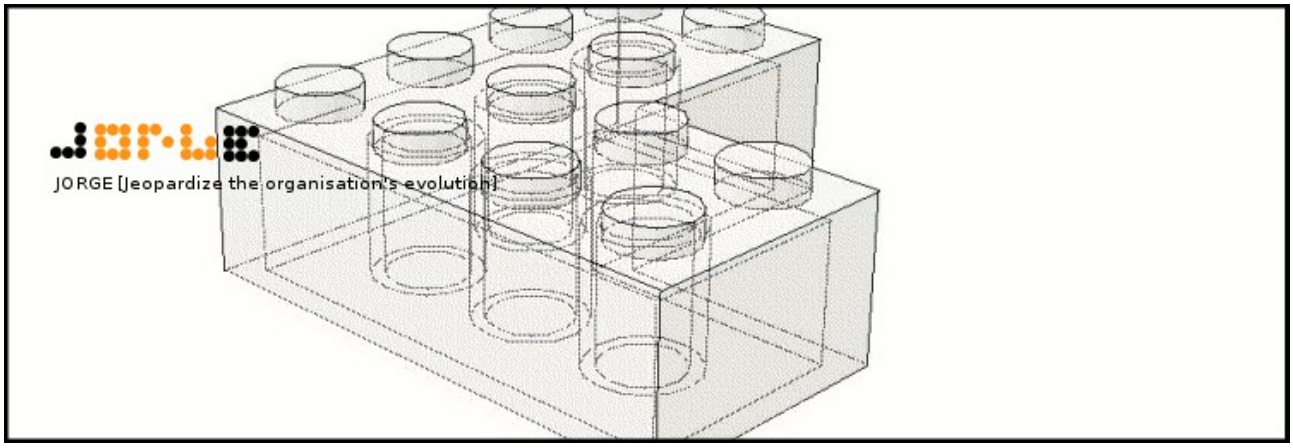
Berner Fachhochschule
Hochschule für Technik und Informatik HTI



Technischer Bericht

[Jorge – Lego MindStorms]

Autoren	Nik Lutz [I3STW, lutzn@hti.bfh.ch]
	Stefan Feissli [I3STW, feiss@hti.bfh.ch]
	Christof Seiler [I3STW, seilc@hti.bfh.ch]
Version	1.0
Datum	24.06.05
Status	Freigegeben
Webseite	jorge.hta-bi.bfh.ch
Projektauftraggeber	Claude Fuhrer
PM-Coaching	Frank Helbling
Projektarbeit	Sommersemester 2005, HTI Biel, Abt. Informatik

Inhaltsverzeichnis

1 Einleitung

1.1 Zweck des Dokuments

Dieses Dokument zeigt die Arbeiten sowie deren Aufwand, die während unserer Semesterarbeit auftraten. Die Form wurde weitmöglichst chronologisch gehalten, beginnend bei der Evaluation bis hin zur Implementation.

2 Zeitkontrolle

Phase	**Task**	**Stunden / Person**
Evaluation	IDE	40
	Emulator	70
	Rendering System	10
	Modeling	40
	Terrain	10
	GUI Frameworks	30
Architektur	Klassenstruktur (Klassendiagramm)	40
	Anwendungsfälle	20
Implementation	Libraries	25
	Boost	10
	Emulator	100
	Scene Graph	5
	XML Konfiguration der Roboters	20
	Physik	50
	Rückwärtsgang	10
	Fühlerkollision	30
Libraries & Tools	Ogre	100
	Ode	40
	OgreOde Wrapper	80
	Boost	10
	Cygwin	10
	LeJos	40
	Blender	50
	Terragen & Freeworld3D	20
Total		**860**

Die Arbeitsstunden waren mit total 320 Stunden / Person gerechnet. Dazu kommen noch die Stunden für die Semesterarbeit im Virtual Reality Lab. Ausserdem müssen noch die Stunden für die Projektmanagement Dokumente und die technischen Dokumente dazugerechnet werden.

3 Evaluation

3.1 Allgemein

3.1.1 Ide

Arbeitsaufwand: 40 Stunden

Den Aufwand eine IDE zu finden wurde von uns komplett falsch eingeschätzt. Wir wollten zu Beginn mit Eclipse arbeiten, mussten dann aber relativ schnell herausfinden, dass nützliche Features wie CodeCompletion und CodeBrowsing mit grösseren Frameworks wie Ogre einfach zu langsam waren.

