundante Materialdefinitionen sind zu entfernen. Etwa wenn für jede einzelne Teilfläche eine separate Materialdefinition besteht, obwohl die Eigenschaften (z.B. Farbe) gleich sind. Statt dessen sollte jedes Material einmalig definiert und den entsprechenden Flächen zugewiesen sein. Abbildung 3.3 (S. 28) zeigt das Zuordnungsprinzip zwischen Flächen und Materialien.
- Sollte das geladene Modell (dateiintern) aus seraraten Teilobjekten bestehen (z.B. Schornstein, Schiffschraube), so müssen diese Einzelteile zu einem Gesamtobjekt zusammengefaßt (verschmolzen) werden. Von außen nicht sichtbare Bestandteile (z.B. Motor) können entfernt werden.
- Jedes Modell darf maximal 15 verschiedene Materialien enthalten. Überzählige werden bei der Zusammenfassung von Teilobjekten verworfen. (Für zukünftige Versionen von Blender ist geplant, mehr Materialien pro Modell zu ermöglichen.)

Anschließend können die Modelle im Ogre-Format gespeichert werden. Nachdem die erzeugten Mesh- und Materialdateien in die Ressourcenverzeichnisse der Visualisierungssoftware kopiert wurden, können sie in Szenen eingebunden und dargestellt werden. Der allgemeine Ablauf zur Integration diverser Modellformate ist in Abbildung 3.4 (S. 29) veranschaulicht.

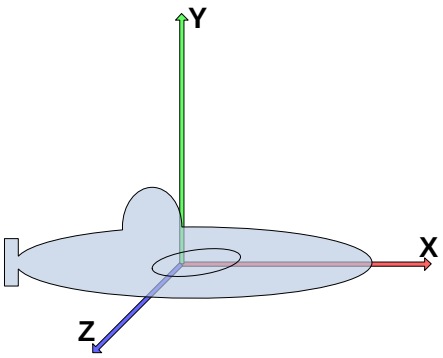


Abbildung 3.2: Notwendige Ausrichtung eines Fahrzeuges entsprechend der Koordinatenachsen (bezüglich Blender- und Ogre-Koordinatensystem)

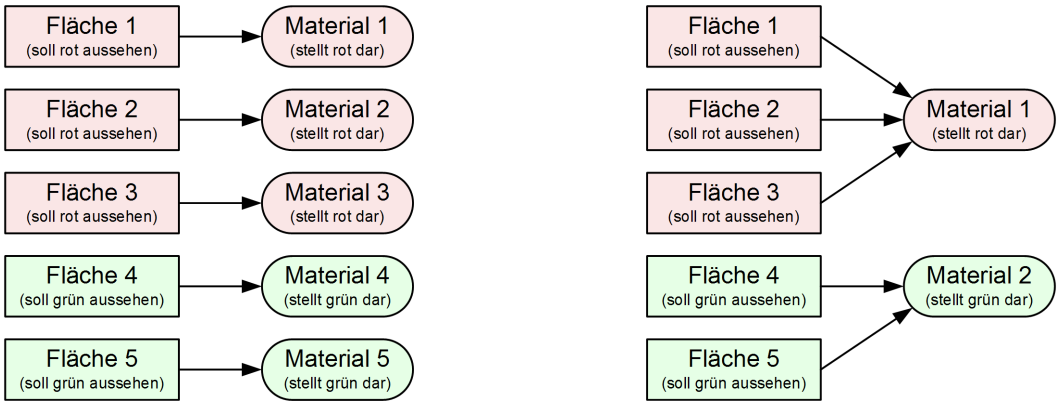


Abbildung 3.3: Prinzip der Zuordnung von Modelloberflächen und deren Materialeigenschaften (u.a. Farbe) .
links: redundante Materialdefinitionen (unerwünscht) .
rechts: gemeinsam genutzte Definitionen

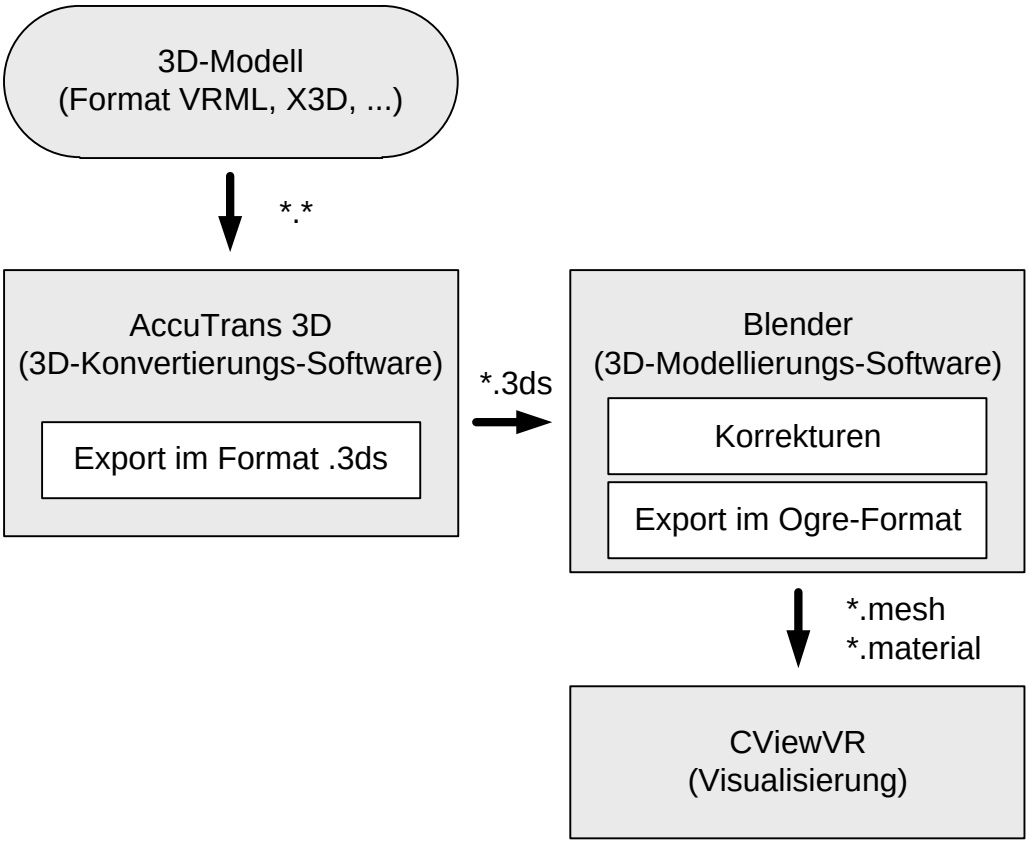


Abbildung 3.4: Arbeitsablauf zur Integration von 3D-Modellen

3.4 Korrekturen in Blender

Blender ist ein kostenloses und umfangreiches Programm zur Erstellung und Modifikation von 3D-Objekten. (Damit wurden sogar komplette Spiele [URL 45] und Animationsfilme [URL 46] erstellt.) Eine gute Einführung in das ungewöhnliche Bedienkonzept bietet der *Blender Anfänger Guide* [URL 43]. Weitere nützliche Information über die 3D-Modellierungssoftware befindet sich auf der Blender-Website [URL 42] und im *Blender Wikibook* [URL 44]. Sehr hilfreich ist auch *Das Blender-Buch* [7].

Im Ogre-Wiki ist ein grober, jedoch unvollständig beschriebener Ablauf zur Einbindung von Objekten in Ogre-Anwendungen mittels Blender zu finden [URL 41]. Zudem sind aufgrund der Vorgaben des NED-Koordinatensystems und der Szenenverwaltung des Visualisierungsprogrammes CViewVR besondere Anpassungen notwendig.

Vorbereitung

Bevor man mit Blender Modelle bearbeiten und im Ogre Mesh-Format speichern kann, müssen einige Vorbereitungen getroffen werden. Blender ist downloadbar auf `www.blender.org`. Nach der Installation muß zusätzlich die Skriptsprache Python [URL 47] installiert werden, da sonst einige Erweiterungen nicht funktionieren. Dabei ist darauf zu achten, nicht die neueste Python-Version zu installieren, sondern die gleiche, auf der die Blender-Installation basiert. Die Versionsnummer wird beim Programmstart angezeigt.

Um Erweiterungen in Blender zu integrieren, müssen die entsprechenden Dateien in das Unterverzeichnis `.blender\scripts` der Blender-Installation kopiert werden (z.B. `C:\Programme\Blender Foundation\Blender\.blender\scripts`). Bei Windows Vista und 7 müssen sie stattdessen (aufgrund des Sicherheitskonzepts) in das Verzeichnis `C:\Users\<name>\AppData\Roaming\Blender Foundation\Blender` ... ... `\.blender\scripts` kopiert werden. Hinzugefügte Erweiterungen werden dann automatisch von Blender geladen und dessen Funktionalität zur Verfügung gestellt.

Der *Blender Exporter* (alias *Ogre Meshes Exporter*) ermöglicht die Speicherung von Modellen im Ogre Mesh-Format und erzeugt die zugehörigen Materialdateien. Informationen und Downloadlinks befinden sich in [URL 63]. Der Blender Exporter besteht aus dem Skript `ogremeshesexporter.py` und zwei Verzeichnissen, die ebenfalls in `.blender\scripts` eingefügt werden müssen.

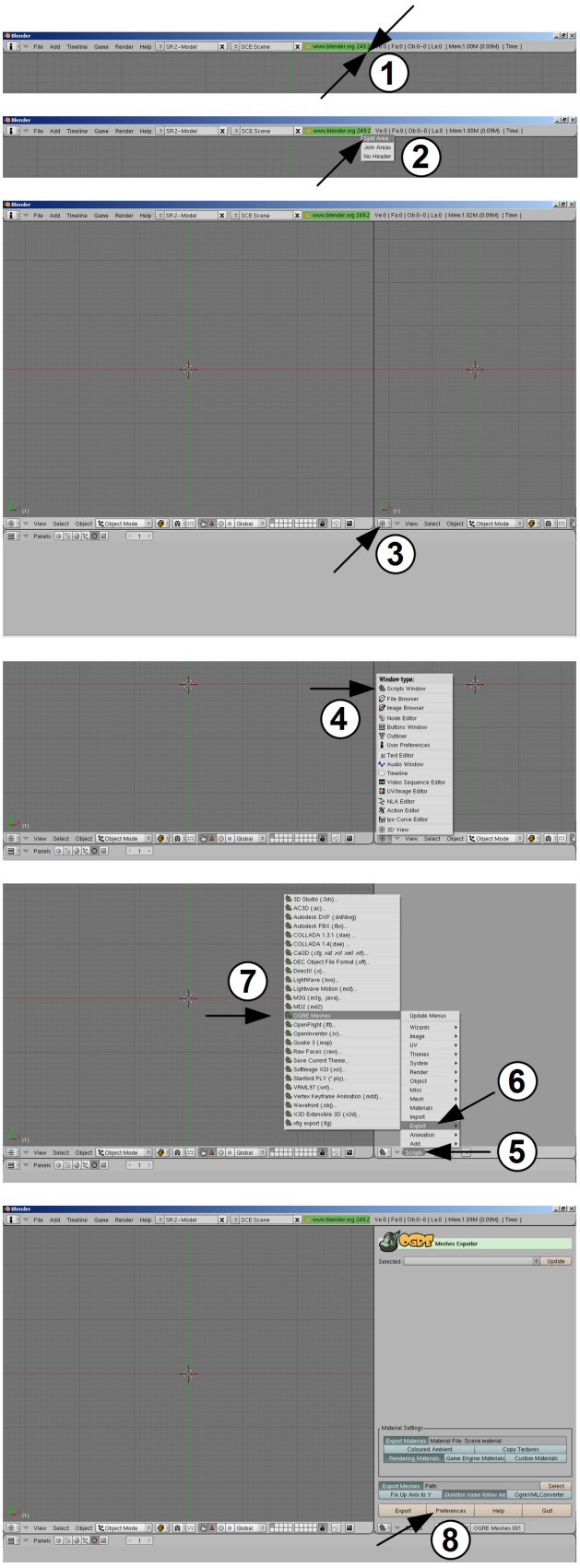
Zusätzlich muß der *OgreXMLConverter* [URL 61] zur Verfügung stehen. Er ist Bestandteil der *OgreCommandlineTools* [URL 62], die separat zu installieren sind.

Abbildung 3.5 (S. 32) zeigt, wie in Blender ein neues Teilfenster erzeugt und darin der Blender Exporter aufgerufen wird. Anschließend muß im Exporter über *Preferences* der erweiterte Konfigurationsdialog aufgerufen und bei *Location* der Pfad zum *OgreXMLConverter* eingetragen werden. Sollte kein Eingabefeld vorhanden sein, so muß erst auf *Manual* geklickt werden.

Die Erweiterung *Material Works* bietet hilfreiche Funktionen zur Verwaltung von Materialien, beispielsweise eine komfortable Umbenennung von Materialnamen. Dies kann nötig sein, um Konflikte mit anderen Modellen zu vermeiden, in denen möglicherweise gleichnamige Materialien vorkommen. Auch die Entfernung redundanter Materialdefinitionen wird durch Material Works erleichtert. Mehr Information gibt es in [URL 48].

Zur Installation wird die Datei `material_works.py` heruntergeladen und ebenfalls im Verzeichnis `.blender\scripts` gespeichert.

Hinweis: Anhang B (S. 85) beschreibt die einzelnen Bearbeitungsschritte in Blender, um die notwendigen Korrekturen von 3D-Modellen und deren Export in das Ogre durchzuführen.



1 Trennlinie klicken

2 *Split Area*

3 Symbol für Fenstertyp klicken

4 *Scripts Window* auswählen

5 *Scripts* klicken

6 *Export* klicken

7 *OGRE Meshes* wählen

8 *Preferences* aufrufen, um Pfad zum *OgreXMLConverter* einzustellen

Abbildung 3.5: Neues Teilfenster in *Blender* erzeugen, *Skript Modus* einstellen und *Blender Exporter* (alias *Ogre Mesh Exporter*) aufrufen

3.5 Einbindung in CViewVR

Es wird davon ausgegangen, daß die Modelle bereits im Format *Ogre Mesh* vorliegen. Zur Verwendung müssen diese in die Visualisierungssoftware CViewVR integriert werden. Mesh-Dateien (`*.mesh`) werden in das Unterverzeichnis `CViewVR\Resources\models` kopiert und Materialdateien nach `CViewVR\Resources\materials`.

Auf der CD dieser Arbeit liegen Batch-Dateien bei (z.B. `integrate_seewolf.bat`), welche die Integration erleichtern und auf einen typischen Fehler prüfen. Wenn nämlich die Materialdatei den Standardnamen `Scene.material` hat, dann wird eine Warnung ausgegeben und die Integration unterbunden. Das soll verhindern, daß bereits integrierte Materialdateien überschrieben werden. Weiterhin sind die Batch-Dateien eine Arbeitserleichterung, wenn ein Modell bei der Bearbeitung wiederholt exportiert und eingebunden werden muß.

Soll ein Fahrzeug (oder anderes Objekt) in der Visualisierungssoftware angezeigt werden, so muß es zuvor in eine auf XML basierende Szenendatei (`*.cv2r`) eingebunden werden. Dazu wird ein neuer XML-Eintrag vom Typ *IEObject* eingefügt. Aus dessen Parametern wird später durch CViewVR ein Ogre Szenenknoten erzeugt und im Szenengraph eingefügt. Die Bearbeitung geschieht über einen Text- oder XML-Editor.

Beim Parameter *nodeName* wird der Name des Fahrzeugs angegeben, über den dieses später durch externe Fahrzeugsimulatoren gesteuert werden kann. Das einzubindende Fahrzeugmodell (`*.mesh`) wird bei *meshName* eingetragen. Koordinaten im Abschnitt *position* beziehen sich auf das Ogre-Koordinatensystem. Die räumliche Ausrichtung wird im Datenblock *orientation* in Form von Quaternionen hinterlegt. In der Grundausrichtung (alle NED-bezogenen Eulerwinkel sind 0°) zeigt das Fahrzeug nach Norden und die entsprechenden Quaternionsparameter sind $(1, 0, 0, 0)$. Soll das Fahrzeug (oder ein anderes Objekt) eine andere Ausrichtung bekommen, so müssen die Parameter entsprechend berechnet und eingetragen werden. Ein Online-Umrechner für Quaternionen befindet sich in [URL 68] und eine installierbare Version in [URL 69]. Der Parameter *parentNode* gibt den übergeordneten Szenenknoten an. Häufig wird dort „*Ogre/SceneRoot*“ eingetragen, wodurch das Modell keinem anderen Szenenknoten untergeordnet ist. Wird statt dessen der Name eines vorhandenen Szenenknotens angegeben, so sind Positionierung und Orientierung relativ zu diesem sogenannten Elternknoten. Das ist beispielsweise notwendig, um eine fahrzeuggebundene Kamera zu integrieren. Die Positionierung und Orientierung bezüglich des Geländes spielt dabei keine Rolle, sondern nur die Relation zum Fahrzeug.

Weitere Parameter und Details zum Aufbau einer Szenendatei sind in Kapitel 4.3

(S. 47) dokumentiert. Das XML-Dokument einer einfachen Beispielszene befindet sich im Anhang E (S. 102).

Szenendateien werden im Unterverzeichnis `CViewVR\Scenes` der Visualisierungssoftware hinterlegt. Zur Validierung einer Szenendatei (z.B. durch einen XML-Editor) kann das XSD-Schema aus Datei `scenefile.xsd` verwendet werden, welche im selben Verzeichnis liegt wie die Szenendateien. Das XSD-Schema ist auch im Anhang E.2 (S. 104) abgedruckt.

Ein Szeneneditor ist in CViewVR nicht vorhanden. Zur Erleichterung kann das Programm *WinClient* verwendet werden, welches während der Entwicklung von CViewVR entstand und dieser Arbeit beiliegt. Es ist kein Bestandteil der Visualisierungssoftware, sondern ein eigenständiges Programm, das sich über die Netzwerkschnittstelle mit CViewVR verbindet. Damit ist es möglich, Fahrzeuge und andere Objekte (z.B. Hafenmauern) in einer Szene zu bewegen. Dies ist hilfreich bei der Erstellung einer Szene. Weitere Informationen zum *WinClient* stehen im Kapitel 4.5 auf Seite 57. Das Programm ist in Abbildung 5.12 (S. 70) zu sehen.

Durch Einbindung von Fahrzeugen in eine Szenendatei werden diese in CViewVR angezeigt. Nun können sie über den *WinClient* wie gewünscht positioniert werden. Anschließend wird die geänderte Szene über das Menü von CViewVR gespeichert.

Bei Szenen mit großem Gelände (z.B. 10×10 km^2) sind kleine Fahrzeuge schwer auffindbar und können beim Positionieren schnell außer Sichtweite geraten oder auf eine Darstellungsgröße von wenigen Pixeln schrumpfen. Als Hilfestellung kann in der Programmoberfläche die Option *coord sys* aktiviert werden, wodurch für jedes Fahrzeug ein objektbezogenes Koordinatensystem mit langen, gut sichtbaren Achsen angezeigt wird (Abbildung 5.8).

Eine weitere Möglichkeit besteht darin, Fahrzeuge durch Erhöhung der Skalierungsparameter *scale* (in der Szenendatei) zu vergrößern – beispielsweise um den Faktor 10 bei großem Gelände.

3.6 Modellmanipulator MeshMagick

Das Kommandozeilenprogramm *MeshMagick* wird von den Ogre-Entwicklern zur Verfügung gestellt. Damit ist es möglich, Modelle zu bearbeiten, die im Ogre Mesh-Format vorliegen. Folgende Aktionen sind durchführbar:

- Informationen über das Modell ausgeben (z.B. Anzahl der Dreiecke)
- Skalierung, Rotation, Translation (hilfreich zur Anpassung von Größe, Orientierung und Lage des Rotationspunktes)
- Optimierung des Modelles (z.B. Entfernung redundanter Eckpunkte)
- Umbenennung von Elementen (z.B. Behebung von Namenskonflikten)
- Verschmelzung von Mesh-Dateien

Trotz der Möglichkeiten von MeshMagick ist es sinnvoller, Modelle statt dessen direkt in Blender zu bearbeiten. Eine Verwendung dieses Programmes kann hilfreich sein, wenn ein Modell ausschließlich im Ogre-Format vorliegt. MeshMagick und weitere Information gibt es im Ogre-Wiki unter [URL 24].

4 Entstandene Software CViewVR

Auf Grundlage der Recherche über bereits vorhandene Visualisierungssoftware in Kapitel 2.1 (S. 8) wurde beschlossen, auf dem Prototyp aufzubauen, der von Stefan Zschäck im Fraunhofer Anwendungszentrum für Systemtechnik erstellt und in Kapitel 2.2 (S. 10) beschrieben wurde.

4.1 Weiterentwickelter Prototyp

Der Prototyp wurde umfangreich überarbeitet, verbessert und erweitert. Abbildung 4.1 zeigt den strukturellen Aufbau der entstandenen Visualisierungssoftware CViewVR. Details der Änderungen sind in den folgenden Abschnitten schwerpunktmäßig zusammengefasst.

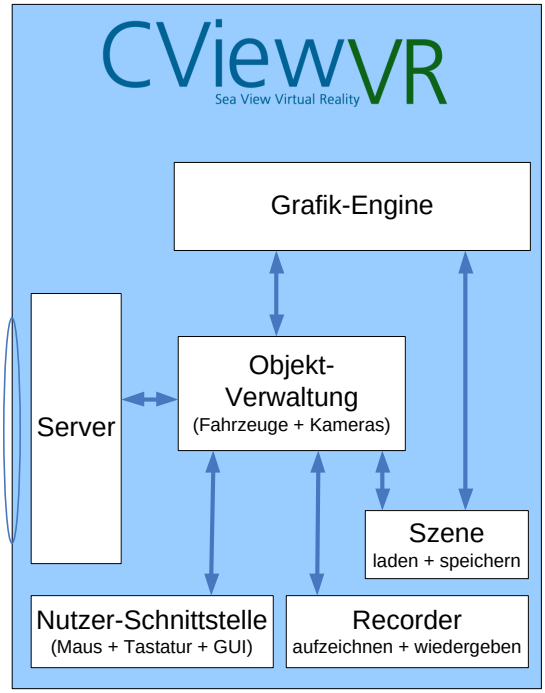


Abbildung 4.1: Aufbau des erweiterten Prototyps

Erweiterte Funktionalität

- Erstellung und Integration einer Kommunikationsklasse, mit deren Hilfe sich Fahrzeugsimulatoren über einen integrierten TCP-Server mit CViewVR verbinden können. Details dazu folgen in Kapitel 4.4 (S. 49).

- Eine weitere Klasse übernimmt die Umrechnung der empfangenen Daten in die von Ogre verwendete Darstellungsform. Positionsangaben werden vom NED-Koordinatensystem in das Ogre-Koordinatensystem konvertiert und die durch Eulerwinkel beschriebene Orientierung wird in Quaternionswerte umgerechnet.

- Fahrzeugbestandteile (z.B. Schiffsschraube) können durch relative Positions- und Rotationsangaben gesteuert werden. Die Werte beziehen sich dann nicht auf das Koordinatensystem der Szene, sondern sind relativ zum Fahrzeug.

- Ein Kompass wurde integriert, um dem Anwender eine bessere Orientierung innerhalb einer Szene zu ermöglichen.

- Das Bewegungsverhalten der (extern gesteuerten) Fahrzeuge wird automatisch aufgezeichnet und standardmäßig in der Datei `streamLog.txt` gespeichert. Durch diese Aufzeichnung ist eine spätere Wiedergabe der Fahrzeugbewegungen mithilfe des erstellten Zusatzprogramms *TCP-Client* (Kapitel 4.5, S. 56) möglich.

- Option *koord sys* blendet für jedes Fahrzeug das lokale Koordinatensystem ein. Abbildung 5.8 (S. 67) zeigt die grafische Darstellung.

- Mithilfe der Option *top view* kann in die Vogelperspektive gewechselt werden. Bei Deaktivierung wird der alte Blickwinkel wiederhergestellt.

- Einzelne Bildschirmfotos können über die Tasten P und I erstellt werden. Bei der ersten Taste wird das Bildschirmfoto über GDI+/.NET erstellt und bei der zweiten über eine Funktion der Ogre-Bibliothek. Die Speichergeschwindigkeit der beiden Methoden unterscheidet sich je nach verwendeter Grafikkarte. Bei Aktivierung der Option *capture pics* werden fortlaufend Aufnahmen erstellt. Jedes gerenderte Bild wird dabei in einer JPG-Datei gespeichert. Anschließend ist es möglich, aus den Einzelbildern –mithilfe eines Zusatzprogramms– ein Video zu erstellen. Es sei angemerkt, daß diese Funktionalität noch nicht ausgereift und als experimentell zu betrachten ist.
  Problematisch ist die hohe CPU-Belastung bei der fortlaufenden Erstellung von Bildschirmfotos und die begrenze Festplattengeschwindigkeit bei der Speicherung

von Dateien. Es wurde festgestellt, daß die Nutzung des Dateiformates PNG eine gute Bildqualität erreicht wird, aber die Komprimierung und Speicherung relativ viel Zeit beantsprucht. Deutlich schneller ist die Nutzung des Dateiformates JPG, welches jedoch aufgrund der verlustbehafteten Kompression schlechtere Bildqualität liefert. In CViewVR wurde ein Verfahren implementiert, das zur Bildkomprimierung mehrere CPU-Threads erstellt, um die Last auf alle CPU-Kerne zu verteilen. Dennoch kommt es –insbesondere bei höheren Auflösungen– zu starkem Ruckeln und einem Jitter-Effekt. Daher wird zur Erstellung von Videos empfohlen, spezielle Software wie beispielsweise FRAPS [URL 28] zu nutzen.

- Weiterhin werden vom überarbeiteten Prototyp mehrere Logdateien angelegt, um eine Fehlersuche zu erleichtern: Da die Logdatei der Ogre-Bibliothek `ogreLog.txt` sehr viele Ausgaben beinhaltet, ist sie weniger übersichtlich. Nun existiert eine zusätzliche Logdatei `ogreLog-onlyProblems.txt`, die nur Warnungen und Fehlermeldungen der Ogre-Bibliothek enthält. Datei `applicationLog.txt` enthält Statuszustände (z.B. welche Szene geladen wurde, daß sich ein Fahrzeugsimulator mit CViewVR verbunden hat, Fehlermeldungenen etc.) und zusätzliche Ausgaben, die sich auf die Visualisierungssoftware CViewVR beziehen. Der Umfang der Ausgaben ist abhängig von der gewählten Detailstufe (0: nur kritische Meldungen, 1: normal, 2: detaillierte Statusmeldungen zur Fehlersuche). Datei `applicationLog-onlyErrors.txt` enthält ausschließlich Warnungen und Fehlermeldungen.

- An vielen Stellen des Quellcodes wurden Fehlerprüfungen und -behandlungen hinzugefügt. Der ursprüngliche Prototyp enthielt keinerlei Fehlerprüfungen, was häufige Abstürze zur Folge hatte, dessen Ursache nicht nachvollziehbar war. Beispielsweise wenn ein nicht vorhandenes 3D-Modell geladen werden sollte, verschiedene Materialien den gleichen Materialnamen verwendeten, Fehler in einer Szenendatei enthalten waren, etc. Neben den internen Fehlerbehandlungen werden dem Anwender konkrete Fehlermeldungen gezeigt und Einträge in die Logdateien gemacht.

- Die Animationsgeschwindigkeit der Wasseroberfläche kann während der Programmlaufzeit geändert werden. Dies ist sinnvoll für die Erstellung von Einzelfotos für Videos mit großer Bildrate und hoher Auflösung. Dabei wird eine aufgezeichnete „Mission“ (Fahrzeugbewegungen) während der fortlaufenden Bilderstellung langsamer wiedergegeben und bei der späteren Videoerstellung wieder beschleunigt.

Verbesserte Bedienung

- Die Bedienung der freien Kamera wurde verbessert. Bewegungen erfolgen nun standardmäßig parallel zur Oberfläche. Änderungen der Geschwindigkeitseinstellung erfolgen nun logarithmisch. Dies ist praxistauglicher, da häufig zwischen schnellen und langsamen Bewegungen gewechselt werden muß. (Schnell beim Agieren in einer großen Szene und langsam im Nahbereich.) Gedreht wird die freie Kamera mithilfe der linken Maustaste, was für den Anwender angenehmer und intuitiver ist als die rechte Maustaste, welche vom alten Prototyp zum Drehen verwendet wurde.

- Eine fps-Anzeige informiert den Anwender über die aktuelle Bildrate.

- Logdateien sind über das Menü aufrufbar.

- Beim Stoppen der Visualisierung wird die derzeit aktive Kamera, sowie deren Position und Orientierung zwischengespeichert, um beim erneuten Start die gleiche Kameraperspektive zu haben. (Im alten Prototyp wurden statt dessen die aktuellen Kameraeinstellungen verworfen und die Konfiguration aus der Szenendatei geladen.)

- Beim Laden einer neuen Szene werden die Einstellungen der grafischen Oberfläche übernommen (z.B. die Nebeleinstellung).

- Verbesserte Bedienbarkeit beim Laden und Speichern von Szenen: Das zuletzt verwendete Verzeichnis wird im Dateidialog voreingestellt. Beim Speichern einer Szene wird der Name der geladenen Szenendatei vorgeschlagen. (Im alten Prototyp war das Namensfeld leer und das Verzeichnis mußte jedes Mal neu gewählt werden.) Nach Bearbeitung einer Szenendatei (durch einen Texteditor) kann diese über den Menüeintrag *Reload* neu geladen werden, ohne die Datei im Dateidialog suchen zu müssen.

- Eine kleine Hilfe ist in der Seitenleiste der Visualisierungssoftware permanent sichtbar und zeigt die Tastaturbelegung und Maussteuerung an.

Programminterne Verbesserungen

- Der Quellcode zum Laden und Speichern von Szenendateien wurde neu erstellt, verbessert und erweitert. Nun wird eine Validitätsprüfung der zu ladenden Datei

anhand eines XSD-Schemas durchgeführt und bei Fehlern eine konkrete Fehlermeldung ausgegeben (statt kommentarlos abzustürzen wie der alte Prototyp). Weitere Fehlerquellen werden ebenfalls überprüft (z.B. Gültigkeit der Quaternionsparameter). Zudem können Szenendateien zusätzliche Informationen enthalten – etwa die Latitude-Longitude-Position der Karte und das Speicherdatum (welches im Dateiverwaltungssystem Subversion verlorengeht). (Details zum XSD-Schema folgen im Kapitel 4.3 (S. 47). Ein Abdruck der Szenendatei befindet sich in Anhang E.2 (S. 104).)

- An vielen weiteren Stellen des Quellcodes wurden Fehlerprüfungen eingefügt, um Programmabstürze und Blockierungen (Deadlocks) zu vermeiden.
- Die Verzeichnisstruktur wurde überarbeitet und viele unnötige Dateien entfernt (Texturbilder, Materialien, Modelle, etc.).
- Redundanter und unstrukturierter Code wurde überarbeitet und teilweise in neu erstellte Klassen ausgelagert.
- Es wurde eine Objektverwaltung für Kameras, Fahrzeuge und sonstige Modelle erstellt. Diese kann zusätzliche Informationen zu jedem Objekt enthalten und Umrechnungen für die Koordinatensysteme durchführen. Für zukünftige Erweiterungen kann die Objektverwaltung genutzt werden, um auch Sensoren und dynamisch generierte Objekte zu verwalten. (Im alten Prototyp wurden die Objekte der Szenendatei direkt in den Szenengraph eingefügt, bzw. davon ausgelesen. Das war wenig flexibel und konnte in bestimmten Situationen zu Fehlern führen.)
- Ein auffälliger Darstellungsfehler beim Himmel wurde behoben. (Zuvor war „unterhalb des Horizonts“ alles schwarz.)
- Wenn eine Visualisierung gestoppt wird, bleibt das zuletzt gerenderte Bild als Standbild erhalten und ein Pause-Symbol wird eingeblendet. (Im alten Prototyp war statt dessen ein Standbild mit Bildstörungen zu sehen.)

Unterstützung für Entwickler

- Der Quelltext der Visualisierungssoftware wurde umfangreich dokumentiert. Alle Klassen, Methoden, Parameter und viele Variablen wurden mit XML-Kommentaren versehen. Dadurch wird während der Programmierung automatisch ein kontextsensitiver Hilfetext zu den verwendeten Elementen eingeblendet. Zudem

wurde mithilfe des Programms *Sandcastle Help File Builder* [URL 65] aus den Quellcode-Kommentaren eine Hilfe-Datei im CHM-Format[7] erstellt.

- Für unbehandelte Ausnahmen von unerwarteten Fehlern gibt es zwei Modi. Im Normalfall werden detaillierte Informationen zur Absturzursache in den Logdateien gespeichert. Wenn CViewVR innerhalb der Programmierumgebung Visual Studio läuft, dann befindet es sich automatisch im „Entwickler-Modus“, der keine unbehandelten Ausnahmen abfängt. Vorteil dieses Modus ist, daß dadurch die Programmierumgebung genau die Stelle im Quellcode anzeigen kann, wo das Problem aufgetreten ist. Zudem ist die Programmausführung nur unterbrochen und die Werte der verwendeten Variablen sind einsehbar.

- Die Option *debug line* blendet in der grafischen Oberfläche ein mehrzeiliges, scrollbares Textfeld ein, welches für Debugging-Ausgaben genutzt werden kann. Dies ist besonders hilfreich für die Überwachung von Parametern, die sich während der Laufzeit schnell ändern – beispielsweise Koordinaten von fahrenden Fahrzeugen.

3D-Inhalte

- Mehrere 3D-Modelle wurden in die Visualisierungssoftware integriert, beispielsweise von den Fahrzeugen *Seewolf*, *Infante* (Instituto Superior Técnico, Lissabon, Portugal) und *Delfim* (Atlas Elektronik, Bremen, Deutschland), welche im Rahmen des EU-Forschungsprojektes GREX [URL 2] verwendet wurden. Während der Entwicklung einer Fahrzeugführungssoftware für GREX wurden die Modelle an der TU Ilmenau im Fachgebiet Systemanalyse genutzt, um Simulationsergebnisse mithilfe der Visualisierung zu überprüfen. Die 3D-Modelle standen im VRML-Format zur Verfügung, mußten jedoch für die Überführung in das Ogre Mesh-Format aufwändig bearbeitet werden. Beispielsweise bestand das Seewolf-Modell aus 2600 Einzelteilen mit redundanten Materialdefinitionen, die entfernt, gobal neu definiert und für alle Einzelteile referenziert werden mußten.

- Mit dem Geländegenerator L3DT [URL 58] wurde ein Gelände erstellt, das $10 \,\mathrm{x}\, 10$ km^2 groß ist und etwa zu 80% aus Wasserfläche besteht. Es enthält flache Gewässer, steile Unterwasserabhänge und einen tief liegenden, unebenen Meeresboden. Abbildung 2.6 (S. 22) zeigt das Gelände.

[7] CHM steht für *Compiled HTML Help* und ist Dateiformat von Microsoft, das für die Speicherung von Hilfedateien genutzt wird, welche ohne Zusatzsoftware geöffnet werden können.

- Basierend auf diesem Gelände wurden verschiedene Szenen erstellt und in Szenendateien hinterlegt. Sie enthalten Unterwasser- und Oberflächenfahrzeuge in verschiedenen Konstellationen und teilweise Mauern, die einen stark vereinfachten Hafen darstellen sollen. Eine weitere Szene enthält am Meeresgrund eine vereinfachte Pipeline, die für die Visualisierung von Pipelineinspektionen genutzt werden kann.

- Weiterhin wurde das Modell eines Geländefahrzeugs und eine Landszene (überwasser) mit verschiedenen Hindernissen bei CViewVR integriert, die ein Mitarbeiter des Fraunhofer AST mit dem 3D-Modellierungsprogramm *3ds Max* erstellte. Die Szene wurde zur Validierung eines Simulators verwendet, der die Fahrzeugdynamik des Geländefahrzeugs simuliert.

- Da Szenen nur mit einem Text- oder XML-Editor erstellt und verändert werden können, wurde in CViewVR eine Funktionalität integriert, die gespeicherte Szenendateien neu formatiert, um sie für den Anwender übersichtlicher zu machen. Dazu werden an bestimmten Stellen Leerzeilen eingefügt und die enthaltenen Objekte (z.B. Fahrzeugmodelle und Kameras) durch Kommentarzeilen hervorgehoben, die den Modellnamen enthalten.

4.2 Grafische Oberfläche

Die Oberfläche der Software CViewVR besteht aus einem großen Anzeigebereich für die 3D-Visualisierung, einer Seitenleiste für Funktionen und Optionen, sowie einer Menüleiste (Abbildung 4.2).

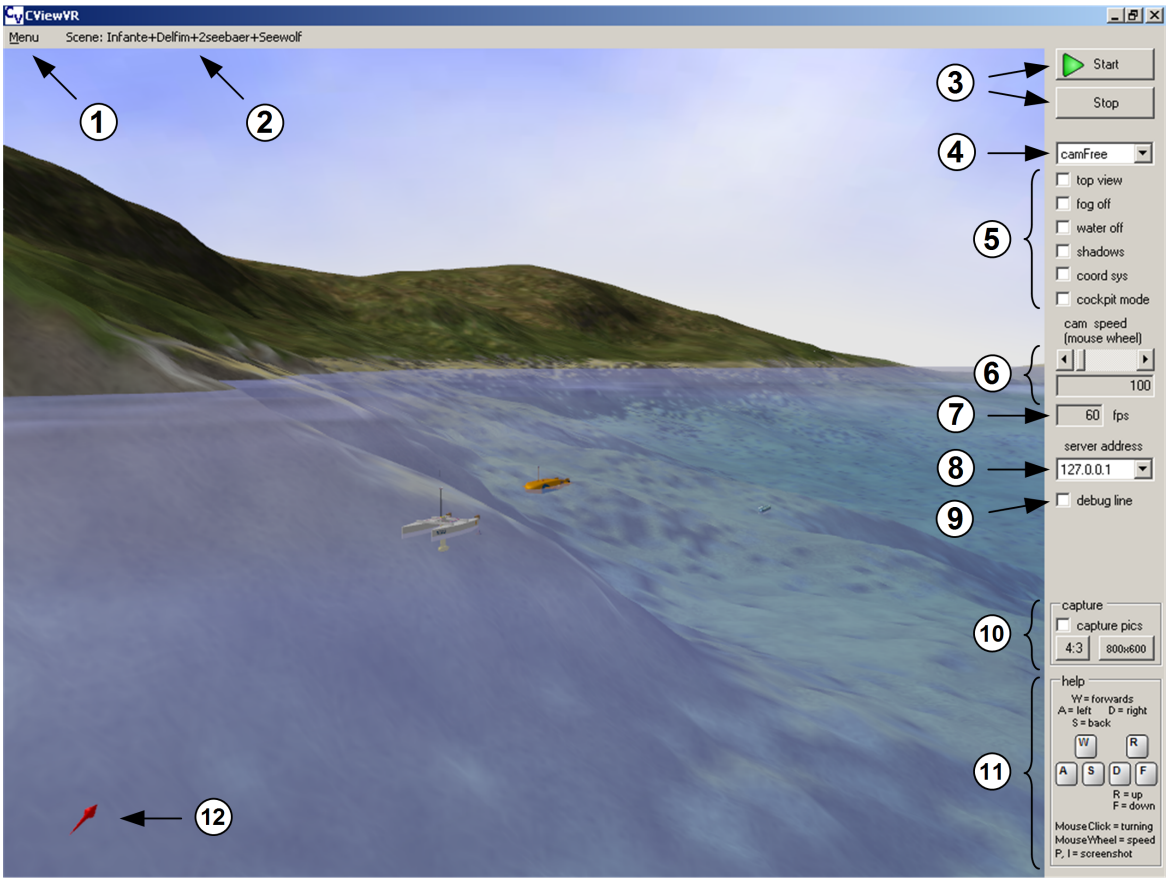


Abbildung 4.2: Grafische Oberfläche von CViewVR

① Menü zum Laden und Speichern von Szenen, Aufruf von Logdateien, etc.

② Name der aktuellen Szenendatei

③ Knöpfe zum Starten und Beenden der Visualisierung

④ Auswahl der Kamera, dessen Ansicht dargestellt werden soll

⑤
- *top view*: Wechsel in die Vogelperspektive
- *fog off*: Deaktivierung von Nebeleffekt und Wassertrübung (siehe Abbildungen 5.6, 5.7 auf S. 67)
- *water off*: Ausblendung der Wasseroberfläche

shadows: Anzeige von Schatteneffekten

coordsys: Für jedes Objekt in der Szene wird ein Koordinatensystem eingeblendet (siehe Abbildung 5.8 auf S. 67)

cockpit mode: Nutzung eines alternativen Bewegungsmodus für die freie Kamera — Standardmäßig bleibt die Flughöhe bei Vorwärts- und Rückwärtsbewegungen konstant. Beim aktivierten Cockpitmodus bewegt sich die Kamera statt dessen entlang der Sichtachse (Blickrichtung).

⑥ Änderung der Geschwindigkeit für Kamerabewegungen (auch möglich über das Mausrad); Anzeige des aktuellen Geschwindigkeitsfaktors im Feld unter dem Scrollbalken

⑦ Anzeige der Rendergeschwindigkeit in Bildern pro Sekunde (fps = frames per second)

⑧ Anzeige der IP-Adresse, über den der TCP-Server erreichbar ist – eine Änderung der Adresse ist ebenfalls möglich

⑨ Einblendung von Informationen zur Analyse und Fehlersuche während der Entwicklung von CViewVR

⑩

capture pics: Fortlaufende Aufnahme von Screenshots, aus denen später ein Video generiert werden kann (experimentelle Funktion; siehe Details auf S. 37, Punkt 8)

4:3: Änderung des Seitenverhältnisses auf 4:3

800x600: Programmoberfläche auf 800x600 px^2 reduzieren

⑪ Übersicht der Tastenkürzel

⑫ Kompass

Beim Programmstart von CViewVR wird die Standardszene (`default.cv2r`) geladen. Andere Szenen können über das Menü ① geladen werden. Der Name der aktuellen Szenendatei ② wird neben dem Menü angezeigt. Die Visualisierung beginnt nach Anklicken des *Start*-Knopfes ③. Dadurch startet ebenfalls der integrierte TCP-Server, über den sich externe Fahrzeugzeugsimulatoren verbinden können. Der *Stop*-Knopf ③ beendet die Visualisierung, sowie den TCP-Server. Eine Änderung der IP-Adresse für den TCP-Server ist über eine weitere Auswahlliste ⑧ möglich. Dabei wird der Server neu gestartet und vorhandene Verbindungen beendet. Fahrzeuggebundene Kameras können über eine Auswahlliste ④ gewählt werden. Bei Aktivierung der Option *capture pics* ⑩ werden fortlaufend Bildschirmaufnahmen erzeugt und als einzelne Bilddateien im Unterverzeichnis `CViewVR\Screenshots` gespeichert. Daraus kann später (durch ein zusätzliches Programm) ein Video erzeugt werden. Diese Funktion ist verbesserungswürdig und als experimentell zu betrachen. Ein Kompass ⑫ zeigt nach Norden und hilft bei der Orientierung. Sämtliche Logdateien sind über das Menü aufrufbar.

Kamerasteuerung

Während der Visualisierung kann die Ansicht der freien Kamera (*camFree*) über Tastatur und Maus geändert werden. Bewegungen (Positionsänderungen) werden über die Tastatur und Kameradrehungen über die Maus gesteuert. Eine Anpassung der Bewegungsgeschwindigkeit ist durch Drehen des Mausrades oder die grafische Oberfläche ⑥ möglich. Üblicherweise wird die Kamera mit der linken Hand bewegt (siehe Abbildung 4.3) und und über die rechte Hand rotiert (siehe Abbildung 4.5). Die Tasten **A**, **S**, **D**, **W** bewegen die Kamera vorwärts/rückwärts, bzw. nach links/rechts. Dabei wird die Bewegung immer horizontal ausgeführt (unabhängig vom Nickwinkel) und der Abstand zur Wasseroberfläche bleibt konstant. Die vertikale Position kann über die Tasten **R** und **F** geändert werden. Alternativ ist es möglich, die Kamera über die *Pfeiltasten* und *Bild auf/ab*-Tasten zu bewegen (siehe Abbildung 4.4), was jedoch bei gleichzeitiger Bedienung durch Maus und Tastatur nicht ergonomisch ist.

Bei Aktivierung der Option *cockpit mode* ⑤ ändert sich das Bewegungsverhalten der Kamera und ähnelt der Sichtweise eines Flugzeugpiloten. Vorwärts- und Rückwärtsbewegungen entsprechen der aktuellen Blickrichtung und sind nicht mehr an die horizontale Ebene gebunden. Auf- und Abwärtsbewegungen durch die Tasten **R** und **F** sind senkrecht zur Blickrichtung.

Zum Drehen der Kamera muß die Maus in die gewünschte Richtung bewegt und gleichzeitig die linke Maustaste gedrückt werden.

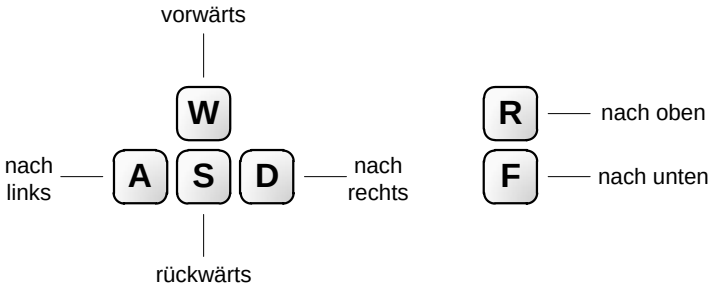


Abbildung 4.3: Kamerapositionierung über Tastatur (Bedienung durch linke Hand)

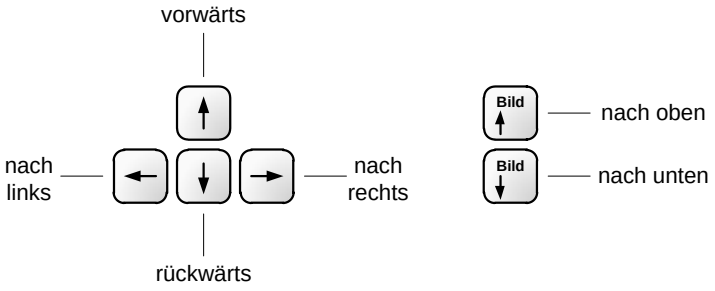


Abbildung 4.4: Alternative Kamerapositionierung (Tastaturbelegung für rechte Hand)

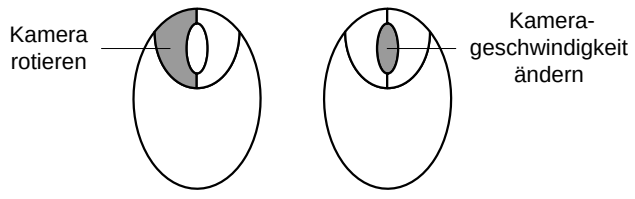


Abbildung 4.5: Kamerabedienung über die Maus

4.3 Szenen-Dateiformat

In diesem Abschnitt werden Details zum Aufbau einer Szenendatei erklärt. Die Integration von 3D-Modellen wurde im Kapitel 3.5 (S. 33) beschrieben. Grundlagen zur Geländeerstellung befinden sich im Kapitel 2.7 (S. 21).

Szenen sind in XML-Dateien gespeichert und bestehen aus zwei Hauptbestandteilen. Im ersten stehen allgemeine Definitionen bezüglich der Szene, beispielsweise der Name der verwendeten Konfigurationsdatei für das zu ladende Gelände. Der zweite Teil ist eine Liste (`ObjList`) aller enthaltenen Objekte (Fahrzeuge, Bauwerke, Kameras, etc.) mit zugehörigen Parametern. Jedes Objekt ist in einem eigenen XML-Element namens `IEObject` definiert und entspricht einem Szenenknoten, der beim Laden einer Szene erstellt wird.