Auch mussten wir feststellen, dass Eclipse die einzige IDE ist, die auf mehreren Plattformen läuft. So haben wir uns entschlossen, auf den jeweiligen Betriebssystemen unterschiedliche IDE's einzusetzen. Auf Windows war das Problem recht schnell gelöst, die Lösung hiess Visual Studio. Auch auf Mac fiel die Entscheidung nicht schwer: Xcode.

Mehr Sorgen machte uns Linux. KDevelop und Anjuta, mit Integration von den Autotools, eignen sich zwar für kleinere Projekte, haben jedoch Einschränkungen in Bezug auf Verzeichnis Struktur. Auch stürzte KDevelop häufig ab. Das grösste Manko aber war die CodeCompletion, welche nur auf Klassen innerhalb des Projekts funktionierte. Zwar besteht die Möglichkeit externe Header Files anzugeben, diese Funktion war jedoch, ähnlich zu Eclipse, einfach zu langsam. Somit haben wir uns entschlossen ein Tool zu kaufen. Aber auch hier hatten wir mit Debian keine grosse Auswahl. Unsere Wahl fiel auf CodeForge.

3.2 Emulator

Arbeitsaufwand: 70 Stunden

Damit ein Benutzer das Verhalten seiner Programme (mit JORGE) simulieren kann, musste ein Weg gefunden werden, um die vom Benutzer geschriebenen Programme auszuführen oder zu interpretieren.

3.2.1 Evaluation Laufzeitumgebungen

Wir haben zahlreiche Projekte auf dem Internet gefunden, die sich mit der Ausführung von RCX-Programmen ausserhalb des RCX (d.h. auf dem PC) befassen. Deshalb haben wir zuerst abgeklärt, ob sich eines dieser Projekte für den Einsatz in JORGE eignen würde. Der ausführliche Bericht findet man im Dokument “Emulator Emulation“, in diesem Dokument präsentieren wir nur das Resultat dieser Auswertung.

3.2.1.1 Kriterien

- LeJos: Es muss möglich sein, Programme die auf dem LeJos-API aufbauen auszuführen oder zu interpretieren.
- Einfache Integration: Wird bestehender Code verwendet, sollte dieser in C oder C++ geschrieben worden sein .
- Platform: Die gewählte Lösung sollte im Minimum unter Windows, wenn möglich aber auch unter Linux und MacOsX compilierbar und ausführbar sein.

3.2.1.2 Auswertung

Hier wird Zusammenfassend beschrieben, welche Projekte wir getestet haben, und welche Erkentnisse wir daraus gezogen haben.

Das Resultat der Evaluation ist ernüchternd, von den 7 getesteten Projekten, kommen nur zwei in die engere Auswahl: BrickEmu und Emulejos-run.
IntelLego, jLegoEmu und RCXSimulator können nicht eingesetzt werden, da sie in Java geschrieben sind. Mit EmuLegOs können keine LeJos Programme ausgeführt werden und RcxEmu scheitern am Crossplatform Anspruch.
Die “Eigene Implementation“ beschreibt nur eine Lösungs-Idee, da wir kein Prototyp dazu erstellt haben, wissen wir nicht ob dieser Ansatz überhaubt funktioniert. Da wir aber zwei andere funktionierende Kandidaten haben, wollen

wir nichts riskieren (bezüglich Zeitaufwand und Funktionalität) und verwerfen die eigene Implementation.

Somit verbleiben BrickEmu und Emulejos-run. BrickEmu ist der interessantere Ansatz, da es sich um einen vollständigen Emulator handelt, der verschiedenste Lego-Betriebssystem laden und ausführen kann. Emulejos-run ist eine abgespeckte Virtuelle Maschine die nur LeJos-Programme ausführen kann. Gegen BrickEmu spricht, dass dort 3 verschiedene Programmiersprachen(C, Perl und Tcl/TK fürs GUI) eingesetzt werden, was unter Windows Probleme verursachen könnte. Ausserdem benötigt man zum Ausführen von BrickEmu ein ROM-Image, womit wir sehr grosse Schwierigkeiten hatten, und es bezüglich der Lizenz rechtliche Probleme gibt. Für Emulejos-run spricht, dass er für LeJos Programme optimiert wurde und auch in der LeJos Distribution enthalten ist.
Im Rahmen der Semesterarbeit ist emulejos-run die bessere Lösung, bezüglich Aufwand und Risiko. Da beide, unter Windows, Cygwin voraussetzen, können wir mit emulejos-run erste Erfahrungen in Zusammenspiel mit Cygwin und Visual Studio sammeln, ohne das wir uns mit Perl oder Tcl/Tk herumschlagen müssen. Mit diesen Erfahrungen können wir bei der Planung der Semesterarbeit nochmals über diese Entscheidung diskutieren.