In Tabelle 4.1, 4.2 und 4.3 ist eine Beschreibung der einzelnen XML-Elemente enthalten. Die XML-Beschreibung einer einfachen Beispielszene befindet sich im Anhang E (S. 102).

Es wurde eine Schema-Definition erstellt, welche beschreibt, nach welchen Regeln Szenendateien aufgebaut sein müssen. Darin ist festgelegt, welche XML-Elemente erlaubt und in welcher Reihenfolge sie zu verwenden sind. Weiterhin wird das Datenformat der enthaltenen Daten definiert und bei einigen String-Datentypen vorgeschrieben, daß nur bestimmte Zeichen erlaubt sind. Mithilfe dieser Schema-Definition wird jede zu ladende Szenendatei von CViewVR auf ihre Gültigkeit geprüft. Sie kann zudem hilfreich sein, wenn eine Szenendatei mithilfe eines XML-Editors erstellt oder bearbeitet wird. Die Schema-Definition befindet sich in einer XSD-Datei mit dem Namen `scenefile.xsd` und liegt im Unterverzeichnis `Scenes` von CViewVR. Sie ist auch im Anhang E.2 (S. 104) abgebildet.

Hinweis: Für den Namen eines Objektes (`nodeName`) gibt es grundsätzlich keine Längenbeschränkung. Bei Fahrzeugen ist jedoch darauf zu achten, daß der Name maximal 20 Stellen lang ist, da sonst eine Steuerung über die TCP-Schnittstelle nicht möglich ist.

XML-Element	Datentyp	Bedeutung
`IEScene`	benutzerdefiniert	Wurzelelement der XML-Datei
`ObjList`	benutzerdefiniert	Liste mit allen Szenenknoten
`IEObject`	benutzerdefiniert	Enthält Parameter eines Szenenknotens

Tabelle 4.1: Spezielle XML-Elemente der Szenendatei

XML-Element	Datentyp	Bedeutung
`saveDate`	String	Tag der Speicherung (Format: yyyy-mm-dd)
`terrainConfigFile`	String	Datei mit Geländeparametern – siehe Abschnitt 2.7 (S. 21)
`waterlevel`	Float	Höhe der Wasseroberfläche (Einheit: Meter)
`worldPosition _isDefined`	Boolean	Wenn *false*, dann werden die folgenden zwei Parameter ignoriert
`worldPosition _latitude_degree`	Double	Latitude-Position des Geländes
`worldPosition _longitude_degree`	Double	Longitude-Position des Geländes – siehe Abschnitt 4.6 (S. 58)

Tabelle 4.2: Szenenbezogene XML-Elemente der Szenendatei

XML-Element	Datentyp	Bedeutung
`nodeName`	String	Name des Szenenknotens; Empfehlung: maximal 20 Zeichen; Groß-/Kleinschreibung wird unterschieden; erlaubte Zeichen: `a-z A-Z 0-9 _ . - /`
`meshName`	String	Name der Mesh-Datei (3D-Modell); Endung: `*.mesh`; erlaubte Zeichen: `a-z A-Z 0-9 _ . -`
`parentNodeName`	String	Name des übergeordneten Szenenknotens
`position`	3x Float	Position des Szenenknotens bezüglich des Ogre-Koordinatensystems, siehe Abbildung 2.3 (S. 15)
`orientation`	4x Float	Räumliche Ausrichtung des Szenenknotens; beschrieben durch Quaternionen; `(1,0,0,0)` entspricht der Ausrichtung nach Norden
`castShadows`	Boolean	Wenn *true*, dann wirft das Objekt bei aktivierter GUI-Option *shadows* einen Schatten. Wenn *false*, dann wirft es nie einen Schatten
`camNode`	Boolean	Wenn *true*, dann repräsentiert dieses Objekt eine Kamera
`scale`	3x Float	Skalierung des Objektes (üblicherweise `(1,1,1)`)

Tabelle 4.3: Objektbezogene XML-Elemente der Szenendatei

4.4 TCP-Schnittstelle für Fahrzeugsimulatoren

Ein Teilaufgabe dieser Studienjahresarbeit war, eine Kommunikationsmöglichkeit für die Anbindung eigenständiger Simulationsprogramme zu schaffen. Das soll die Steuerung von Fahrzeugen ermöglichen, die sich in der Visualisierungssoftware befinden. Im Vordergrund stand eine unidirektionale Kommunikation zum Senden von fahrzeugbezogenen Zustandsdaten vom Simulator zur Visualisierungssoftware. Ein Rückkanal war nicht vorgesehen, jedoch sollte die Möglichkeit bestehen, CViewVR zu einem späteren Zeitpunkt auf eine bidirektionale Kommunikation zu erweitern. Entsprechend der Anforderungsanalyse in Kapitel 2.6 (S. 19) wurde TCP als Transportprotokoll für die Kommunikation gewählt. Darauf basierend wurde ein verbindungsorientiertes Anwendungsprotokoll entworfen und implementiert, um die erforderlichen Daten (z.B. Fahrzeugposition) zu übertragen.

CViewVR enthält einen selbst erstellten TCP-Server, der Verbindungsanfragen auf Port 8000 der PC-internen IP-Adresse 127.0.0.1 entgegennimmt. Adresse und Port können in der Datei `ipconfig.txt` geändert werden. Weiterhin besteht die Möglichkeit, die IP-Adresse während der Programmlaufzeit zu ändern. Alle im Betriebssystem verfügbaren Adressen werden beim Programmstart ermittelt und können über die grafische Oberfläche ausgewählt werden. Der TCP-Server läuft in einem eigenen Thread und erstellt für jede neue TCP-Verbindung einen weiteren Thread zur Verarbeitung. Die empfangenen und ausgewerteten Daten werden an den Hauptthread der Visualisierungssoftware weitergereicht.

Der TCP-Server kann mehrere Verbindungen zu verschiedenen Fahrzeugsimulatoren parallel verarbeiten. Die Fahrzeugsimulatoren können sich auch auf einem anderen PC befinden, wobei jedoch darauf zu achten ist, dafür in CViewVR die entsprechende IP-Adresse einzustellen.

4.4.1 Aufbau des Übertragungsprotokolls

Zur Übertragung eines Fahrzeugzustands wird ein Rahmen (Datenblock) verwendet, der 96 Byte lang ist und 14 Felder mit binär codierten Daten enthält. Abbildung 4.6 (S. 50) zeigt den Aufbau des Rahmens. Die Länge der einzelnen Felder, deren enthaltenen Datentypen und kurze Anmerkungen befinden sich in Tabelle 4.4 (S. 50), eine Übersicht der bisher verwendeten Flags stehen in Tabelle 4.5. Eine detaillierte Erklärung der Felder folgt ab Seite 51. Bei jeder Zustandsänderung der Fahrzeugmodelle im Simulator oder in einem gewissen Intervall werden dann Datensätze über den TCP-Kanal gesendet.

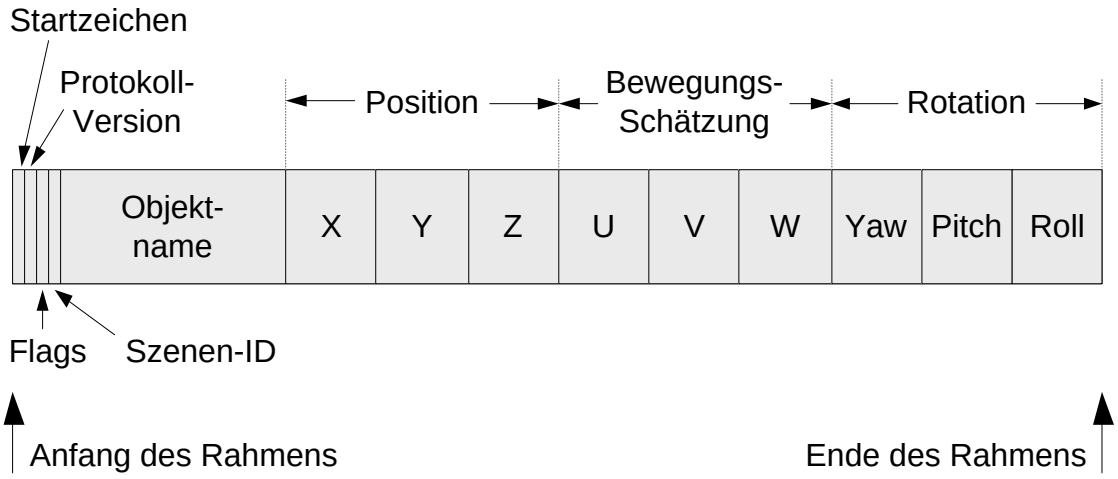


Abbildung 4.6: Aufbau des Rahmens zur Datenübertragung

Anz. d. Bytes	Feldname	Datentyp	Bemerkung
1	Startzeichen	Byte	Wert = 255 (hex: FF)
1	Protokoll-Version	Byte	derzeit Version 2
1	Flags	Byte	beschreibt 8 Bit-Flags
1	Szenen-ID	Byte	ungenutzt
20	Objekt-Name	20 Chars	8 bit pro Zeichen (ASCII)
8	X		Position (NED-Koordinaten in Meter;
8	Y	Double	Bei gesetztem LatLong-Flag:
8	Z		X$\widehat{=}$Latitude, Y$\widehat{=}$Longitude in RAD)
8	U		
8	V	Double	Parameter zur Bewegungs-Schätzung
8	W		(bisher nicht verwendet)
8	Yaw		
8	Pitch	Double	Rotation (seit Version 2: Angabe in RAD)
8	Roll		
= 96 Bytes			

Tabelle 4.4: Felder des Rahmens zur Datenübertragung

Flag-Position	Byte-Wert	Hex-Wert	Flag-Name	Bedeutung
`x___ ____`	128	`0x80`	LatLong	Wenn *true*, dann werden die Positionsfelder X und Y als Latitude-Longitude-Werte im Bogenmaß interpretiert.

Tabelle 4.5: Bisher verwendete Flags

Hinweise

Übertragene Positionsangaben entsprechen dem NED-Koordinatensystem, dessen Bezug zur Szenendatei in Abbildung 4.7 dargestellt ist. Eine Positionierung bezüglich globaler Latitude-Longitude-Koordinaten ist möglich, indem das entsprechende Flag im Feld *flags* gesetzt wird. Dann werden die Felder X und Y für Latitude- und Longitude-Werte genutzt. Voraussetzung für eine korrekte Funktion ist, daß in der verwendeten Szenendatei eine kartenbezogene Latitude-/Longitude-Position definiert ist. Details zur vereinfachten Latitude-Longitude-Umrechnung werden in Kapitel 4.6 (S. 58) beschrieben.

Die räumliche Orientierung der Fahrzeuge wird in Form von Eulerwinkeln angegeben. Diese sind unabhängig von der vorherigen Fahrzeugorientierung und beziehen sich immer auf die Grundausrichtung, bei der das Fahrzeug waagerecht steht und die Vorderseite nach Norden ausgerichtet ist. Durch diese absoluten Rotationsangaben werden Probleme mit Singularitäten vermieden. Details zu den Rotationsvorschriften befinden sich in Kapitel 2.5 (S. 16). Zu beachten ist, daß das Übertragungsprotokoll bei Version 1 Rotationswerte im Gradmaß erwartet und ab Protokollversion 2 im Bogenmaß.

Wenn eine TCP-Verbindung durch ein Fahrzeugsimulator geschlossen wird, sollte zuvor ein Rahmen geschickt werden, der als Objektname `CV-closeConnection` enthält. Dadurch wird CViewVR mitgeteilt, daß der Server diese Verbindung schließen kann. Grund für diesen speziellen Rahmen ist, daß keine Möglichkeit gefunden wurde, ein korrektes Schließen einer TCP-Verbindung durch die Gegenseite zu erkennen. Nur die Erkennung von Verbindungsabbrüchen ist möglich.

Felder des Rahmens

Startzeichen: Das Startzeichen hat den konstanten Wert 255 und dient zur groben Fehlerprüfung. Hat ein empfangenes Startzeichen einen anderen Wert, dann wurden ungültige Daten übertragen. Das kann bei der Entwicklung eines Clients vorkommen, wenn ein zuvor empfangener Rahmen aufgrund falscher Codierung eine andere Rahmenlänge hatte.

Protokollversion: Dieses Feld wird genutzt, um verschiedene Protokollversionen zu unterscheiden. Dadurch werden zukünftige Protokolländerungen bei gleichzeitiger Abwärtskompatibililät ermöglicht.

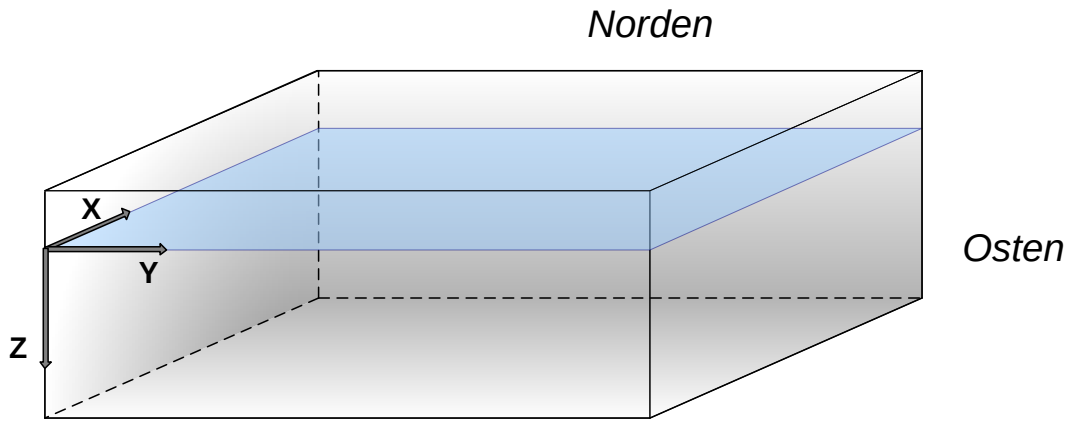


Abbildung 4.7: Szene mit NED-Koordinatensystem, welches für die Fahrzeugpositionierung genutzt wird

Eine Änderung gab es bereits: Bei Version 1 werden die Rotationsfelder (*Yaw*, *Pitch*, *Roll*) als Grad-Werte interpretiert und ab Version 2 als RAD-Werte.

Flags: Dieses Feld dient zur Übertragung von 8 boolischen Flags (Statuszuständen), die jeweils durch 1 Bit repräsentiert werden.

Bisher wird nur das hochwertigste Flag verwendet. Es befindet sich an der Position `x___ ____` (binär) und wird als *LatLong-Flag* bezeichnet. Ist es gesetzt, dann werden die Positionswerte X und Y als Latitude-Longitude-Werte interpretiert.

Hinweis zur Erstellung eines Clients: Gesetzt wird das LatLong-Flag durch `flags = flags | 0x80` für *true*, bzw. durch `flags = flags & 0x7F` für *false*. Geprüft werden kann es durch `if ((flags & 0x80) == 0x80)`.

Szenen-ID: Dieses Feld wird *nicht benutzt*. Ursprünglich sollte damit geprüft werden, ob die Umgebungsmodelle der verbundenen Simulatoren zu der in CViewVR geladenen Szene passen. Die Idee wurde verworfen, da sie in der Praxis keinen Vorteil bringt, sondern stattdessen den Umgang erschwert. Aufgrund der Kompatibilität zu einem bereits fertigen TCP-Client (zur Fahrzeugsteuerung durch eine Matlab-Simulation) wurde das Feld nicht aus dem Rahmen entfernt.

Objektname: Der Objektname gibt an, welches Fahrzeug durch den Rahmen zu steuern ist. Er entspricht dem Eintrag `nodeName` eines Objektes in der Szenendatei. Der Name kann maximal 20 Zeichen lang sein. Im Rahmen wird er als ASCII codiert und bei weniger als 20 Zeichen mit Leerzeichen aufgefüllt. Aufgrund der Beschänkung verwendbarer Zeichen für Objektnamen in Szenendateien sollten

für dieses Feld auch keine anderen verwendet werden. Eine Auflistung zulässiger Zeichen befindet sich in Tabelle 4.3 (S. 48) beim XML-Element `nodeName`, bzw. in der Schema-Datei `scenefile.xsd` von CViewVR.

Position: Durch die Felder *X*, *Y* und *Z* werden Positionsangaben der Fahrzeuge übertragen, die sich auf das kartesische NED-Koordinatensystem beziehen. Die X-Achse zeigt nach Norden, die Y-Achse nach Osten und Z beschreibt die Wassertiefe. Details stehen in Kapitel 2.3 (S. 14). Wenn im *Flags*-Feld das *LatLong-Bit* auf *true* gesetzt ist, dann werden die Positionsangaben als globale Polarkoordinaten bezüglich der Erdkugel interpretiert und von CViewVR umgerechnet. In diesem Fall enthält das Feld *X* den Latitude-Wert und *Y* den von Longitude im Bogenmaß. Hintergründe zur vereinfachten Latitude-Longitude-Umrechnung sind in Kapitel 2.3 (S. 14) und 4.6 (S. 58) beschrieben.

Rotation: Die Orientierung (räumliche Ausrichtung) eines Fahrzeugs wird beim Übertragungsprotokoll in Form von *Eulerwinkeln* übertragen. Für diese werden die Felder *Yaw*, *Pitch* und *Roll* genutzt. Die Werte sind absolut und beziehen sich auf die Grundausrichtung eines Fahrzeugs. Sie sind daher unabhängig von der Orientierung im vorherigen Zeitpunkt. Probleme mit Singularitäten können daher seitens CViewVR nicht auftreten.

Bei der Berechnung der Eulerwinkel im Fahrzeugsimulator müssen Bezug und Reihenfolge der einzelnen Rotationen beachtet werden, da sich sonst das Fahrzeug in der Visualisierungssoftware anders dreht als erwartet. Details dazu befinden sich in Kapitel 2.5 (S. 16). In der Formelsammlung 2.2 (S. 16) stehen die empfohlenen Wertebereiche der Eulerwinkel.

Wichtig: Bei Protokollversion 1 werden die Winkelwerte als Angabe in Grad interpretiert, ab Version 2 als Bogenmaß.

Weg-Schätzung: Die Felder *U*, *V* und *W* werden bisher *nicht genutzt*.
Sie sind dazu gedacht, Fahrzeugzustandsdaten zu übertragen. (z.B. Geschwindigkeit, Beschleunigung, Winkelgeschwindigkeit oder Winkelbeschleunigung). Dadurch soll eine vorausschauende Schätzung für die Wegstrecke ermöglicht werden. Das wäre hilfreich für eine flüssige Bewegungsdarstellung bei kurzzeitigen Verbindungsunterbrechungen (z.B. bei Steuerung über das Internet) oder zur flüssigen Bewegungsdarstellung, wenn ein Fahrzeugsimulator nicht in genügend kurzen Intervallen Daten senden kann (z.B. nur einmal pro Sekunde). Zur Implementierung einer solchen Funktion könnte ein Kalmann-Filter genutzt werden.

4.4.2 Erweiterungsmöglichkeit für Rotation über Quaternionen

Fahrzeugorientierungen werden in Form von Eulerwinkeln übertragen. Alternativ wäre eine Übertragungsmöglichkeit für Quaternionen sinnvoll, da auch diese für Berechnungen in Fahrzeugmodellen üblich sind. Weiterhin wären dadurch Inkompatibilitäten bezüglich der Rotationskonvensionen vermieden. Quaternionen benötigen 4 Parameter zur Lagebeschreibung, jedoch sind nur 3 Felder des Rahmens für die Orientierung vorgesehen. Im Folgenden werden verschiedene Möglichkeiten vorgestellt, wie die Übertragung von Quaternionen realisiert werden kann.

1) Flag für alternative Feldinterpretation

Für den vierten Quaternionsparameter wird ein Feld der Wegschätzung benutzt, etwa das Feld *W*.

Vorteil: Die Variante ist mit geringem Aufwand umzusetzen. Da die Felder zur Wegschätzung bisher ungenutzt sind, gäbe es keine Konflikte.

Nachteil: Eine zukünftige Wegschätzung auf Grundlage der Parameter *U*, *V* und *W* ist bei gleichzeitiger Übertragung von Quaternionsparametern nicht mehr möglich.

2) Aufbau des Protokolls ändern

Das Protokoll wird um ein Feld erweitert, der für den vierten Quaternionsparameter genutzt wird.

Vorteil: Für die Wegschätzung stehen weiterhin alle 3 Parameter zur Verfügung. Zudem könnte das Protokoll an weitere mögliche Anforderungen angepasst werden. Etwa mehr als 3 Parameter zur Wegschätzung, Steuerbefehle für Sensoren, Anzeigeoptionen für die Visualisierungsoftware (z.B. Kamerawechsel durch Simulator), etc.

Nachteil: Eine Abwärtskompatibilität ist nicht mehr gegeben und vorhandene Clients von Fahrzeugsimulatoren funktionieren nicht mehr. Diese müssten dann –sofern möglich– angepasst werden.

3) Adaptives Protokoll

Der Rahmen hat keinen fest vorgegebenen Aufbau. Statt dessen wird ein Statusflag am Rahmenanfang ausgewertet und die folgenden Daten entsprechend unterschiedlich interpretiert.

Vorteil: Es ist möglich, das adaptive Protokoll so zu gestalten, daß der bisherige Rahmenaufbau weiterhin gültig ist und bereits vorhandene Clients von

Fahrzeugsimulatoren ohne Anpassung funktionieren.

Nachteil: Das Protokoll wird deutlich komplexer. Durch die gemischte Verwendung verschiedener Rahmenvarianten mit unterschiedlichem Aufbau und ungleicher Rahmenlänge können sich deutlich schneller Programmierfehler einschleichen, wenn TCP-Server und Client erweitert oder neu erstellt werden. Weiterhin besteht die Gefahr, daß bei der Verwendung von einem Client ein falsch gesetztes Flag gesendet wird und die folgenden Daten falsch interpretiert werden. Die Folge wäre eine unbrauchbare Verbindung und schlimmstenfalls ein Programmabsturz.

4) Version des Protokolls wird vorgegeben

Für jedes Protokoll wird ein anderes Modul zur Verarbeitung verwendet. Bei Benutzung der Visualisierungssoftware muß der Anwender festlegen, welche Protokollversion er verwenden möchte.

Vorteil gegenüber Variante 3): Durch die geringere Komplexität ist die Warscheinlichkeit von Fehlern deutlich geringer. Es ist ebenfalls möglich, auch Clients zu nutzen, die das ältere Protokoll verwenden.

Nachteil: Alle gleichzeitig laufenden Fahrzeugsimulatoren müssen das selbe Protokoll benutzen. Ein Mischbetrieb ist nicht möglich.

5) Parallel laufende Server

Diese Variante ist ähnlich wie 4), mit dem Unterschied, daß in CViewVR mehrere Server parallel laufen und über verschiedene Ports erreichbar sind. Jeder Server ist nur für ein bestimmtes Protokoll zuständig.

Vorteil: Dadurch können verschiedene Protokolle parallel verwendet werden. Anpassungen an Clients mit älteren Protokollen sind nicht nötig.

4.5 Erstellte Hilfsprogramme

Während der Entwicklung des TCP-Servers von CViewVR wurden zwei kommandozeilenbasierte Hilfsprogramme erstellt. Später wurde ein weiteres erstellt, das eine grafische Benutzeroberfläche besitzt. Alle drei Programme befinden sich auf der CD, die dieser Arbeit beiliegt.

Bildschirmaufnahmen der Hilfsprogramme sind im Kapitel *Resultate* zu sehen. Der *WinClient* ist in Abbildung 5.12 (S. 70) dargestellt. Abbildung 5.13 (S. 71) zeigt den *TCP-Server* beim Empfang von Daten eines Clients. In Abbildung 5.14 (S. 71) wird der *TCP-Client* gezeigt, während er Daten aus einer Logdatei sendet, um drei Fahrzeuge zu steuern.

TCP-Server

Das *TCP-Server* genannte Programm nutzt die gleiche (selbst erstellte) Kommunikationsklasse, die in CViewVR verwendet wird. Durch Entkopplung von der Visualisierungssoftware ist es möglich, die korrekte Funktionalität der Kommunikation auf einfache Weise zu prüfen. Nach Programmstart werden automatisch alle Verbindungsanfragen von Clients angenommen und die empfangene Daten in Textform auf der Kommandozeile ausgegeben.

Bei Problemen mit der Steuerung von Fahrzeugen kann geprüft werden, ob die TCP-Verbindung korrekt funktioniert und ob die empfangenen Daten die erwarteten Werte besitzen. Zudem ist es möglich, die empfangenen Binärdaten des TCP-Streams als 1:1 Kopie in eine Datei zu schreiben. Dies ist hilfreich bei der Suche nach Fehlern, die während der Entwicklung eines Clients auftreten können.

Aufgerufen wird das Programm über `server.exe`. Danach ist der Server standardmäßig über IP-Adresse `127.0.0.1` auf Port `8000` erreichbar. Dies ist die gleiche Adresse-Port-Konfiguration wie bei CViewVR, um stellvertretend für die Visualisierung laufen zu können. Das hat den Vorteil, daß für Tests keine Änderungen in der Konfiguration der Clients, bzw. Fahrzeugsimulatoren gemacht werden müssen. Eine Änderung der IP-Adresse oder des Ports ist entweder über die Datei `ipconfig.txt` oder als Kommandozeilenparameter beim Start des Servers möglich.

Der Parameter `-bin` aktiviert den binären Empfangsmodus. Dieser schreibt die empfangenen Daten in die Dateien `binaryStreamLog.txt` und `binaryStreamLog2.txt`. Grundsätzlich haben beide den gleichen Inhalt, jedoch mit dem Unterschied, daß bei der zweiten Logdatei nach jeder erwarteten Rahmenlänge (96 Byte) ein Leerzeichen eingefügt wird. Das ist eine Erleichterung, um bei der Entwicklung eines Clients schnell prüfen zu können, ob die Rahmenlänge korrekt ist. Nach dem Leerzeichen muß immer der hexadezimale Wert `FF` stehen.

Die Syntax lautet: `server.exe [<Adresse>[:<Port>]] [-bin]`

TCP-Client

Das Kommandozeilenprogramm *TCP-Client* versucht sich mit CViewVR bzw. dem *TCP-Server* zu verbinden und sendet anschließend aufgezeichnete oder generierte Fahrzeugsteuerdaten. Wenn gerade kein Server läuft, wird in regelmäßigen Abständen geprüft, ob einer gestartet wurde. Sobald eine Verbindung hergestellt werden konnte, werden über diese Daten gesendet. Dabei gibt es zwei Modi:

Beim einfachen Aufruf werden in einer Endlosschleife (bis Programmabbruch) Steu-

erdaten für zwei Fahrzeuge gesendet. Ein Fahrzeug fährt im Kreis und das andere bewegt sich nicht. Dieser Modus ist geeignet für eine einfache Prüfung, ob bei rechnerübergreifender Kommunikation eine Verbindung zustande kommt.

Alternativ kann der *TCP-Client* über einen Parameter auf eine Logdatei zugreifen und die enthaltenen Steuerdaten an die Visualisierungssoftware senden. Dadurch ist es möglich, in CViewVR aufgezeichnete Fahrzeugbewegungen wiederzugeben. Über einen optionalen Parameter läßt sich die „Abspielgeschwindigkeit“ erhöhen. Beispielsweise bewirkt der Parameter `s4` eine vierfache Geschwindigkeit.

IP-Adresse und Port werden aus der Datei `ipconfig.txt` gelesen (standardmäßig `127.0.0.1:8000`) oder über einen Parameter vorgegeben.

Die Syntax lautet:

```
client.exe [<Adresse>[:<Port>]] [<Datei> [-s<Geschwindigkeit>]]
```

WinClient

Das Programm *WinClient* hat eine grafische Oberfläche und kann sich über die TCP-Schnittstelle mit CViewVR verbinden. Es wurde erstellt, um Fahrzeuge und andere Objekte manuell steuern zu können. Ein Zweck ist die Positionierung von Objekten bei der Erstellung und Modifikation von Szenen. Dies ist wesentlich komfortabler als manuelle Änderungen an XML-basierten Szenendateien – besonders im Bezug auf Orientierungangaben, die dort in Form von Quaternionen angegeben werden müssen. Ein weiterer Zweck des *WinClient* ist die Nutzung als primitiver Fahrzeugsimulator, mit dem Fahrzeuge bewegt werden können. Änderungen von Position und Orientierung können durch Schieberegler oder direkte Eingabe von Zahlen erfolgen. Die Sensibilität der Schieberegler kann über drei Stufen eingestellt werden: kleine, mittlere und große Änderungen. Positionsangaben können als NED- oder Latitude-Longitude-Werte angegeben werden. Rotationsangaben durch Eulerwinkel erfolgen standardmäßig in Grad oder alternativ im Bogenmaß. Häufig genutzte Fahrzeug- und Objektnamen können über eine vordefinierte Liste ausgewählt werden. Andere Namen können direkt in den *WinClient* eingegeben oder zur Liste hinzugefügt werden. Die gesendeten Daten werden in Textform angezeigt, um dem Nutzer eine direkte Kontrolle zu ermöglichen.

Eine Bildschirmaufnahme des *WinClients* ist in Abbildung 5.12 (S. 70) zu sehen.

4.6 Vereinfachte Latitude-Longitude-Umrechnung

Es wurde eine stark vereinfachte Lat-Long-Umrechnung implementiert. Dabei wurde angenommen, daß die Erde eine ideale Kugel ist. In einem stark begrenzen Bereich der Erdoberfläche (z.B. 10 x 10 km^2) ist der Kugelausschnitt näherungsweise eine Ebene. Somit die räumliche Verzerrung durch die Abbildung von $\Re 3$ auf $\Re 2$ sehr gering. Wenn man in diesem Bereich die Ungenauigkeit ignoriert, dann können konstante Umrechnungsfaktoren verwendet werden, um globale (polare) Lat-Long-Koordinaten in lokale (kartesische) X-Y-Koordinaten umzurechnen. Die lokalen X-Y-Koordinaten beziehen sich dabei auf einen Referenzpunkt (lat_{ref}, $long_{ref}$), welcher beschreibt, wo sich der abgebildete Bereich auf der Erdoberfläche befindet. Die Umrechungsfaktoren besagen, daß eine Einheit im Breitengrad (Latitude) einem Abstand von la Metern entspricht, bzw. eine Einheit im Längengrad (Longitude) einem Abstand von lo Metern. Während la konstant ist, hängt lo vom Breitengrad lat_{ref} des Referenzpunktes ab.

Die Formeln in 4.1 zeigen die Berechnung der Umrechnungsfaktoren la, lo und die Formeln in 4.2 die vereinfachte Umrechnung von Lat-Long-Koordinaten in lokale X-Y-Koordinaten. Die Konstante $EU = 40\,074\,156$ $[m]$ ist der Erdumfang (laut Wissensdatenbank `wolframalpha.com`) und die Variablen lat und $long$ beschreiben eine Fahrzeugposition in Polar-Koordinaten mit der Einheit RAD. Das RAD-Maß wurde verwendet, da Fahrzeugsimulatoren im Fachgebiet Systemanalyse der TU Ilmenau, sowie im Fraunhofer AST damit arbeiten.

Wenn dem Gelände einer Szene ein Referenzpunkt (lat_{ref}, $long_{ref}$) zugeordnet wurde, ist durch die vereinfachte Koordinatenumrechnung eine Fahrzeugpositionierung innerhalb einer Szene möglich. Abbildung 4.8 zeigt eine Karte mit beiden Koordinatensystemen, Referenzpunkt und einer beispielhaften Fahrzeugposition.

Eine Umrechnung von globalen Altitude-Werten war nicht vorgesehen, ließe sich jedoch bei Bedarf nachträglich implementieren. Vertikale Fahrzeugpositionen entsprechen der Tauchtiefe in Metern.

$$\begin{aligned} la &= \frac{EU}{\pi} = 12.756.000 && [m/RAD] \\ lo &= \cos(lat_{ref}) \cdot \frac{EU}{2 \cdot \pi} && [m/RAD] \end{aligned} \tag{4.1}$$

$$\begin{aligned} x &= (lat - lat_{ref}) \cdot la && [m] \\ y &= (long - long_{ref}) \cdot lo && [m] \end{aligned} \tag{4.2}$$

Hintergründe und Verweise auf weiterführende Quellen bezüglich globaler Koordinatensysteme sind im Kapitel 2.4 (S. 15) nachzulesen.

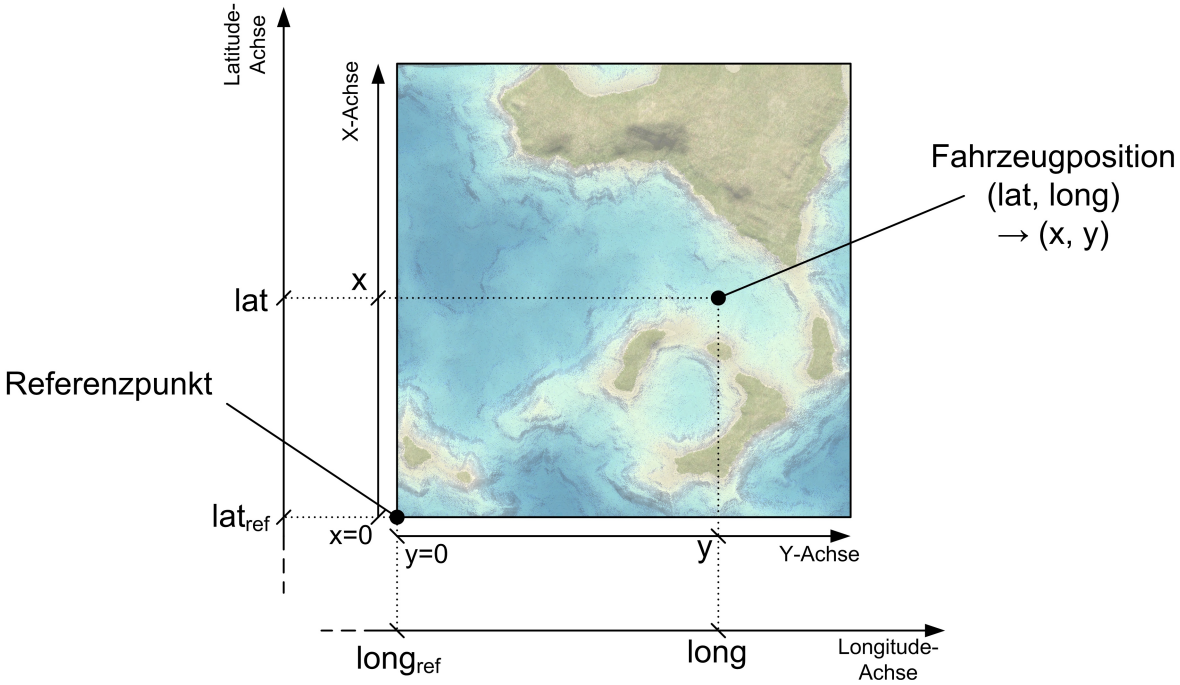


Abbildung 4.8: Karte einer Szene mit Referenzpunkt für Lat-Long-Umrechnung

4.7 Programmabläufe

In den Abbildungen 4.9 bis 4.12 sind wichtige Programmabläufe der Visualisierungssoftware CViewVR dargestellt. Aufgrund der Komplexität und Übersichtlichkeit wurden diese auf die wesentliche Funktionalität reduziert.

Programmstart

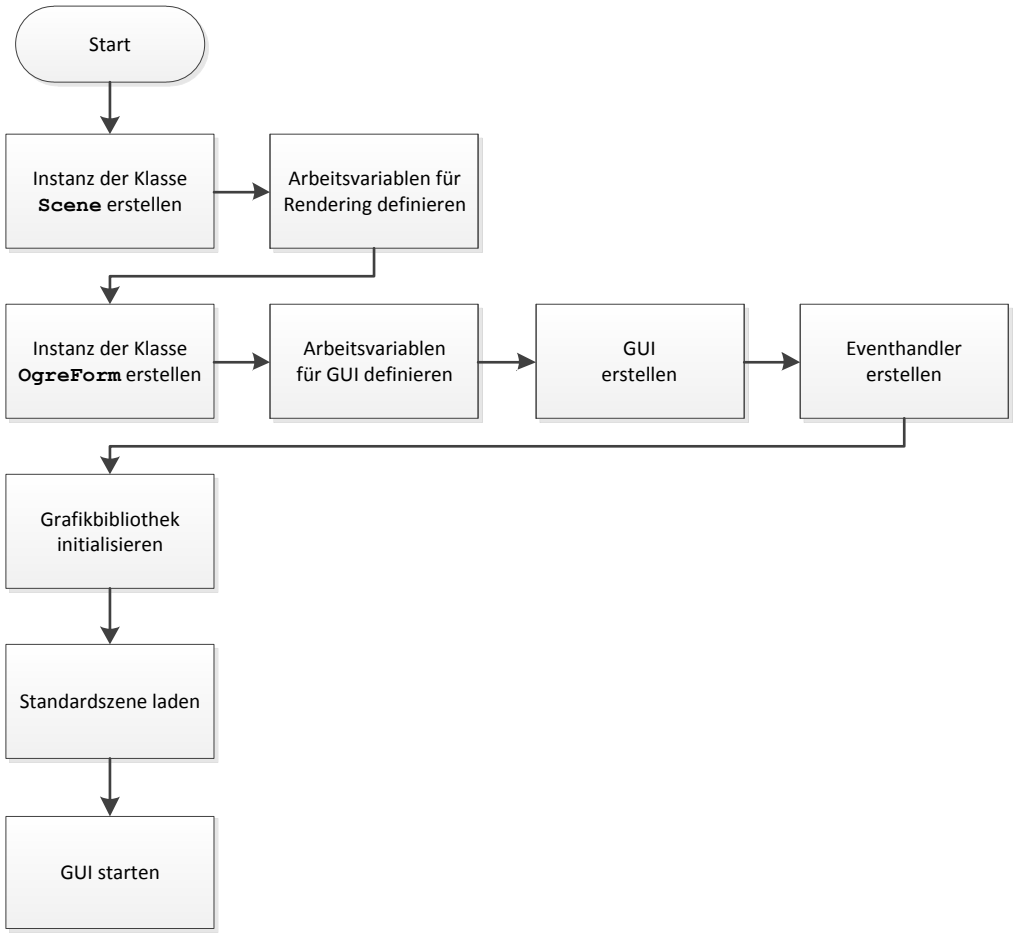


Abbildung 4.9: PAP Programmstart

Renderingstart

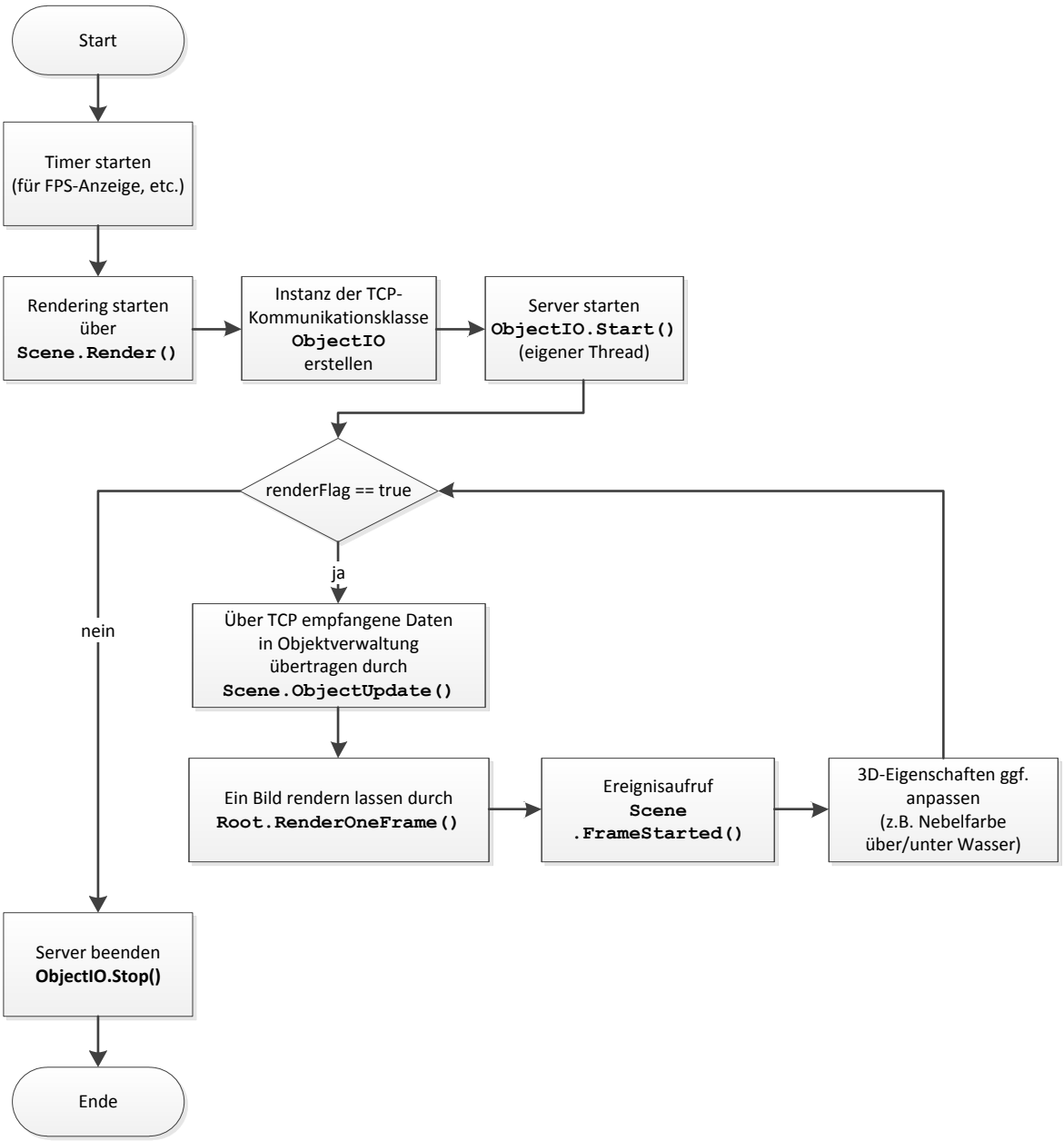


Abbildung 4.10: PAP Renderingstart

Serverstart

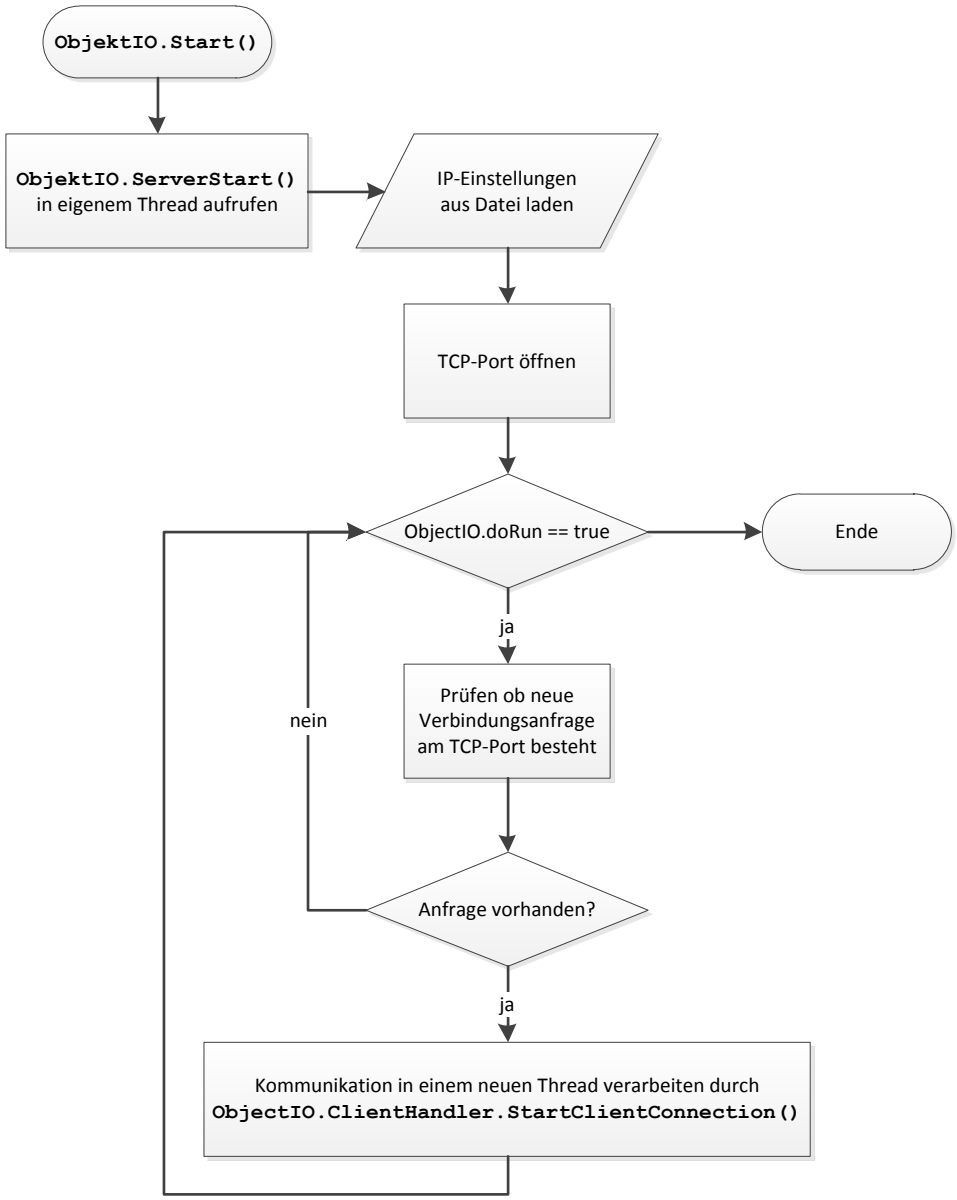


Abbildung 4.11: PAP Serverstart

Empfang von Steuerdaten für Fahrzeuge über TCP

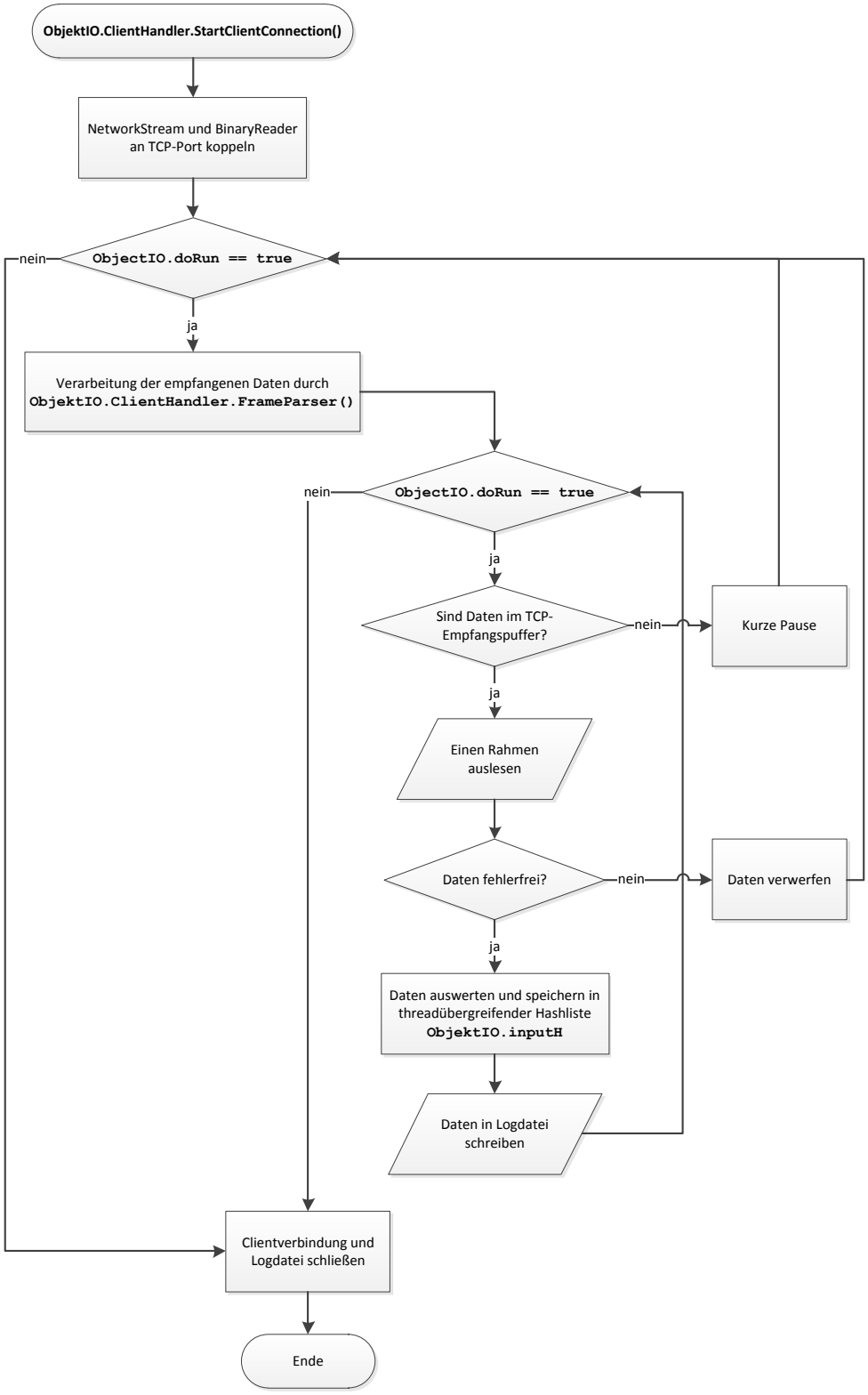


Abbildung 4.12: PAP Datenempfang vom Client

5 Anwendungsbeispiele & Screenshots

Über die TCP-basierte Schnittstelle der entstandenen Visualisierungssoftware CViewVR können sich externe Programme (z.B. Fahrzeugsimulatoren) verbinden. Abbildung 5.1 zeigt auf der rechten Seite den inneren Aufbau von CViewVR. Auf der linken Seite befinden sich drei beispielhafte Programme (Clients) abgebildet, die mit der Visualisierungssoftware verbunden sind. Links oben ist ein Simulator zu sehen, der CViewVR nur zur visuellen Darstellung nutzt. Darunter befindet sich ein Simulator, der den Rückkanal der bidirektionalen TCP-Verbindung verwendet. Dies soll zukünftig genutzt werden, um Informationen über die verwendete Szene oder Daten von simulierten Sensoren zum Simulator zu übertragen. Ein Anwendungsbeispiel wäre die Simulation von Sonarsensoren, die an Fahrzeuge gekoppelt sind und mit der virtuellen Umgebung von CViewVR interagieren. In der Abbildung links unten ist ein Programm dargestellt, welches manuelle Steuerbefehle (via Joystick) von einem Nutzer entgegennimmt, auf ein Fahrzeugmodell anwendet und die resultierenden Fahrzeugbewegungen an die Visualisierungssoftware sendet.