3.2.1.3 Weiteres Vorgehen

Unter Linux und verwandten Systemen werden wir keine Probleme beim Kompilieren haben. Anders sieht es mit Windows aus. Es muss deshalb zuerst abgeklärt werden, ob es möglich ist, emulejos-run mit Visual Studio zu kompilieren. Falls dies möglich ist, können wir uns sofort um den Datenaustausch (Sensor- und Motordaten) zwischen emulejos-run und der virtuellen Welt (Ogre) kümmern. Falls wir emulejos-run nicht mit Visual Studio kompilieren können, müssen wir entweder emulejos-run portieren oder einen Weg finden, emulejos-run unter Cygwin zu kompilieren, und dann irgendwie in Visual Studio einzubinden. Falls wir keinen Weg finden JORGE (Ogre Teil) und emulejos-run in ein Program zu verschmelzen (linken), verbleibt die Möglichkeit den Datenaustausch über Pipes oder über eine Netzwerk-Schnittstelle zu realisieren.

3.3 Virtual Reality

3.3.1 Rendering System

Arbeitsaufwand: 10 Stunden

3.3.1.1 Vergleich

Am Anfang unserer Evaluation des Vitual Reality Teils stellte sich uns die Frage welches Rendering System wir benutzen. Für uns kamen: Java3D, Ogre3D und OpenGL in die engere Auswahl und mussten sich in einem offenen Schlagabtausch gegeneinander bewähren. In der Tabelle 1 sehen sie einen direkten Vergleich dieser Systeme.

Kriterium	Java3D	Ogre3D	OpenGL
OOP	****	****	*
Bestehende API	****	****	*
Bekanntheitsgrad	***	**	****
Community	***	****	****
Physikanbindungen	****	****	***
Lernfaktor	***	****	***
Effizienz (FPS)	**	****	****
Effektivität der Arbeit	****	***	*
Total Sterne	**26**	**29**	**21**

Tabelle 1: Vergleich Rendering Systeme

Die Bewertung:

*	schlecht
**	ungenügend
***	gut
****	hervorragend

3.3.1.2 Fazit

Wie aus der Tabelle ersichtlich haben wir uns für Ogre3D entschiedenen. Uns war vor allem wichtig ein effizientes Programm zu erstellen und Know-How in einer Umgebung zu entwickeln, die dann später auch realistisch ist für ein kommerzielles Produkt. Ausserdem hat uns nach den Jahren der Java Entwicklung auch noch einen Einblick in andere Programmiersprache gereitzt. Natürlich haben wir bereits C++ Erfahrung gesammelt im Verlaufe unseres Studiums, jedoch sind diese Erfahrungen meist theoretischer Natur. Es ist uns aber auch wichtig in einem praktischen Lernprozess Erkenntnisse zu erlangen, die wir mit dieser Arbeit zweifelsohne erfahren haben.

3.3.2 Modeling

Arbeitsaufwand: 40 Stunden

Wir wollte ein angenehmes Modeling Tool weil es sehr wichtig ist auch im Hinblick auf die Diplomarbeit, dass wir möglichst effizient Objekte herstellen und vor allem modifizieren können. Zu diesem Zweck haben wir allle gängigen Modling Tools evaluiert. Zu diesen gehören:

- Maya (Nur auf Windows und Mac)
- MilkShape (Nur Basisfunktionalität)
- 3ds Max (Keine Linzenz, nur Windows)
- Blender

Wir haben uns für Blender entschieden, für nähere Angaben siehe Punkt Tools -> Blender.

3.3.3 Terrain

Arbeitsaufwand: 10 Stunden

Um das Terrain zu erstellen haben wir ebenfalls Tools evaluiert. Damit das Terrain von Ogre interpretiert werden konnte, waren folgende Punkte wichtig:

- Heightmap entweder als Bitmap- oder als Raw-Format

Unsere Bedürfnisse an das Tool waren:

- Intuitive Erstellung der Hügel
- Einfache Kreation der Texture
- Realistisches Aussehen

Die folgenden Tools haben wir in Betracht gezogen:

- Freeworld3D, kommerziell (http://freeworld3d.org/)
- gile[s], kommerziell (http://www.frecle.net/giles/)
- Terragen, freie Version (http://www.planetside.co.uk/terragen/)
  verschiedene Converter

Unsere Wahl fiel hier auf die kommerzielle Lösung Freeworld3D kombiniert mit Terragen.

3.4 GUI Frameworks

Arbeitsaufwand: 30 Stunden

Um dem Benutzer eine gewohnte Oberfläche mit Menus, Buttons, usw. Anzubieten, haben wir verschiedene Crossplatform GUI Framworks angeschaut und sie mit Ogre getestet.

3.4.1 WxWindows

WxWindows (http://www.wxwindows.org) konnte die Ogre-Szene nur für nicht bewegte Bilder benutzen. Für bewegte Szenen müsste man die Ogre-Szene in einem eigenen Thread manuell updaten, was sicher keine gute Lösung ist. Unser Beispiel (samples/wxWidgets/) baut auf einer Vorlage aus dem Ogre-Wiki auf, wo das erwähnte Problem heftig diskutiert wird. Wir hoffen bei der Diplomarbeit dort eine funktionierende Lösung zu finden.

3.4.2 CGui

CGui (http://www.idt.mdh.se/~csg/cgui/) ist eine sehr kleines GUI-Framework, das zum Beispiel keine vorgefertigten Datei-Öffnen Dialoge enthält. Wir haben aber eine Vorlage gefunden, die mit Hilfe von Ogre-Overlays unter Windows einen Datei-Öffnen Dialog realisiert. Wir haben einen halben Tag investiert, und versucht dieses Beispiel für Linux anzupassen. Es hat sich gezeigt, dass wir einen weiteren Tag hätten investieren müssen. Da uns die Lösung mit Overlays nicht

überzeugte (kein dem Benutzer bekanntes Look-And-Feel), haben wir diesen Ansatz verworfen.

3.4.3 FLTK

Im Ogre-Wiki haben wir ein weiteres Beispiel gefunden, das FLTK (http://www.fltk.org/) als GUI Framwork zusammen mit Ogre benutzt. Leider haben wir es nicht geschafft das dort beschriebene Beispiel zum Laufen zu bringen. Während unserer Diplomarbeit, werden wir uns mit dem Autor von diesem Beispiel in Verbindung setzen um dieses Problem zu lösen.GTK
Mit GTK (http://www.gtk.org) hatte wir die genau gleichen Probleme wie mit WxWindows.