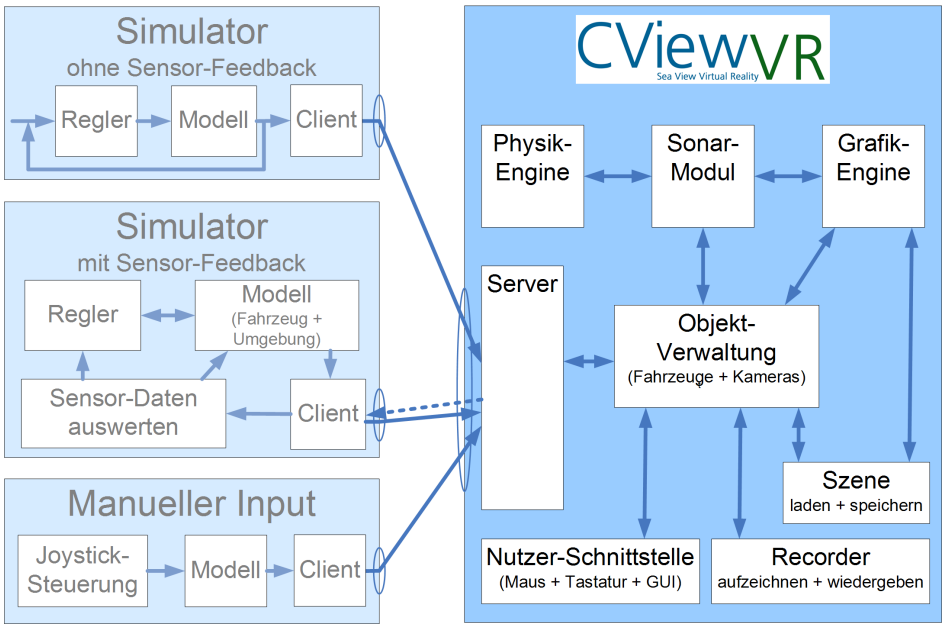


Abbildung 5.1: Aufbau der Software CViewVR (rechts) und beispielhafte Clients (links)

Bildschirmfotos

Die folgenden Abbildungen zeigen die Visualisierungssoftware CViewVR. Abbildung 5.2 zeigt drei Unterwasserfahrzeuge und ein Oberflächenfahrzeug. Abbildung 5.3 zeigt ein AUV bei der Suche nach Objekten. In Abbildung 5.4 ist die Ansicht einer fahrzeuggebundenen Kamera dargestellt und Abbildung 5.5 zeigt eine Szene mit deaktivierter Wasseroberfläche aus der Vogelperspektive.

Abbildung 5.2: CViewVR: Fahrzeuge Delfim (links oben), Seebär (links mittig), Infante (links unten) und Seewolf (rechts) Fahrzeuge 1 und 3 sind von Instituto Superior Técnico, Lissabon, Portugal; Fahrzeug 4 ist von Atlas Elektronik, Bremen

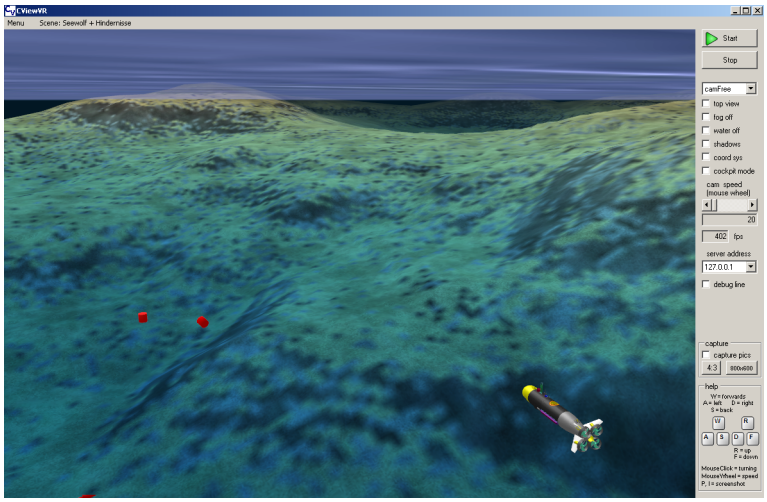


Abbildung 5.3: CViewVR: Fahrzeug Seewolf und zu suchende Objekte

Abbildung 5.4: CViewVR: Ansicht einer fahrzeuggebundenen Kamera; das Schiffmodell stammt aus [URL 21]

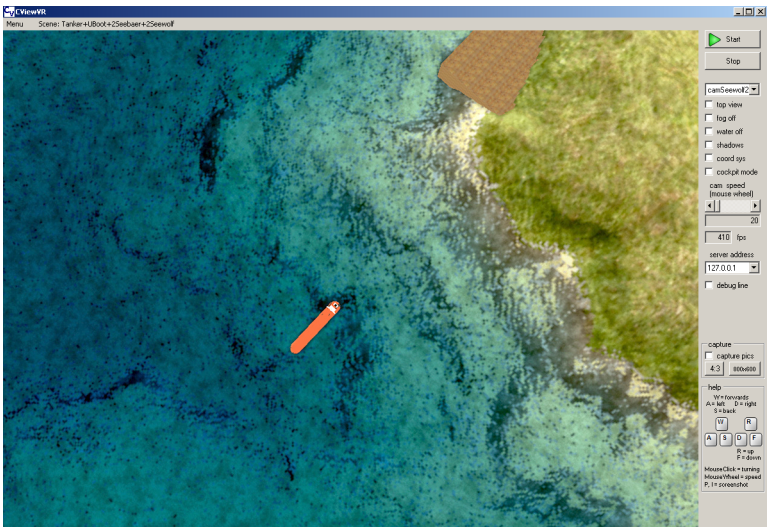


Abbildung 5.5: CViewVR: Vogelperspektive bei deaktivierter Wasseroberfläche

Auf der folgenden Seite sind drei weitere Abbildungen (5.6 bis 5.8). Die obere zeigt einen Vergleich mit und ohne Nebeleffekt. Die mittlere Abbildung zeigt den Unterschied mit und ohne Wassertrübung. Nebel und Wassertrübung sind standardmäßig aktiviert. Besonders unter Wasser ist die Nebelfunktion wichtig für eine realistische Wirkung. Die Option zur Deaktivierung des Nebels ist eingebaut, um sich bei Bedarf besser orientieren zu können. Unterschiede bezüglich der Rechenleistung wurden nicht festgestellt. In der dritten Abbildung ist das lokale Fahrzeugkoordinatensystem zu sehen, welches über die Option *coord sys* eingeblendet wurde.

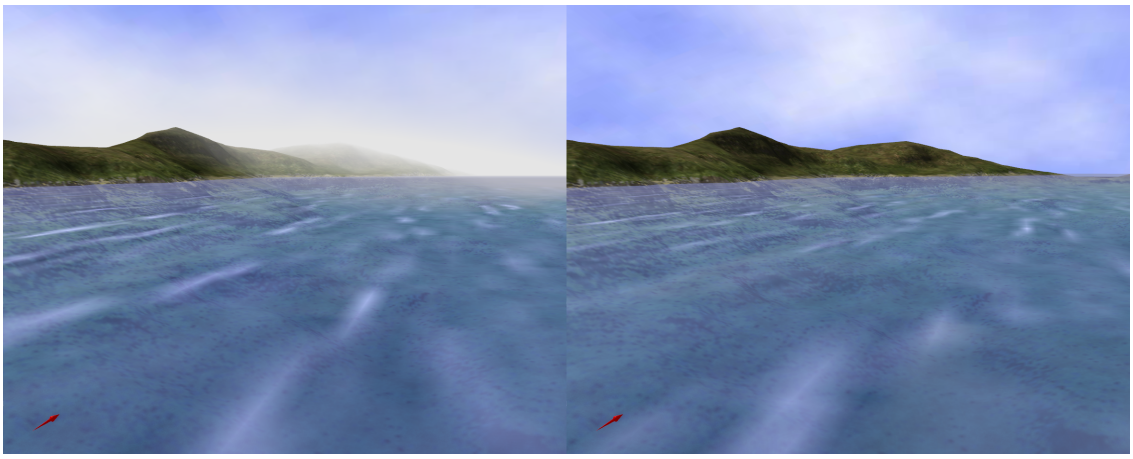


Abbildung 5.6: Szene über Wasser – links: mit Nebeleffekt (Standard), rechts: gleiche Szene ohne Nebeleffekt (Option *fog off*)

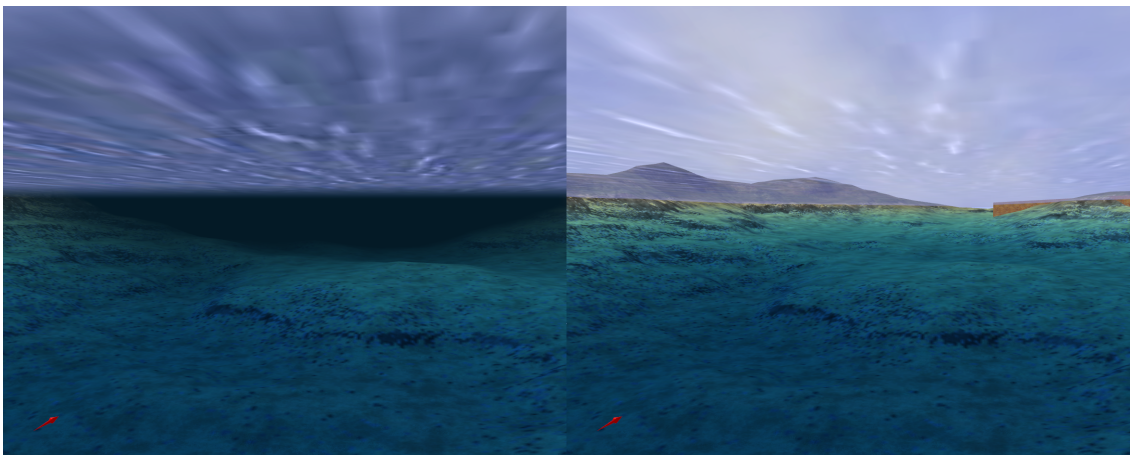


Abbildung 5.7: Szene unter Wasser – links: Wassertrübung aktiviert (Standard), rechts: gleiche Szene ohne Wassertrübung (Option *fog off*)

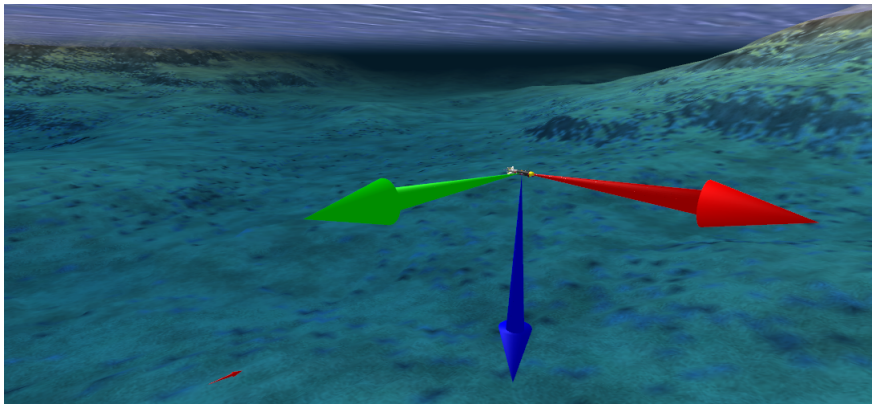


Abbildung 5.8: Fahrzeug mit Koordinatensystem bei aktivierter Option *coord sys*

Integration von Modellen

Eine Teilaufgabe dieser Studienjahresarbeit war zu untersuchen, wie vorhandene 3D-Modelle in die Visualisierungssoftware integriert werden können. Die folgenden Abbildungen zeigen den Integrationsprozess beispielhaft an einem Schiffsmodell [URL 21]. Abbildung 5.9 zeigt auf der linken Seite das Konvertierungsprogramm *AccuTrans3D*, welches das Modell vom VRML98-Format in das 3ds-Format konvertiert. Dieses Format kann vom Modellierungsprogramm *Blender* (rechts) geladen werden. Nach den nötigen Anpassungen des Modells (u.a. die räumliche Ausrichtung entsprechend des NED-Koordinatensystems) kann Blender das Modell im Ogre-Format speichern. Anschließend ist es möglich, das Modell in Szenendateien einzubinden. Abbildung 5.10 zeigt das Schiff innerhalb der Visualisierungssoftware CViewVR.

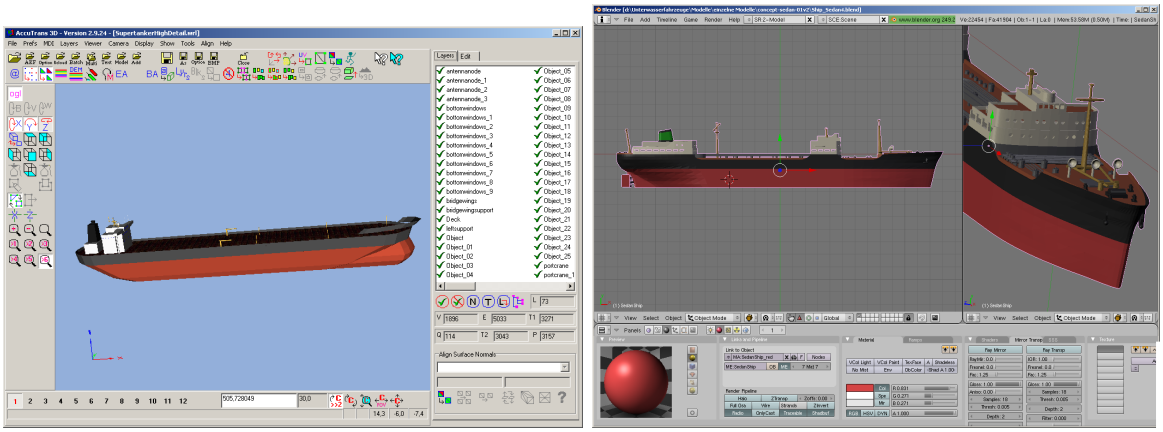


Abbildung 5.9: Schiffmodell in AccuTrans3D (links) und Blender (rechts)

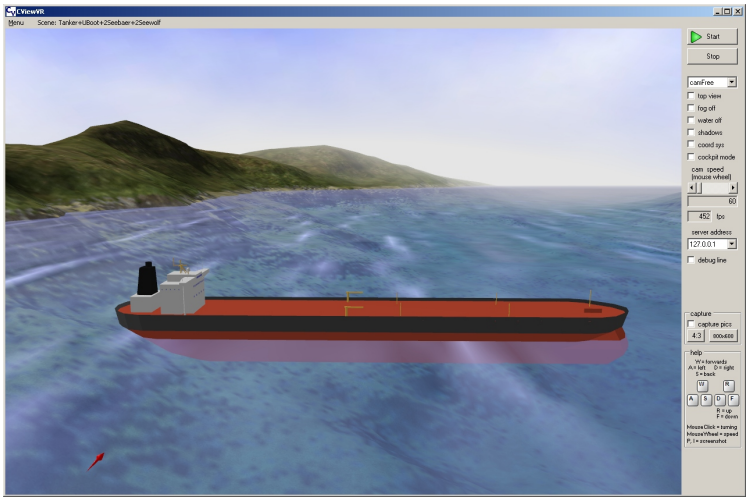


Abbildung 5.10: Schiffmodell nach Integration in CViewVR

Anbindung an Matlab/Simulink

Simulatoren werden während der Entwicklungsphase häufig mit der Simulationssoftware Matlab/Simulink erstellt. Für die Anbindung von Simulink-Modellen an die Visualisierungssoftware CViewVR wurde von Mike Eichhorn (Angestellter der TU Ilmenau im Fachgebiet Systemanalyse) eine Client-Software programmiert. Dadurch steht ein spezieller „Block“ zur Verfügung, der in Simulink-Modellen verwendet werden kann. Dieser hat drei Eingänge für Position (x, y, z), Bewegungsschätzung (u, v, w) und Rotation (yaw, pitch, roll), besitzt einen Konfigurationsdialog und kann Fahrzeuge steuern, die in CViewVR dargestellt werden.

Abbildung 5.11 zeigt beispielhaft ein sehr einfaches Simulink-Modell, welches eine Fahrzeugbewegung generiert. Im oberen Teil werden die Positionen und im unteren Teil die Rotationen zeitabhängig berechnet. Auf der Rechten Seite ist der Simulink-Block zu sehen, der die Steuerdaten entgegennimmt und an CViewVR sendet.

Verwendet wurde die Simulink-Anbindung an der TU Ilmenau im Zusammenhang mit dem EU-Forschungsprojekt GREX [URL 2], bei dem eine in Matlab/Simulink erstellte Simulationssoftware das Teamverhalten von mehreren Unterwasserfahrzeugen koordiniert. CViewVR wurde zur Visualisierung verwendet, um einerseits das Verhalten anschaulich darzustellen und anderseits um Videos für Präsentationen zu erstellen. Weiterhin gab es eine Nutzung der Simulink-Anbindung im Fraunhofer Anwendungszentrum für Systemtechnik [URL 29], um die Ausgaben eines Simulator-Prototyps visuell zu überprüfen, der die Fahrzeugdynamik eines Geländefahrzeuges simuliert.

Weitere Informationen zur Einbindung und Verwendung des Matlab/Simulink-Clients sind im Anhang C (S. 89) nachzulesen.

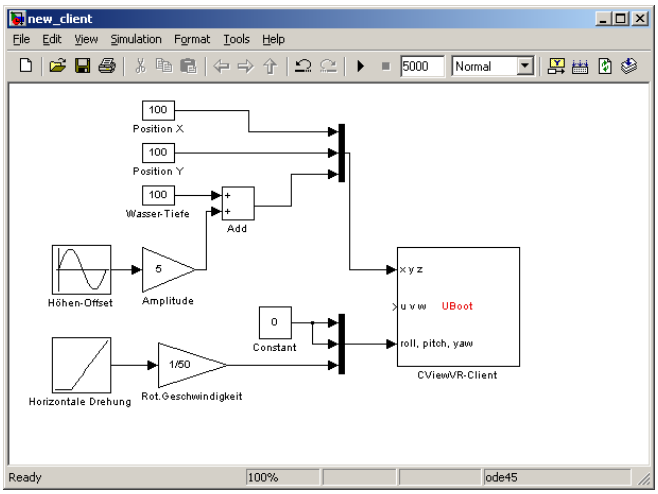


Abbildung 5.11: Ein einfaches Simulink-Modell

WinClient und TCP-Hilfsprogramme

Während der Entwicklung von CViewVR wurden drei Hilfsprogramme erstellt. Der *WinClient* (Abbildung 5.12) kann sich über die TCP-Schnittstelle mit CViewVR verbinden. Anschließend können Objekte der Szene verschoben und rotiert werden. Das Programm dient einerseits zur vereinfachten Manipulation von Szenen und anderseits zur vereinfachten Simulation von Fahrzeugbewegungen.

Die zwei weiteren Programme haben keine grafische Oberfläche, sondern werden über die Kommandozeile oder durch Batch-Dateien (`*.bat`) ausgeführt. Optionen werden als Kommandozeilenparameter übergeben. Das Programm *TCP-Server* enthält die gleiche Kommunikationsschnittstelle wie CViewVR und kann bei Problemen zur Untersuchung der empfangenen Fahrzeugsteuerdaten genutzt werden. Abbildung 5.13 zeigt das Programm beim Empfang von Daten, die ein per TCP verbundener Fahrzeugsimulator sendet. Das Programm *TCP-Client* kann einen Fahrzeugsimulator emulieren und Steuerdaten an CViewVR senden. Abbildung 5.14 zeigt das Programm beim „Abspielen“ einer aufgezeichneten „Mission“ mit drei Fahrzeugen.

Detaillierte Informationen zu den Hilfsprogrammen stehen in Kapitel 4.5 (S. 55).

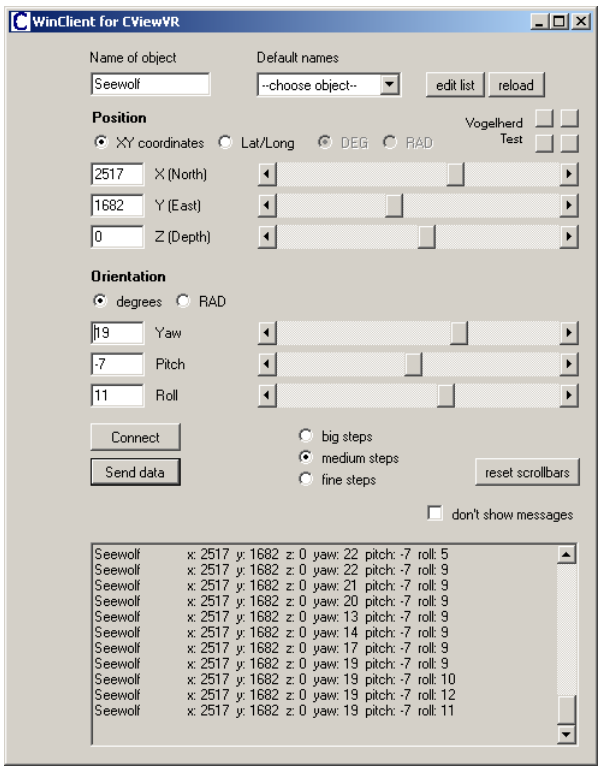


Abbildung 5.12: Zusatzprogramm *WinClient* zur manuellen Positionierung und Steuerung von Fahrzeugen über die TCP-Schnittstelle

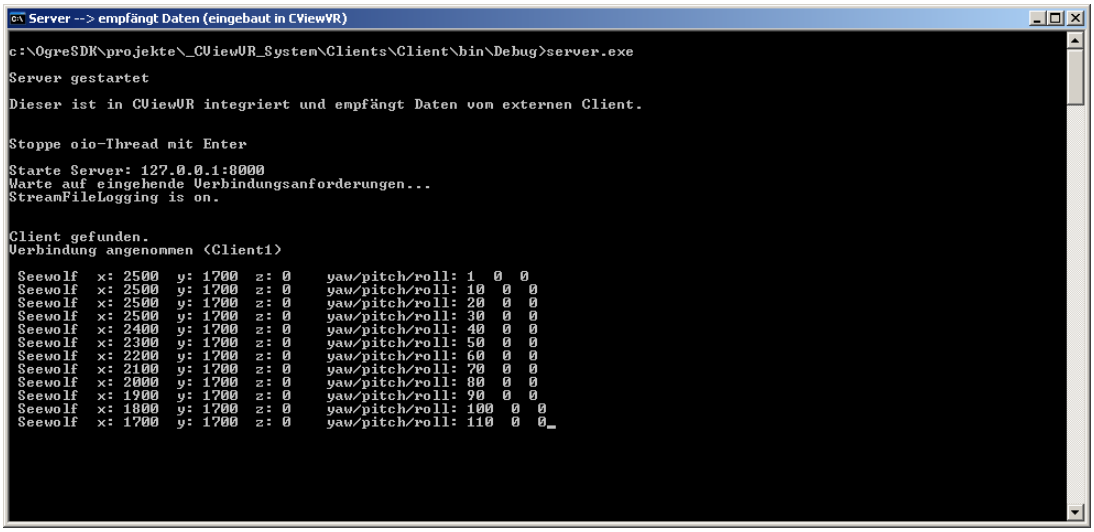


Abbildung 5.13: Kommandozeilenprogramm *TCP-Server* beim Empfang von Steuerungsdaten

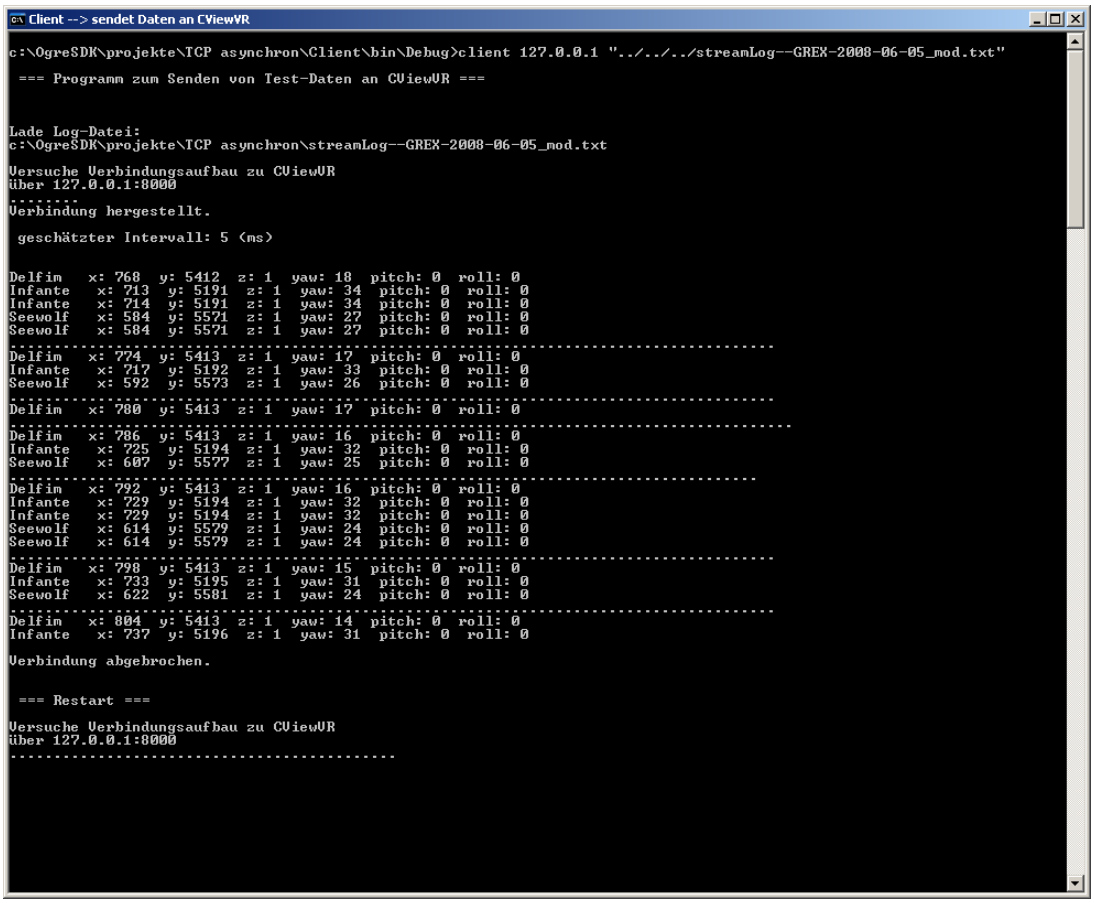


Abbildung 5.14: Kommandozeilenprogramm *TCP-Client* beim Senden von Steuerungsdaten, zur Wiedergabe einer Mission mit 3 Fahrzeugen, die zuvor in CViewVR aufgezeichnet wurde

6 Zusammenfassung

Hintergrund

Diese Studienjahresarbeit befasste sich mit dem Thema *Visualisierung von Unterwasserfahrzeugen*. Hintergrund ist der zunehmende Bedarf an unbemannten Unterwasserfahrzeugen für vielseitige Zwecke.

Die notwendigen Fahrzeugführungssysteme sind komplex und deren Entwicklungsaufwand ist groß. Zur Prüfung der korrekten Funktion müssen teure, zeitaufwändige Tests in Gewässern durchgeführt werden und die Beobachtung kann durch Wassertrübung erschwert sein. Zudem können Kollisionen durch Fehlfunktion schwere Schäden verursachen und bei „verlorengegangenen“ Fahrzeugen müssen diese gesucht und diese im Falle eines Untergangs geborgen werden.

Für die Entwicklung, Validierung und Optimierung von Führungssoftware autonomer Fahrzeuge ist daher eine Computersimulation hilfreich, die ein Modell der Fahrzeugdynamik beinhaltet und das Bewegungsverhalten darstellt. Bereits vorhandene Fahrzeugsimulatoren –etwa im Fachgebiet Systemanalyse der TU Ilmenau– geben nur einfache Grafiken aus. Beispielsweise den zeitlichen Verlauf der Geschwindigkeit, Tauchtiefe und eine 2D-Grafik zum Abbilden der gefahrenen Route aus der Vogelperspektive.

Aufgabe

Grundlegendes Ziel der Studienjahresarbeit war, nach Möglichkeiten zu suchen, um eine realistisch wirkende Visualisierung von Unterwasserszenarien zu ermöglichen und das Fahrzeugverhalten darzustellen. Wichtige Anforderungen waren die Integration existierender 3D-Modelle von Unterwasserfahrzeugen und eine Netzwerkschnittstelle, die Fahrzeugsimulatoren nutzen können, um Fahrzeuge in der visualisierten Unterwasserszene zu steuern. Für die Übertragung von Positionsangaben sollte das North-East-Down-Koordinatensystem (NED) verwendet werden und die räumliche Lagebeschreibung in Form von Eulerwinkeln erfolgen. Eine Erweiterbarkeit für die Integration zusätzlicher Funktionalität war erwünscht, um beispielsweise Kollisionen zu erkennen oder Sensoren zu simulieren.

Rechercheergebnisse

Zunächst wurde eine direkte Integration in die Simulationssoftware *Matlab/Simulink* angestrebt, doch es stellte sich heraus, daß eine für 3D-Darstellung notwendige Teilkomponente nicht mit der verwendeten Matlab-Version kompatibel ist. Zudem war die Nutzerinteraktion mit der Visualisierung umständlich.

Danach wurde nach bereits vorhandener Software recherchiert, die sich für die Erfüllung der Anforderungen eignet. Für maritime Visualisierung gab es nur wenig Software und deren Produktbeschreibungen enthielten kaum Details. Da sie kommerziell bzw. unverkäuflich waren, wurden sie nicht weiter untersucht.

Weiterhin wurde nach Software recherchiert, die zur Entwicklung von Computerspielen angeboten wird. Zum einen gibt es umfangreiche Entwicklungsumgebungen für Spiele und zum anderen eigenständige Level-Editoren zur Erstellung von Szenen. Als Nachteil erwies sich, daß Software für Spiele überwiegend proprietäre Datenformate verwendet, wodurch ein Import und Export von Modellen kaum möglich ist. Zudem sind eigene Erweiterungen der Software nur beschränkt machbar.

Eine Alternative war die eigene Entwicklung einer Software, die eine freie Grafikbibliothek für die 3D-Visualisierung verwendet. Nach problembehafteten Versuchen mit *Irrlicht* erwies sich die Grafikbibliothek *Ogre* als erfolgversprechender, gut dokumentierter Kandidat. Zudem gab es im Fraunhofer Anwendungszentrum für Systemtechnik (AST) bereits eine Untersuchung, die sich mit der Darstellung von Unterwasserszenen beschäftigt. Ergebnis war ein sehr einfacher Softwareprototyp, der die Ogre-Engine verwendet und eine Szene darstellt.

Entstandene Software

Auf dem einfachen Prototyp aufbauend wurde eine Visualisierungssoftware entwickelt, die die Anforderungen der Aufgabenstellung erfüllt. Der Quellcode wurde an vielen Stellen überarbeitet und umfangreich erweitert. Die Bedienung wurde verbessert und viele Fehlerquellen beseitigt, um Abstürze zu vermeiden und bei Problemen konkrete Fehlermeldungen auszugeben. Der Anwender kann Unterwasserszenen aus beliebigen Blickwinkeln sehen und hat die Möglichkeit, auf fahrzeuggebundene Kameras zu wechseln, um beispielsweise Fahrzeugbewegungen aus „Fahrersicht“ zu betrachten.

Eine auf TCP basierende Netzwerkschnittstelle wurde implementiert, über die sich Fahrzeugsimulatoren mit der CViewVR genannten Visualisierungssoftware verbinden und Fahrzeuge steuern können. Eine gleichzeitige Anbindung mehrerer Simulatoren ist möglich und Fahrzeugbewegungen können sowohl aufgezeichnet als auch wiedergegeben

werden. Zur Anbindung von Simulationssoftware stehen Clientmodule für Matlab/Simulink, C# und C++ zur Verfügung. Die Dokumentation des entworfenen Datenprotokolls kann genutzt werden, um bei Bedarf weitere Clientanbindungen zu erstellen.

Szenarien mit Gelände, Bauwerken und Fahrzeugen können in XML-basierten Szenendateien hinterlegt werden. Für eine korrekte Darstellung und Steuerung müssen einzubindende 3D-Modelle erst modifiziert und dann in das Datenformat der Grafikengine Ogre überführt werden. Es wurde untersucht und ausführlich beschrieben, wie dabei vorzugehen und worauf zu achten ist. Zudem wurden bereits vorhandene Modelle von Unterwasserfahrzeugen in die Visualisierungssoftware integriert und verschiedene Szenarien erstellt.

Für Fahrzeugsimulatoren, die Positionsangaben als globale Latitude-Longitude-Werte ausgeben, wurde eine vereinfachte Umrechnung in das szenengebundene, kartesische Koordinatensystem implementiert. Informationen zu den verwendeten Koordinatensystemen, deren Umrechnung und die Rotationsvorschriften wurden im Grundlagenkapitel dokumentiert.

Praktische Anwendung

Bereits während der Entwicklung wurde die Visualisierungssoftware praktisch verwendet. Zum einen im Zusammenhang mit dem EU-Forschungsprojekt GREX, an dem das Fachgebiet Systemanalyse der TU Ilmenau beteiligt ist. Dabei wurde eine Gruppe von Fahrzeugen über eine Simulation in Matlab/Simulink gesteuert. Der eingebettete Client zur Kommunikation mit CViewVR wurde von einem Mitarbeiter der TU-Ilmenau erstellt. Eine weitere praktische Verwendung der Visualisierungssoftware fand im Fraunhofer AST statt, bei der die Ausgaben eines Simulators für Fahrzeugdynamik untersucht wurde. Zwar handelte es sich nicht um ein Unterwasserfahrzeug, doch das Prinzip der Kommunikation und grafischen Darstellung war gleich. Im Rahmen dieser Zusammenarbeit entstand ein Clientmodul zur Anbindung von in C++ geschriebenen Simulatoren.

Sowohl das GREX-Projekt, als auch der Dynamiksimulator waren hilfreich bei der Entwicklung der Visualisierungssoftware und zeigten, daß damit eine realistisch wirkende Darstellung von Simulatorberechnungen möglich ist. Dadurch kann unter anderem die Qualität von Fahrzeugführungssystemen untersucht werden.

Ausblick

Es wurde gezeigt, daß die entstandene Software CViewVR Unterwasserszenarien visualisieren kann und die Möglichkeit bietet, darin enthaltenen Fahrzeuge durch extern angebundene Simulatoren steuern zu lassen. Darauf aufbauend wäre es sinnvoll, einen Szeneneditor zu integrieren, der die Erstellung und Modifikationen von Szenarien vereinfacht. Weiterhin kann die Visualisierungssoftware als Basis für zukünftige Erweiterungen genutzt werden. Beispielsweise um Kollisionen zu erkennen, verschiedene Sensoren zu simulieren oder Umwelteinflüsse wie Wasserströmungen nachzubilden.

Im Fraunhofer AST wurde bereits damit begonnen, ein Modul für die vereinfachte Simulation von Sonarsensoren zu integrieren. Die in CViewVR simulierten Sensormesswerte werden dann an die externen Fahrzeugsimulatoren weitergereicht. Eine Veröffentlichung zur grundsätzlichen Funktionsweise befindet sich in [URL 1].

Eine weitere –wenn auch zweckentfremdete– Verwendung fand die Software, indem sie für die Simulation von Solaranlagen zur Herstellung regenerativer Energie genutzt wurde. Dabei wurden die standort- und zeitabhängigen Sonnenstände zu verschiedenen Jahreszeiten berechnet, um zu untersuchen, welchen Einfluß der Schattenwurf benachbarter Gebäude auf die Solaranlagen hat.

Die Visualisierungssoftware wurde auch für Planungen bezüglich eines Tauchbeckens für Unterwasserfahrzeuge verwendet. Es ging es darum, eine Unterwasserbeleuchtung zu installieren und deren Wirkung im Vorfeld zu veranschaulichen. Die verwendeten Leuchtmittel hatten einen bestimmten Öffnungswinkel und abhängig von Position und Richtung ergaben sich unterschiedliche Beleuchtungsverhältnisse.

Quellen

Gedruckte Literatur

[1] Bericht über Grafikkarten; enthält Angaben über Bildraten (fps) und menschliche Bewegungswahrnehmung
c't Magazin 23/2008 S. 138ff
Heise Zeitschriften Verlag, Hannover

[2] Spielefabrik - 3D-Entwicklungsumgebung Unity 2.0
c't Magazin 23/2007 S. 58f
Heise Zeitschriften Verlag, Hannover

[3] Vergleich von Entwicklungsumgebungen für Spiele
c't Magazin 12/2005 S. 164ff
Heise Zeitschriften Verlag, Hannover

[4] Vergleich von Level-Editoren für Spiele
c't Magazin 12/2005 S. 156ff
Heise Zeitschriften Verlag, Hannover

[5] Gregory Junker: Pro OGRE 3D programming
Verlag Apress, 2006, New York City
ISBN 978-1590597101

[6] Andreas Kühnel: Visual C# 2005 - Das umfassende Handbuch
Verlag Galileo Computing, 2006, Bonn
(auch als kostenloses e-book – siehe [URL 67])
ISBN 978-3898425865

[7] Das Blender-Buch: 3D-Grafik und Animation mit freier Software
Carsten Wartmann, dpunkt-Verlag, Heidelberg
ISBN 978-3898644662, 3. Auflage

Web-Links

[URL 1] CViewVR: A High-performance Visualization Tool for Team-oriented Missions of Unmanned Marine Vehicles
Autoren: Andreas Voigt, Thomas Glotzbach
COMPIT 2009, Budapest/Ungarn S. 150ff
`www.compit.info` (Website)
`www.ssi.tu-harburg.de/doc/webseiten_dokumente/compit/ ...`
`... dokumente/compit2009.pdf` (Download der Veröffentlichung)

[URL 2] GREX - Coordination and control of cooperating heterogeneous unmanned systems in uncertain environments
`www.tu-ilmenau.de/fakia/Grex.5235.0.html`

[URL 3] Firma Atlas Elektronik, Bremen – Projektpartner des GREX-Projektes
`www.atlas-elektronik.com`

[URL 4] AUV ABYSS (Leibnitz-Institut für Meereswissenschaften, Universität Kiel)
`www.ifm-geomar.de/index.php?id=auv`

[URL 5] San Diego iBotics / Stingray Project (studentisches Forschungsprojekt der Universität von San Diego, Californien/USA)
`www.sdibotics.org`

[URL 6] International Autonomous Underwater Vehicle Competition (San Diego / USA)
`www.auvsifoundation.org`

[URL 7] Autonomous Undersea Vehicle Applications Center - Zentrale Informationsstelle für alle AUV relevanten Themen weltweit

www.auvac.org

[URL 8] Liste mit AUV-Gruppen und -Projekten
www.transit-port.net/Lists/AUVs.Org.html

[URL 9] SONIA AUV - Projekt der Université du Québec (Kanada)
sonia.etsmtl.ca

[URL 10] UTD Autonomous Underwater Vehicle - Ein Projekt der Universität in Dallas (Texas, USA)
auv.utdallas.edu

[URL 11] Unreal Engine/Editor/SDK
www.unrealtechnology.com

[URL 12] NeoAxis Game Engine
www.neoaxisgroup.com

[URL 13] UNITY Game Development Tool
www.unity3d.com

[URL 14] Reality Factory - Software zur Entwicklung von Spielen
www.realityfactory.info

[URL 15] Crystal Space - quelloffene Entwicklungsumgebung für 3D-Anwendungen
www.realityfactory.info

[URL 16] Irrlicht Engine - freie Grafikbibliothek
irrlicht.sourceforge.net

[URL 17] SubSim - Simulationssystem für AUVs
robotics.ee.uwa.edu.au/auv/subsim.html

[URL 18] Remotely Operated Vehicle Simulator - Software zur Visualisierung von Unterwasserfahrzeugen inklusive Sonarsimulation
www.marinesimulation.com

[URL 19] Gazebo - Simulator zur Visualisierung von Robotern in 3D-Umgebungen; enthält Methoden zur Physik und Sonarsimulation; frei verwendbare Software
`playerstage.sourceforge.net/gazebo/gazebo.html`

[URL 20] AUV Framework - Simulationsumgebung für autonome Unterwasserfahrzeuge, inklusive Visualisierung und Steuerung
`www.iais.fraunhofer.de/4814.html`

[URL 21] VRML-Modell vom Fahrzeug *Supertanker*, Savage Datenbank
`https://savage.nps.edu/Savage/ShipsCivilian/Supertanker`

[URL 22] Visual C++ 2005 Redistributable Package
`www.microsoft.com/downloads/details.aspx?FamilyID=32BC1BEE-A3F9-4C13-9C99-220B62A191EE`
(Datei `vcredist_x86.exe`)

[URL 23] DirectX Redistributable 9.0c November 2007
`www.microsoft.com/downloads/details.aspx?FamilyID=1A2393C0-1B2F-428E-BD79-02DF977D17B8`
(Datei `directX.9.0c_nov2007_redist.exe`)

[URL 24] MeshMagick
`www.ogre3d.org/wiki/index.php/MeshMagick`
(Datei `meshmagick.exe`)

[URL 25] Visual C# Express Edition (kostenlos) oder Visual Studio
`www.microsoft.com/germany/express`
`www.microsoft.com/germany/visualstudio`

[URL 26] Visual Studio 2005 SP1
`msdn2.microsoft.com/en-us/vstudio/bb265237.aspx`
(Datei `VS80sp1-KB926606-X86-DEU.exe`)

[URL 27] AccuTrans 3D
`www.micromouse.ca`

[URL 28] FRAPS – Software zur Erstellung von Videos
`www.fraps.com`

[URL 29] Fraunhofer Anwendungszentrum für Systemtechnik in Ilmenau
`www.ast.iitb.fraunhofer.de`

[URL 30] OGRE - Grafikbibliothek
`www.ogre3d.org`

[URL 31] Mogre Tutorials - Beispiele für den Einstieg
`www.ogre3d.org/wiki/index.php/Mogre_Tutorials`

[URL 32] Spiele und weitere Software, die mit Ogre erstellt wurden
`www.ogre3d.org/wiki/index.php/Projects_using_OGRE`

[URL 33] Ogre Wiki
`www.ogre3d.org/wiki`

[URL 34] MOGRE Wiki (innerhalb des Ogre Wikis)
`www.ogre3d.org/wiki/index.php/MOGRE`

[URL 35] Ogre-Forum
`www.ogre3d.org/phpBB2`

[URL 36] Mogre-Forum (innerhalb des Ogre Addon-Forums)
`www.ogre3d.org/phpBB2addons/viewforum.php?f=8`

[URL 37] Ogre API-Referenz
(nützlich: Sortierung nach *Class List* und *Class Members*)
`www.ogre3d.org/docs/api/html`

[URL 38] Erweiterungsmodule für 3D-Software zur Speicherung von 3D-Modellen im Ogre-Format
`www.ogre3d.org/wiki/index.php/OGRE_Exporters`

[URL 39] VRML converter - Kommandozeilenprogramm zur Konvertierung von

VRML-Dateien ins Ogre-Format
`www.ogre3d.org/wiki/index.php/VRML_Converter`

[URL 40] Web3D.org - Konverter und Werkzeuge für VRML-Dateien
`www.web3d.org/x3d/vrml/tools/utilities_and_translators`

[URL 41] Einbindung von Objekten in Ogre-Anwendungen mittels Blender
`www.ogre3d.org/wiki/index.php/Blender_to_Ogre`

[URL 42] Website der 3D-Modellierungssoftware Blender
`http://www.blender.org`

[URL 43] Tutorial: Blender Anfänger Guide (PDF)
`www.ccsh.ch/fileadmin/downloads/blender/Tutorials/ ...`
`... blender_anfaenger_guide.pdf`

[URL 44] Blender e-Book
`de.wikibooks.org/wiki/Blender_3D`

[URL 45] Mit Blender erstelltes Spiel: Yo Frankie
`www.yofrankie.org`

[URL 46] Mit Blender erstellter Film: Elephants Dream
`www.elephantsdream.org`

[URL 47] Website der Skiptspache Python
`www.python.org`

[URL 48] Blender-Skript *Material Works* zur vereinfachten Materialverwaltung (Ersetzung, Umbenennung, etc.)
`http://blender-house.spaces.live.com/ ...`
`... blog/cns!C04B5FFE6DF805D6!165.entry`

[URL 49] myCSharp.de - größtes deutschsprachiges C#-Forum, Artikel, Quellcode
`www.mycsharp.de`

[URL 50] The Code Project - Artikel über Softwareentwicklung, Quellcode
www.codeproject.com

[URL 51] Kartenbezugssysteme, Geoide, Ellipsoide und die Form der Erde
www.kowoma.de/gps/geo/mapdatum.htm

[URL 52] U.S. Geological Survey - umfangreiche geologische Informationen über die Erde www.usgs.gov

[URL 53] A WGS84 to Swedish National Grid (RT90) Projection Class - The Code Project - C# Programming
www.codeproject.com/KB/cs/WGS84toSwedish.aspx

[URL 54] C# und C++ code to convert DD (LatLong) to UTM (World Wind Forums)
forum.worldwindcentral.com/showthread.php?t=9863

[URL 55] PROJ.4 - Cartographic Projections Library
trac.osgeo.org/proj

[URL 56] Details zum *Terrain Scene Manager* von Ogre
www.ogre3d.org/wiki/index.php/Terrain_Scene_Manager

[URL 57] Übersicht der Szenenmanager für Ogre
www.ogre3d.org/tikiwiki/SceneManagersFAQ

[URL 58] Large 3D Terrain Generator (L3DT)
www.bundysoft.com/L3DT

[URL 59] Wilbur (Editor für Höhenkarten)
www.ridgenet.net/ jslayton/software.html

[URL 60] Software zur Geländeerstellung (Übersicht)
www.ogre3d.org/wiki/index.php/DCC_Tools#Terrain_generation

[URL 61] Ogre Xml Converter
www.ogre3d.org/wiki/index.php/OgreXmlConverter

[URL 62] Ogre command-line tools
`www.ogre3d.org/download/tools`

[URL 63] Blender Exporter (alias Ogre Meshes Exporter); Infos, Doku, Download
`www.ogre3d.org/tikiwiki/tiki-index.php?page=Blender+Exporter`

`ogre.svn.sourceforge.net/viewvc/ogre/trunk/Tools/BlenderExport/ ...`
`... ogrehelp/ogremeshesexporter.html`

`http://ogre.svn.sourceforge.net/viewvc/ogre/branches/v1-6/ ...`
`... Tools/BlenderExport`

[URL 64] Sandcastle Help File Builder
`http://shfb.codeplex.com`

[URL 65] Sandcastle Help File Builder
`http://shfb.codeplex.com`

[URL 66] TortoiseSVN - Versionsverwaltung von Dateien, insbesondere für Quellcode
`www.tortoisesvn.net`

[URL 67] Visual C# 2008 (Andreas Kuehnel, Verlag Galileo)
Komplettes Buch downloadbar als PDF-Dokument
`openbook.galileocomputing.de/visual_csharp`

[URL 68] Online-Rechner für Quaternionen
`www.onlineconversion.com/quaternions.htm`

[URL 69] Rotation Calculator - Rechner für Quaterionen
`www.steve-m.com/downloads/tools/rotcalc`

A Genutzte Software

AccuTrans 3D	Konvertiert VRML Modelle in andere Formate [URL 27]
Blender	3D Modellierung [URL 42]
L3DT	*Large 3D Terrain Generator* zur Erstellung von Höhenkarten und zugehörigen Texturen [URL 58]
Ogre	Grafikbibliothek für 3D-Darstellung [URL 30]
OgreMeshExporter	Erweiterung für *Blender* zum von Modellen ins Ogre-Format [URL 63]
OgreXMLConverter	Konvertierung von XML-basierten Ogre Mesh Dateien (`*.mesh.xml`) in das binäre Ogre Mesh Dateiformat (`*.mesh`); Der OgreXMLConverter ist Bestandteil der *OgreCommandLineTools* [URL 61]
Python	Notwendig für die Benutzung des *OgreMeshExporters* innerhalb von *Blender* [URL 47]
Rotation Calculator	Umrechnung von Eulerwinkeln in Quaternionsparameter [URL 69]
Sandcastle Help File Builder	Erstellt eine Hilfe-Datei (`CViewVR.chm`) aus den XML-Kommentaren im Quellcode der Visualisierungssoftware zur Dokumentation von Klassen, Methoden, Parametern, Eigenschaften, Variablen, etc. [URL 65]
TortoiseSVN	Software für Versionsverwaltung von Dateien, insbesondere für Quellcode [URL 66]
Visual Studio	Entwicklungsumgebung für C#/.NET [URL 25]
Wilbur	Editor für Höhenkarten [URL 59]

B Blender - Korrektur und Export von Modellen

Dieser Abschnitt ist keine detaillierte Schritt-für-Schritt-Anleitung zur Bedienung des komplexen Programmes *Blender*. Er ist eher als Hilfestellung zu sehen, um die notwendigen Korrekturen aus Kapitel 3.3 (S. 26) durchzuführen. Grundkenntnisse im Umgang mit Blender werden vorausgesetzt. Weiterhin wird davon ausgegangen, daß das Modell bereits in einem Format vorliegt, das von Blender geöffnet werden kann (z.B. `*.3ds`).

Modell laden: Zum Laden eines Modells im Format `*.3ds` (3D Studio Max) darf nicht der Menübefehl *File* > *Open* verwendet werden. Statt dessen muß man *File* > *Import* > *3D Studio* aufrufen.

Räumliche Orientierung: Das Koordinatensystem des Fahrzeugmodells muß entsprechend Abbildung 3.2 (S. 28) ausgerichtet werden. Bezogen auf das Fahrzeug zeigt die Y-Achse nach oben und die X-Achse nach vorne. Durch Drücken der Taste **7** auf dem Ziffernblock der Tastatur, wird die X-Y-Ebene von oben angezeigt. Aus dieser Perspektive muß das Fahrzeug bei korrekter Ausrichtung aufrecht zu sehen sein, mit Blickrichtung nach rechts. Die Rotation des Fahrzeuges muß im *Edit Mode* erfolgen und auf das lokale Koordinatensystem bezogen sein. (Rotationen im *Object Mode* werden beim Speichern im Ogre-Format ignoriert.) Präzise Rotationen können über die Tastatur erfolgen: Erst **R** drücken, dann die Rotationsachse (z.B. **X**), dann die Gradzahl eintippen (z.B. 90) und anschließend Enter drücken.

Fahrzeugmitte: Ist der Ursprung des lokalen Koordinatensystems nicht in der Fahrzeugmitte, so muß das Modell entsprechend verschoben werden. Dies muß ebenfalls im *Edit Mode* erfolgen. Alternativ kann auch der 3D-Cursor in der Fahrzeugmitte positioniert und anschließend der Knopf *Centre Cursor* in der *Mesh*-Palette angeklickt werden.

Maßstab: Zur Anzeige der Maße diehnt das Fenster *Transform Properties*, welches im *Object Mode* über die Taste **N** aufgerufen wird. Die Werte *DimX*, *DimY*, *DimZ* zeigen die Modelllänge bezüglich der Achsen. Sie sind jedoch nur korrekt (bezüglich der Modelldaten, die im Ogre-Format gespeichert werden), wenn die Skalierung auf 1 steht. Skalierungen müssen im *Edit Mode* erfolgen. Zur Anzeige der Maße muß jedoch wieder in den *Object Mode* zurückgeschaltet werden. Hilfreich sind gegebenfalls die Funktionen *Clear Scale* und *Apply Scale/Rotation to ObData*, die im *Object Mode* aufrufbar sind über *Leertaste* > *Transform* > *Clear/Apply*.

Materialien: Eine Übersicht der verwendeten Materialdefinitionen ist in der Palette *Links and Pipeline* zu sehen, wenn man links unter *Link to Object* auf das Symbol mit dem Doppelpfeil und dann auf *DataBrowse* klickt. Die Übersicht zeigt alle Materialnamen und daneben die jeweilige Anzahl deren Verwendung. Wird die Palette *Links and Pipeline* nicht angezeigt, so muß man im *Object Mode* den *Shading*-Kontext aufrufen.

Die Anzahl der Materialien muß auf 15 reduziert werden. (Für zukünftige Versionen von Blender ist geplant, mehr Materialien pro Objekt zu ermöglichen.) Dies ist insbesondere dann der Fall, wenn redundante Materialdefinitionen vorhanden sind. Zur Entfernung werden im *Edit Mode* Modellbestandteile oder Teilflächen gleicher Farbe ausgewählt und diesen ein gemeinsames Material zugewiesen. Gegebenfalls ist neben der Farbe nach weiteren Materialeigenschaften zu differenzieren. Bei Modellen mit sehr vielen Teilobjekten ist es hilfreich, die Ebenenfunktion von Blender zu verwenden und alle Teilobjekte gleicher Farbe auf eine eigene Ebene zu verschieben (mit Taste **M**).

Ungenutzte Materialien sind in der Materialübersicht an der Anzahl 0 zu erkennen und können gelöscht werden. Weiterhin ist es vorteilhaft, aussagekräftige Materialnamen zu vergeben statt anonymen Bezeichnungen wie *material.001*. Wichtig ist, daß jeder Materialname einmalig ist und in keinem anderen Modell vorkommt, das in der Visualisierungssoftware eingebunden ist. (Falls es doch einen Konflikt gibt, weist eine Fehlermeldung in CViewVR darauf hin.) Ein Präfix bezüglich des Fahrzeugmodells oder -typ ist daher sinnvoll. Für das Fahrzeug Seewolf könnten beispielsweise Materialnamen wie *seewolf_grau* oder *seewolf_schraube* verwendet werden.

Hilfreich bei der Arbeit mit Materialien ist die Erweiterung *Material Works*. Sie kann Materialien umbenennen, ersetzen und ausgewählten Flächen zuweisen. Die

Bedienung ist recht intuitiv. Zum Aufrufen wird in einem Teilfenster von Blender der Fenstertyp *Scripts Window* eingestellt (siehe S. 32, Abbildung 3.5, Schritte 1 bis 4) und in dessen Menü *Scripts* > *Materials* > *Material Works* gewählt. Abbildung B.1 zeigt die Erweiterung.

Teilobjekte verschmelzen: Besteht das Modell aus separaten Teilobjekten, so müssen diese zu einem Gesamtobjekt zusammengefaßt werden. Dazu wählt man mit Taste **A** alle Teile im *Object Mode* aus und drückt die Tastenkombination **Strg** + **J**. Verändern durch das Zusammenfassen einige Flächen ihre Farbe, dann hatte das Modell mehr Materialdefinitionen als zulässig.

Ogre-Format: Zur Speicherung eines Modells im Ogre-Format wird der *Blender Exporter* (alias *Ogre Meshes Exporter*) benutzt. Der Aufruf des Exporters wurde bereits in Abbildung 3.5 (S. 32) gezeigt. Im Exporter müssen die Schaltflächen *Export Materials*, *Export Meshes*, *Coloured Ambient*, *Rendering Materials* und *OgreXMLConverter* aktiviert (dunkel) sein. Abbildung B.2 zeigt die richtigen Einstellungen.

Im Eingabefeld *Path* wird das Verzeichnis angegeben, wo die zu erstellende Mesh-Datei gespeichert wird. Als Zielpfad sollte nicht die Visualisierungssoftware gewählt werden, da sonst bereits vorhandene Modelle mit gleichem Namen ohne Warnung überschrieben werden. Der Dateiname kann nicht angegeben werden. Statt dessen wird der Name verwendet, den Blender intern für Objekte benutzt. (Dieser kann in der Palette *Link and Materials* geändert werden.) Im Eingabefeld *Material Settings* muß der Standardname *Scene.material* umbenannt werden. Sinnvoll ist es, den gleichen Namen zu verwenden wie bei der Mesh-Datei.

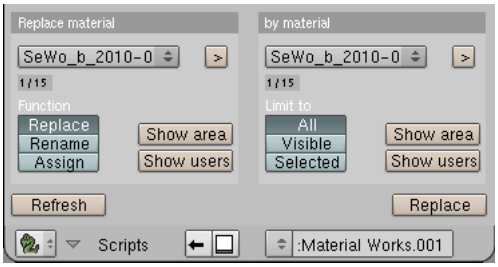


Abbildung B.1: Blender-Erweiterung *Material Works*

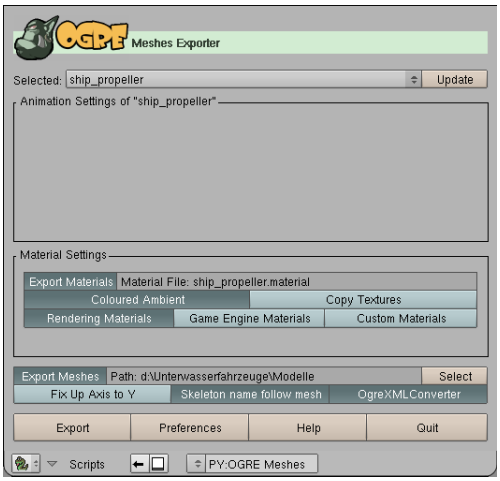


Abbildung B.2: *Blender Exporter* alias *Ogre Mesh Exporter*

C Matlab/Simulink - Einbindung des Clients

Für die Anbindung der Simulationssoftware Matlab/Simulink an die Visualisierungssoftware CViewVR wurde von Mike Eichhorn (Angestellter der TU Ilmenau im Fachgebiet Systemanalyse) eine Client-Software erstellt. Dabei handelt es sich um eine *s-function*, die als Simulink-Block (Abbildung C.1) eingebunden wird und Eingänge für Position (x, y, z), Bewegungsschätzung (u, v, w) und Rotation (yaw, pitch, roll) besitzt. Mithilfe dieses Blockes kann ein Fahrzeug in CViewVR gesteuert werden. Bei mehreren Fahrzeugen wird jeweils ein eigener Block eingefügt. Dabei ist zu beachten, daß in jedem Block der entsprechende Fahrzeugname (bzw. Objektname) eingetragen wird.

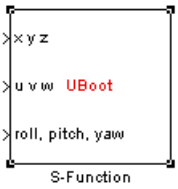


Abbildung C.1: Simulink-Block

Folgende Schritte sind notwendig, um den Client als Simulink-Block in ein Modell einzufügen:

- Die Datei `Client.dll`[8] muß in ein beliebiges Verzeichnis kopiert werden (z.B. in das Verzeichnis, wo auch das Simulink-Modell liegt). Anschließend wird dieser Pfad in Matlab eingestellt (siehe Abbildung C.2 auf Seite 92).
- Zum Einfügen in ein Simulink-Modell wird der Block namens *S-Function* genutzt. Dieser ist über den *Simulink Library Browser* (Abbildung C.3) zu finden. Nach dem Einfügen ist der Block noch ohne Funktion und besitzt die Beschriftung *system* (siehe Abbildung C.4).

[8] Der Client für Matlab/Simulink befindet sich auf der CD, die dieser Arbeit beiliegt. Sowohl die compilierte Bibliothek `Client.dll` als auch der Quellcode sind vorhanden.

- Über das Kontextmenü des eingefügten Blockes können dessen Parameter eingegeben werden. Abbildung C.5 (S. 93) zeigt den Aufruf des Konfigurationsdialoges und Abbildung C.6 die Eingabemaske. Im Feld *S-function name* wird `Client` angegeben, was dem Namen der DLL-Datei entspricht. Im darunter liegenden Feld werden die Parameter der S-function eingetragen.

- Die Parameter zur Konfiguration sind in Tabelle C.1 beschrieben und werden durch Leerzeichen separiert. Es ist darauf zu achten, daß einige Parameter einfache Anführungszeichen benötigen. Folgende Syntax wird für die Konfiguration des Simulink-Blockes verwendet:
  `<Intervall> '<Objektname>' '<IP-Adresse>' '<Port>' <Szenen-ID> <Flags>`
  Beispiel: `.01 'U-Boot' '127.0.0.1' '8000' 0 0`

- Abbildung C.7 (S. 93) zeigt das Aussehen des Blockes nach der Parametrierung. Um die Blockeingänge zu beschriften wird im Kontextmenü des Blockes der Eintrag *Edit Mask* aufgerufen (siehe Abbildung C.8 auf Seite 94). Im Feld *Drawing commands* der Eingabemaske werden die Eingänge beschriftet und optional ein individueller Blockname definiert. Die nötigen Eintragungen sind Abbildung C.9 zu entnehmen und Abbildung C.10 zeigt den fertig beschrifteten Simulink-Block.

- Da die Parameter für Bewegungsschätzung (u, v, w) derzeit nicht benötigt werden, kann der zweite Eingang des Blockes offen bleiben.

Name	Bemerkung
Intervall	Zeitlicher Abstand in Sekunden, mit dem die Zustandsdaten übertragen werden – dieser entspricht der reziproken Aktualisierungsfrequenz
Objektname	Name des zu steuernden Fahrzeugs – Maximal 20 Zeichen
IP-Adresse	Adresse des Servers von CViewVR
Port	Port des Servers von CViewVR – Standardnummer: 8000
Szenen-ID	Ungenutzter Parameter – 0 eintragen
Flags	Steuerflags, die durch einen Byte-Wert repräsentiert werden. Bisher wird nur das Latitude-Longitude-Flag verwendet. Bei lokaler Positionierung wird der Wert 0 verwendet und bei Lat-Long-Positionierung der Wert 128.

Tabelle C.1: Matlab/Simulink - Parameter für s-function

Eingänge des Blockes

Der Simulink-Block besitzt drei Eingänge, die jeweis drei multiplexte Parameter entgegennehmen.

- Beim ersten Eingang werden die Positionsangaben x, y und z entsprechend des NED-Koordinatensystems (siehe Kapitel 2.3 auf Seite 14) übergeben. Bei gesetzem Latitude-Longitude-Flag entspricht x dem Latitude- und y dem Longitude-Wert.

- Der zweite Eingang ist für Parameter der Bewegungsschätzung vorgesehen. Dieser Eingang kann offen bleiben, denn er wird bisher nicht verwendet.

- Der dritte Eingang nimmt Rotationsangaben entgegen, die in Form von Eulerwinkeln beschrieben sind. Zu beachten sind folgende Dinge:

  - Die Werte müssen das Bogenmaß verwenden. Gibt der Fahrzeugsimulator Grad-Werte aus, so muß ein Multiplikationsblock mit dem Wert $\pi/180$ vorgeschaltet werden.
  - Die multiplexten Rotationswerte müssen in folgender Reihenfolge übergeben werden: *roll*, *pitch*, *yaw*
  - Anders als die Reihenfolge der Rotationswerte für den Blockeingang suggeriert, gilt für die Rorationskonvention die Reihenfolge *yaw*, *pitch*, *roll*.
  - Angaben zu den Rotationsvorschriften und empfohlenen Wertebereichen sind in Kapitel 2.5 (S. 16) beschrieben.
  - Bei einigen Simulatoren müssen die Rotationswerte mit -1 multipliziert werden.
  - Im Simulatormodell muß die gleiche räumliche Ausrichtung verwendet werden, wie in CViewVR. Diese ist in der Abbildung 1.1 (S. 7) zu sehen.

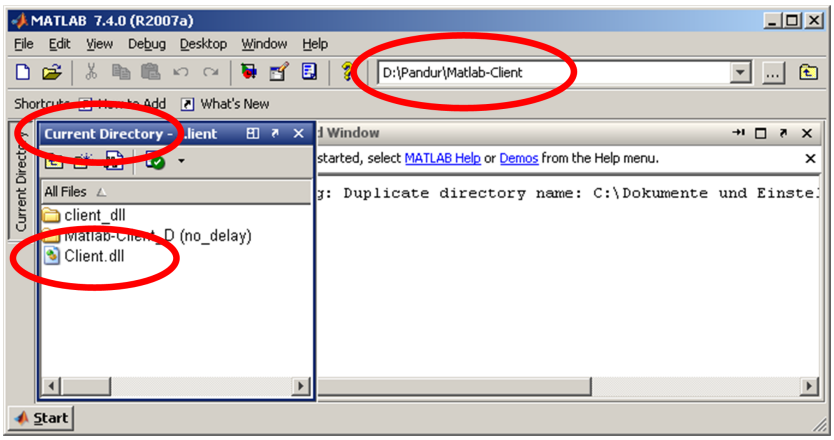


Abbildung C.2: Pfad mit DLL-Datei einstellen

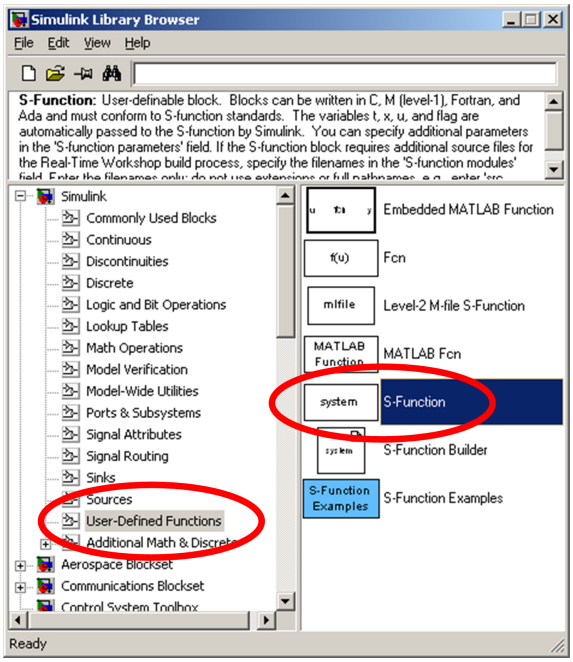


Abbildung C.3: S-Function Block einbinden

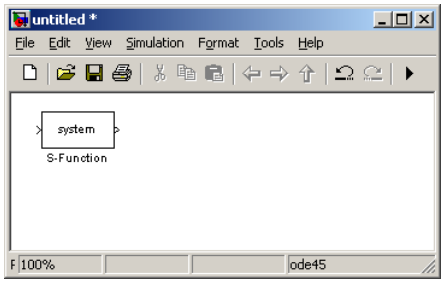


Abbildung C.4: S-Function Block (nicht parametrisiert)

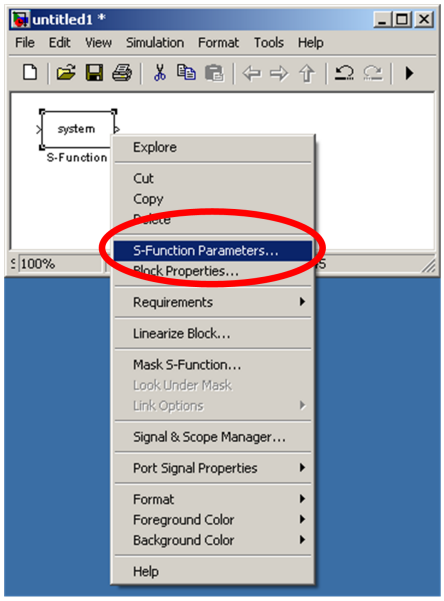


Abbildung C.5: Blockparameter einstellen

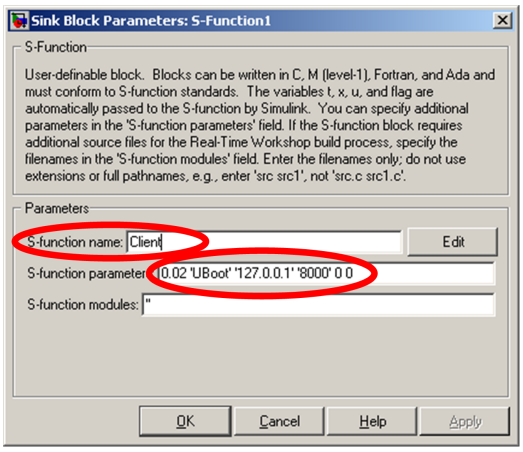


Abbildung C.6: Blockparameter festlegen

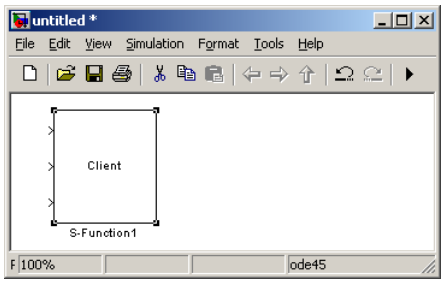


Abbildung C.7: S-Function Block (parametrisiert)

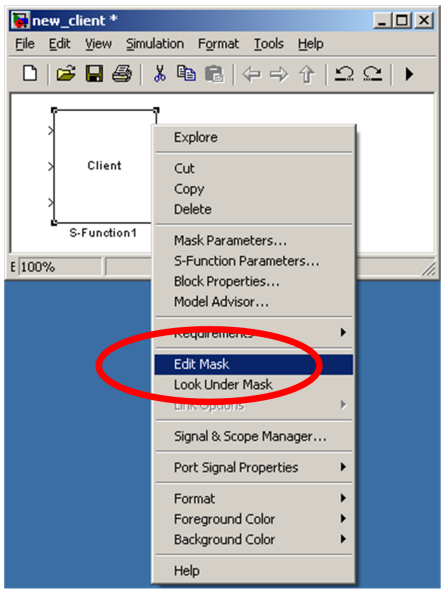


Abbildung C.8: Beschriftung über Kontektmenü einstellen

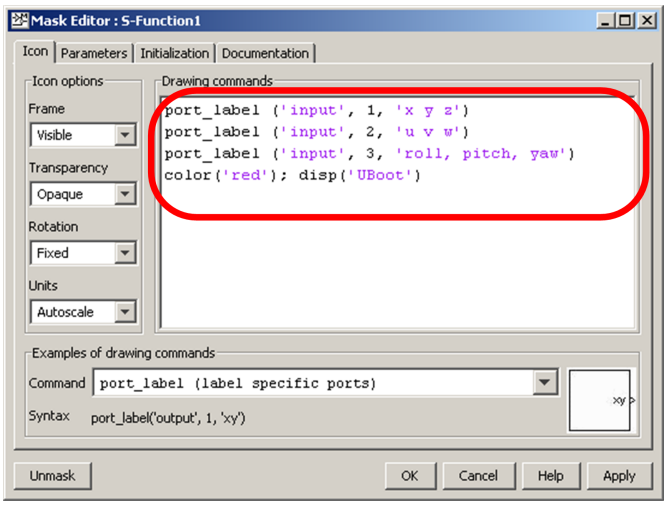


Abbildung C.9: Beschriftung festlegen

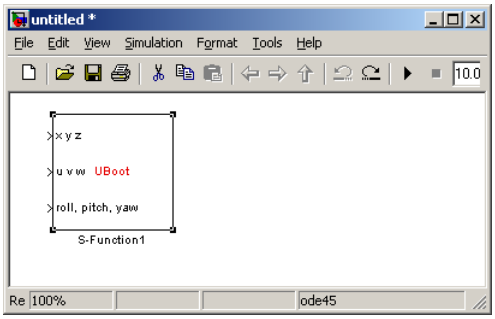


Abbildung C.10: S-Function Block mit fertiger Beschriftung

D Fehlerhandbuch

Im Laufe der Entwicklung und Verwendung von CViewVR sind wiederholt Probleme aufgetreten. Auf den folgenden Seiten sind typische Fälle aufgelistet und mit Hinweisen versehen, die bei der Problembehebung helfen können. Der Begriff *Objekt* bezieht sich in der Regel auf Unterwasserfahrzeuge. Er wurde dennoch allgemein gehalten, da sich die Fehlerbeschreibungen auch auf andere Bestandteile einer Szene beziehen können (z.B. Hafenmauer, Pipelines, Gebäude, Einzelteile von Fahrzeugen). Der Begriff *Client* bezeichnet ein Programm (z.B. Fahrzeugsimulator), das sich über die TCP-Schnittstelle mit CViewVR verbinden und Steuerdaten senden kann.

Allgemein ist es hilfreich, bei Problemen in den Logdateien `applicationLog.txt`, `applicationLog-onlyErrors.txt` und `ogreLog-onlyProblems.txt` nach Fehlermeldungen, Warnungen und sonstigen Hinweisen zu suchen. Die Logdateien befinden sich im Verzeichnis `CViewVR\CViewVR`. Sie können alternativ über das Menü von CViewVR geöffnet werden.

CViewVR lässt sich direkt nach Installation nicht starten

→ ist *dot.NET 2.0* installiert?
→ ist das *Visual C++ 2005 Redistributable Package* [URL 22] installiert?
→ ist *DirectX Redistributable 9.0c November 2007* [URL 23] (oder neuer) installiert?

Beim Programmstart wurde Datei `d3dx9_31.dll` nicht gefunden

→ *DirectX* benötigt neuere Bibliotheken. Die Version *Redistributable 9.0c November 2007* [URL 23] (oder neuer) muß installiert werden.

Ein Objekt ist in der riesigen Welt nur schwer oder garnicht auffindbar

→ Aktivierung von GUI-Option *coord sys* blendet für alle Objekte ein gut sichtbares Koordinatensystem ein.
→ Falls das Objekt eine eigene Kamera hat, kann diese zur Suche benutzt werden.

Ein angesteuertes Objekt bewegt sich nicht

→ Prüfen, ob Objekt-Daten empfangen wurden (siehe Datei `streamLog.txt`). Wenn ja, ist der gesendete Objektname wirklich der gleiche wie in der geladenen Szenen-Datei?

Ein angesteuertes Objekt verschwindet

Problem: Ein Objekt ist beim Laden der Szene vorhanden und nach Verbindungsaufbau bzw. beim Versand der ersten Steuerdaten „verschwindet“ es.
→ Das Objekt wird an eine andere Stelle verschoben und ist dadurch nicht mehr zu sehen. Üblicherweise muß im Client-Programm ein Positions-Offset addiert werden, damit das Objekt an der gewünschten Stelle der Szene agiert. Zum Auffinden des Objektes kann die Option *coord sys* aktiviert werden.
Beachte: Die Szenen-Datei benutzt das Ogre-Koordinatensystem. Die dortigen XYZ-Werte können nicht als Vorlage für die NED-Position im Client verwendet werden.

Ein Objekt liegt auf der Seite oder dreht sich falsch

- Fehlermöglichkeit 1: Das Modell in der Ogre Mesh-Datei hat eine falsche räumliche Ausrichtung (bezogen auf das lokale Koordinatensystem).

→ einfacher Test: GUI-Option *coord sys* aktivieren, um das lokale Koordinatensystem anzuzeigen. Bei korrekter Ausrichtung eines Objektes sind die Achsen wie folgt ausgerichtet: rot = vorne, grün = rechts, blau = unten.
Wenn das nicht der Fall ist, dann muß die Ogre Mesh-Datei (`*.mesh`) neu erstellt werden. (siehe Kapitel 3.4 auf Seite 30) Alternativ können Mesh-Dateien mit der Option *transform* des Kommandozeilenprogramms *MeshMagick* [URL 24] modifiziert werden. (siehe Kapitel 3.6 auf Seite 35)

- Fehlermöglichkeit 2: Das Client-Programm sendet Daten im Bogenmaß

→ einfacher Test: Wert des Winkels in kleinen Schritten ändern und beobachten, ob sich das Objekt auffällig langsam oder schnell dreht.
Lösung: Im Client Korrekturfaktoren für die Winkel einfügen ($\pi/180$ oder $180/\pi$).
Hinweis: Der Matlab-Client erwartet Winkeleingaben im Bogenmaß.

- Fehlermöglichkeit 3: In der Client-Anwendung sind Winkel vertauscht oder das Vorzeichen stimmt nicht.

→ einfacher Test: Nur einen Winkel ändern und beobachten, wie sich das Objekt dreht.

- Fehlermöglichkeit 4: Die Euler-Winkel entsprechen nicht den Vorgaben.
Problem: Für die richtige Beschreibung der Orientierung mittels Eulerwinkel sind folgende Dinge wichtig: Der Bezug auf das NED-Koordinatensystem und die Reihenfolge der Rotationen. (siehe Kapitel 2.3 auf Seite 14)
Merkmal: Wenn man das Objekt nur um eine Achse dreht, ist die Drehung richtig. (Einzeln mit allen drei Achsen testen!) Kombiniert man mehrere Drehungen, dann ist das Ergebnis teilweise oder immer falsch.

→ Es gibt keine einfache Lösung. Gut wäre, wenn der Simulator die Eulerwinkel entsprechend der Vorgaben von CViewVR ausgeben kann. Eine weitere Möglichkeit wäre eine Umrechnung der Eulerwinkel in der Client-Software. Eine allgemeine Formel scheint es dafür jedoch nicht zu geben. Alternativ könnte man den Quellcode von CViewVR anpassen, was das Programm dann aber inkompatibel für andere Client-Anwendungen macht. Der relevante Code steht in Klasse `Scene.objectUpdate()`.

Ein Client kann keine Verbindung herstellen

→ Sichergehen, daß wirklich keine Verbindung aufgebaut wurde.
(Suche in Datei `applicationLog.txt` den Eintrag *„Client gefunden“*.)

→ Wenn der Client auf dem selben Computer läuft, muß die IP-Adresse in der GUI auf 127.0.0.1 stehen. Wenn der Client auf einem anderen Computer läuft, muß eine externe IP-Adresse des Computers ausgewählt werden. (Häufig ist die Adresse im Bereich 192.168.x.x.)

→ Prüfen der Einstellungen von IP-Adresse und Port in Client und CViewVR (Datei `ipconfig.txt`).

→ Firewall-Einstellungen prüfen: Wird die Verbindung server-/clientseitig blockiert? Eventuell auf beiden Rechnern die Firewall testweise abschalten.

→ Den Client erst starten, wenn die gewünschte Szene in CViewVR bereits geladen und die Visualisierung gestartet wurde.

Wenn nichts hilft: Die Testprogramme `Server`, `Client` und `WinClient` können zur Untersuchung der TCP-Verbindung genutzt werden. (siehe Kapitel 4.5 auf Seite 55)

Das Bild ruckelt

• Fehlermöglichkeit 1: Die Bildrate ist zu klein. (siehe fps-Anzeige in GUI)

→ Fenster verkleinern oder PC mit schnellerer Grafikkarte verwenden

→ Bei Mehrbildschirmbetrieb: Auf dem Hauptbildschirm ist die Anzeige bei einigen Computern erheblich schneller.

• Fehlermöglichkeit 2: Die Client-Anwendung (z.B. Matlab/Simulink) beansprucht zu viele Ressourcen.

→ CViewVR und Client-Anwendung auf verschiedenen PCs laufen lassen.

Das Bild ist flüssig, aber die Objekte ruckeln bei Bewegungen

- Fehlermöglichkeit 1: Beim TCP-Protokoll werden die Daten nicht immer sofort gesendet, sondern bei kleinen Datenmengen zwischengepuffert und in periodischen Intervallen gesendet (z.B. alle 0,2 Sekunden).

→ Einfacher Test: Prüfe Zeitstempel in Datei `streamLog.txt` nach periodischen Auffälligkeiten.

Lösung: Im TCP-Client muß die Socket-Option *no_delay* aktiviert sein.

- Fehlermöglichkeit 2: Der Client sendet die Zustandsdaten in zu großen zeitlichen Abständen.

→ Ein guter Wert ist alle 0,01 Sekunden (100 Hz). Ab ca. 0,04 Sekunden (25 Hz) wird es ruckelig. Kleinere Abstände als 0,01 Sekunden sind nicht sinnvoll.

Ein Objekt flimmert beim Abspielen der Simulation

Das passiert, wenn ein Objekt zeitgleich von verschiedenen Clients gesteuert wird. Dann wird das Objekt ständig zwischen den verschiedenen Soll-Positionen hin und her verschoben. Dadurch entsteht das flimmernde Erscheinungsbild.

→ In diesem Fall sollte die Konfiguration aller Client-Anbindungen auf Korrektheit der Objektnamen geprüft werden.

Einige Teile von Objekten haben eine falsche Farbe

→ *Material*definition überprüfen: Ist die entsprechende Materialdatei vorhanden? Enthält sie alle Material*namen*, die das Objekt verwendet? Stimmen in den jeweiligen Materialdefinitionen die *Farb*werte? Sollten die Farben in der Vergangenheit gestimmt haben, später aber nicht mehr, dann wurde vermutlich eine Materialdatei durch eine Version mit anderem Inhalt überschrieben. (siehe Kapitel 3.5 auf Seite 33)

Ein Objekt ist hellgrau oder weiss statt farbig

Allgemein: Die Datei `ogreLog-onlyProblems.txt` kann hilfreiche Infos enthalten.

• Fehlermöglichkeit 1: Materialdatei(en) wurden nicht gefunden.
→ Untersuchen, welche Materialnamen das Modell in der Ogre Mesh-Datei verwendet. Das kann mit dem Kommandozeilenprogramm *MeshMagick* [URL 24] geprüft werden. Der Befehl lautet: `meshmagick.exe info <objektName>.mesh`
→ Für jeden Materialnamen muß eine entsprechende Materialdefinition in den Materialdateien enthalten sein.

• Fehlermöglichkeit 2: Texturdatei(en) wurden nicht gefunden.
→ Materialdateien können Verweise auf Bilddateien enthalten, um Texturen darzustellen. Diese Dateien müssen auch vorhanden sein, was gegenfalls überprüft werden sollte.

• Fehlermöglichkeit 3: Das 3D-Modellierungsprogramm *Blender* hat manchmal Probleme bei der Materialverwaltung.
→ Blender-Projekt speichern; Blender neu starten; Objekt erneut exportieren

• Fehlermöglichkeit 4: Haben die Farben in der Vergangenheit existiert und später sind sie verschwunden?
Das kann daran liegen, daß eine `material`-Datei durch eine andere Version ersetzt wurde. Möglicherweise sind dann nicht mehr alle nötigen Materialdefinitionen enthalten.
→ Materialdefinition überprüfen (siehe Fehlermöglichkeit 1).

• Fehlermöglichkeit 5: Das Objekt wurde in der Szenendatei skaliert.
Bei Objekten mit Skalierung kleiner als 1 kommt es vor, daß die Oberfläche nicht in der richtigen Farbe, sondern teilweise oder vollständig weiß dargestellt wird. Bei Skalierungen größer als 1 wurde dieses Problem nicht beobachtet.
→ In Szenendatei prüfen, ob bei den `<scale>`-Definitionen Werte ungleich 1 vorhanden sind. Wenn ja, dann testweise die Werte auf 1 setzen und prüfen, ob die Farben danach richtig dargestellt werden.
Hinweis: Wenn man die Größe eines Objektes dauerhaft ändern will, sollte man sie nicht in der Szenendatei skalieren, sondern die Größe im 3D-Modellierungsprogramm anpassen und die binäre Modelldatei (`*.mesh`) neu erstellen.

Teile eines Objekt wirken komisch

Problem: Die Oberflächen-Normalen des Modells zeigen nach innen statt nach außen.
Typische Merkmale:

- Objektoberflächen sind nicht von außen, sondern nur von der Innenseite sichtbar.
- Innere Teile eines Objektes sind zu sehen, obwohl sie im Objektinneren verborgen sein mußten (z.B. der Motor eines Fahrzeuges).

→ Lösung: Das Objekt im 3D-Modellierungsprogramm korrigieren und erneut exportieren.
Korrektur in *Blender*: *Edit Mode > Mesh > Normals > Recalculate Outside*

E Szenendatei - XML und XSD

Hier ist der Aufbau einer einfachen Szenendatei zu sehen. Sie enthält das Fahrzeug *Seewolf*, eine an den Seewolf gekoppelte Kamera und Angaben zur frei beweglichen Kamera. Der Szenengraph dieser Szene ist in Abbildung 2.2 (S. 13) dargestellt.

E.1 Beispielszene (XML)

```
<?xml version="1.0" encoding="utf-8"?>
<IEScene xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <saveDate>2008-07-01</saveDate>
  <terrainConfigFile>terrain1.cfg</terrainConfigFile>
  <waterlevel>560</waterlevel>
  <worldPosition_isDefined>false</worldPosition_isDefined>
  <worldPosition_latitude_degree>0</worldPosition_latitude_degree>
  <worldPosition_longitude_degree>0</worldPosition_longitude_degree>
  <ObjList>

    <anyType xsi:type="IEObject">
      <nodeName>Seewolf</nodeName>              <!-- Seewolf -->
      <meshName>Seewolf.mesh</meshName>
      <parentNodeName>SceneRoot</parentNodeName>
      <position>
        <x>3710</x>
        <y>560</y>
        <z>2845</z>
      </position>
      <orientation>
        <w>-0.258819073</w>
        <x>0</x>
        <y>-0.965925932</y>
        <z>0</z>
      </orientation>
      <castShadows>true</castShadows>
      <camNode>false</camNode>
      <sonarNode>false</sonarNode>
      <scale>
        <x>1</x>
        <y>1</y>
        <z>1</z>
      </scale>
    </anyType>
```

```
    <anyType xsi:type="IEObject">
      <nodeName>camSeewolf</nodeName>              <!-- camSeewolf -->
      <parentNodeName>Seewolf</parentNodeName>
      <position>
        <x>-10</x>
        <y>3</y>
        <z>0</z>
      </position>
      <orientation>
        <w>0.7017053</w>
        <x>0.049237188</x>
        <y>0.709060967</y>
        <z>-0.04917264</z>
      </orientation>
      <castShadows>false</castShadows>
      <camNode>true</camNode>
      <sonarNode>false</sonarNode>
      <scale>
        <x>1</x>
        <y>1</y>
        <z>1</z>
      </scale>
    </anyType>

    <anyType xsi:type="IEObject">
      <nodeName>camFree</nodeName>               <!-- camFree -->
      <parentNodeName>SceneRoot</parentNodeName>
      <position>
        <x>3653.21021</x>
        <y>566.8215</y>
        <z>2842.25513</z>
      </position>
      <orientation>
        <w>-0.716082931</w>
        <x>-0.03518269</x>
        <y>-0.6963159</y>
        <z>0.03364316</z>
      </orientation>
      <castShadows>false</castShadows>
      <camNode>true</camNode>
      <sonarNode>false</sonarNode>
      <scale>
        <x>1</x>
        <y>1</y>
        <z>1</z>
      </scale>
    </anyType>

  </ObjList>
</IEScene>
```

E.2 Schema-Definition (XSD)

Die folgende XSD-Definition befindet sich bei CViewVR in der Datei `scenefile.xsd` im Verzeichnis `Scenes`.
Weitere Informationen zur Schema-Definition stehen im Kapitel 4.3 (S. 47).

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="IEScene">
 <xs:complexType>
  <xs:all>    <!-- xs:all = Elemente 1x oder 0x vorhanden; Reihenfolge NICHT egal -->
     <xs:element name="saveDate"          type="xs:string"  minOccurs="0" />
     <xs:element name="terrainConfigFile" type="xs:string" />
     <xs:element name="sonarConfigFile"   type="xs:string"  minOccurs="0" />
     <xs:element name="waterlevel"        type="xs:float" />
     <xs:element name="worldPosition_isDefined"          type="xs:boolean"  minOccurs="0" />
     <xs:element name="worldPosition_latitude_degree"    type="xs:double"   minOccurs="0" />
     <xs:element name="worldPosition_longitude_degree"   type="xs:double"   minOccurs="0" />
     <xs:element name="waterOptions"    minOccurs="0" >
       <xs:complexType>
         <xs:sequence>
           <xs:element name="disableWater"         type="xs:boolean" />
           <xs:element name="applyResizePosition"  type="xs:boolean" />
           <xs:element name="sizeX"                type="xs:float"   />
           <xs:element name="sizeZ"                type="xs:float"   />
           <xs:element name="positionX"            type="xs:float"   />
           <xs:element name="positionZ"            type="xs:float"   />
           <xs:element name="alternativeMaterial"  type="xs:string"  />
         </xs:sequence>
       </xs:complexType>
     </xs:element>
     <xs:element name="camSpeed"        type="xs:float"   minOccurs="0" />

     <xs:element name="ObjList">  <!-- Liste -->
      <xs:complexType>
       <xs:sequence>
          <xs:element name="anyType" type="IEObject" maxOccurs="unbounded" />  <!-- Elemente der Liste -->
       </xs:sequence>
      </xs:complexType>
     </xs:element>  <!-- END Liste -->

     <xs:element name="previewPicture"    type="xs:string" minOccurs="0" />
  </xs:all>
 </xs:complexType>
</xs:element> <!-- END IEScene -->


<!-- complex types -->

<xs:complexType name="IEObject">  <!--  Elemente der Liste -->
  <xs:all>
     <xs:element name="nodeName"       type="nodeNameType" />
```

```
      <xs:element name="parentNodeName" type="nodeNameType" />
      <xs:element name="meshName"       type="meshFileType"  minOccurs="0" />
      <xs:element name="position"       type="positionType" />
      <xs:element name="scale"          type="scaleType" />
      <xs:element name="orientation"    type="orientationType" />
      <xs:element name="camNode"        type="xs:boolean"  minOccurs="0" />
      <xs:element name="sonarNode"      type="xs:boolean"  minOccurs="0" />
      <xs:element name="castShadows"    type="xs:boolean"  minOccurs="0" />
  </xs:all>
</xs:complexType>

<xs:complexType name="positionType">
  <xs:all>
    <xs:element name="x" minOccurs="1" type="xs:float" />
    <xs:element name="y" minOccurs="1" type="xs:float" />
    <xs:element name="z" minOccurs="1" type="xs:float" />
  </xs:all>
</xs:complexType>

<xs:complexType name="scaleType">
  <xs:all>
    <xs:element name="x" minOccurs="1" type="xs:float" />
    <xs:element name="y" minOccurs="1" type="xs:float" />
    <xs:element name="z" minOccurs="1" type="xs:float" />
  </xs:all>
</xs:complexType>

<xs:complexType name="orientationType">
  <xs:all>
    <xs:element name="w" minOccurs="1" type="xs:float" />
    <xs:element name="x" minOccurs="1" type="xs:float" />
    <xs:element name="y" minOccurs="1" type="xs:float" />
    <xs:element name="z" minOccurs="1" type="xs:float" />
  </xs:all>
</xs:complexType>


<!-- restrictions -->

<xs:simpleType name="nodeNameType">
  <xs:restriction base="xs:string">
       <!-- alternativ "xs:token" (erlaubt / entfernt Whitespaces am Anfang+Ende vom String (osä.) -->
    <xs:pattern value="[a-zA-Z0-9_.\-/]+"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="meshFileType">
  <xs:restriction base="xs:string">
    <xs:pattern value="[a-zA-Z0-9_.\-]+\.mesh"/>
  </xs:restriction>
</xs:simpleType>

</xs:schema>
```

Danksagung

Besonderen Dank gilt folgenden Personen, die mich bei der Ausarbeitung dieser Arbeit unterstützt haben:

- Dr.-Ing. Torsten Pfützenreuter (Fraunhofer Anwendungszentrum Systemtechnik) für seine Hilfe bei diversen Problemen, nützlichen Hinweisen und Korrekturvorschlägen dieser Arbeit.
- Dr.-Ing. Thomas Glotzbach (TU Ilmenau, Fachgebiet Systemanalyse) für die Unterstützung, vor allem bei der Ausarbeitung des Schriftteils der Arbeit.
- Dipl.-Ing. Fabian Müller (Fraunhofer Anwendungszentrum für Systemtechnik) für die gemeinsamen Praxistests der Visualisierungssoftware.
- Dr.Ing. Mike Eichhorn (ehemals TU Ilmenau, Fachgebiet Systemanalyse) für die Erstellung der Client-Anbindung für Matlab/Simulink.
- Peter Rudolph (studentischer Mitarbeiter im Fraunhofer AST) für die Erstellung eines C++-Clients zur Anbindung von Fahrzeugsimulatoren, die in C++ geschrieben wurden.
- Dipl.-Ing. Achim Gehr (Fraunhofer Anwendungszentrum für Systemtechnik) für seine Korrekturvorschläge der Schreibarbeit.
- Meine Großeltern und Freundin Marlen für deren Motivation und ausdauernde Geduld, bis zum Abschluß dieser Arbeit.
- Dem Fraunhofer AST für die Zurverfügungstellung des Arbeitsplatzes und die sehr angenehme Arbeitsatmosphäre.

Erklärung

Die vorliegende Arbeit habe ich selbstständig ohne Benutzung anderer als der angegebenen Quellen angefertigt. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Quellen entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer oder anderer Prüfungen noch nicht vorgelegt worden.

Ilmenau, den 18. Juli 2011

Andreas Voigt