4 Architektur

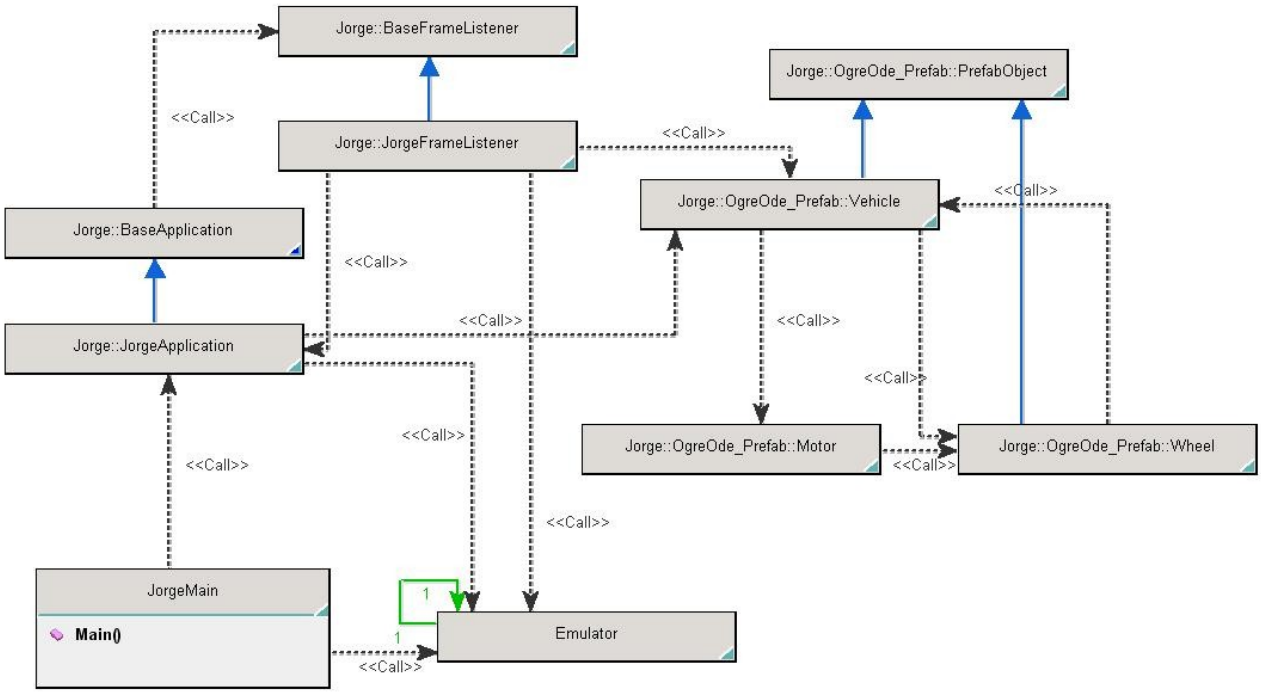
4.1 Klassendiagramme

Arbeitsaufwand: 40 Stunden

Das Klassendiagramm zeigt die Abhängigkeiten der verschiedenen C++ Komponenten. Die Klasse Emulator ist das Bindeglied zur Laufzeitumgebung. Sie wird beim Start von JorgeMain initialisiert (Singleton-Pattern) und testet, ob das gewünschte Program ausgeführt werden kann. Wenn der Benutzer mit einem Tastendruck sein Program startet (JorgeFrameListener behandelt dieses Ereignis), wird in der Klasse Emulator ein neuer Thread gestartet und das Program in der Laufzeitumgebung ausgeführt. Danach dient die Klasse Emulator als Kommunikations-Schnittstelle: Sie verwaltet die Zustände der verschiedenen Sensoren und Motore. JorgeFrameListener fragt vor jedem neu gezeichneten Frame (frameStarted-Methode) den Zustand der Motoren ab und überträgt sie auf die Vehicle-Klasse. Im Fall einer Kollision nutzt JorgeApplication die Emulator-Klasse, um den neuen Zustand des betroffenen Sensors zu setzten.
Die Laufzeitumgebung nutzt die Emulator-Klasse, um Motoren zu starten und Sensor-Daten auszulesen. Da sie in einem eigenen Thread läuft, hat die Klasse Emulator ausserdem die Aufgabe die Datenstrukturen vor gleichzeitigem Zugriff zu schützen (Mutex).

Die Emulator-Klasse dient dazu die Abhängigkeiten zwischen den Komponenten der virtuellen Welt und der Laufzeitumgebung so klein wie möglich zu halten. Wenn

man eine andere Laufzeitumgebung einzusetzen will, muss man nur die Emulator-Klasse verändern und sich nicht um die anderen Komponenten kümmern.



Wir haben die Applikation in vier Namespaces aufgeteilt:

- Jorge
- Jorge::OgreOde
- Jorge::OgreOde_Prefab
- Jorge::Emulators

Nachfolgend liefern wir einen kurzen Überblick zum Code. Für nähere Informationen verweisen wir auf die Doxygen API Dokumentation, welche unter jorge/jorgeMain/doc gefunden werden kann.

4.1.1 Namespace Jorge

Der Namespace Jorge enthält:

- Die Mainklasse JorgeMain
- 2 Base-Klassen, welche die Grundfunktionen für Ogre Applikationen zur Verfügung stellen

- JorgeApplication und JorgeFrameListener

Die Base-Klassen sind generell implementiert und können auch für andere 3D Applikationen genutzt werden. Hier sind vor allem Grundfunktionen wie das Steuern der Kamera mit den Pfeiltasten.

JorgeApplication dient hauptsächlich zum Erstellen der Szene sowie zur Initialisierung der Welt, die von Ode genutzt wird. Ebenfalls findet man hier die „collision“-Methode, welche das Verhalten überschreibt bei einer Kollision.

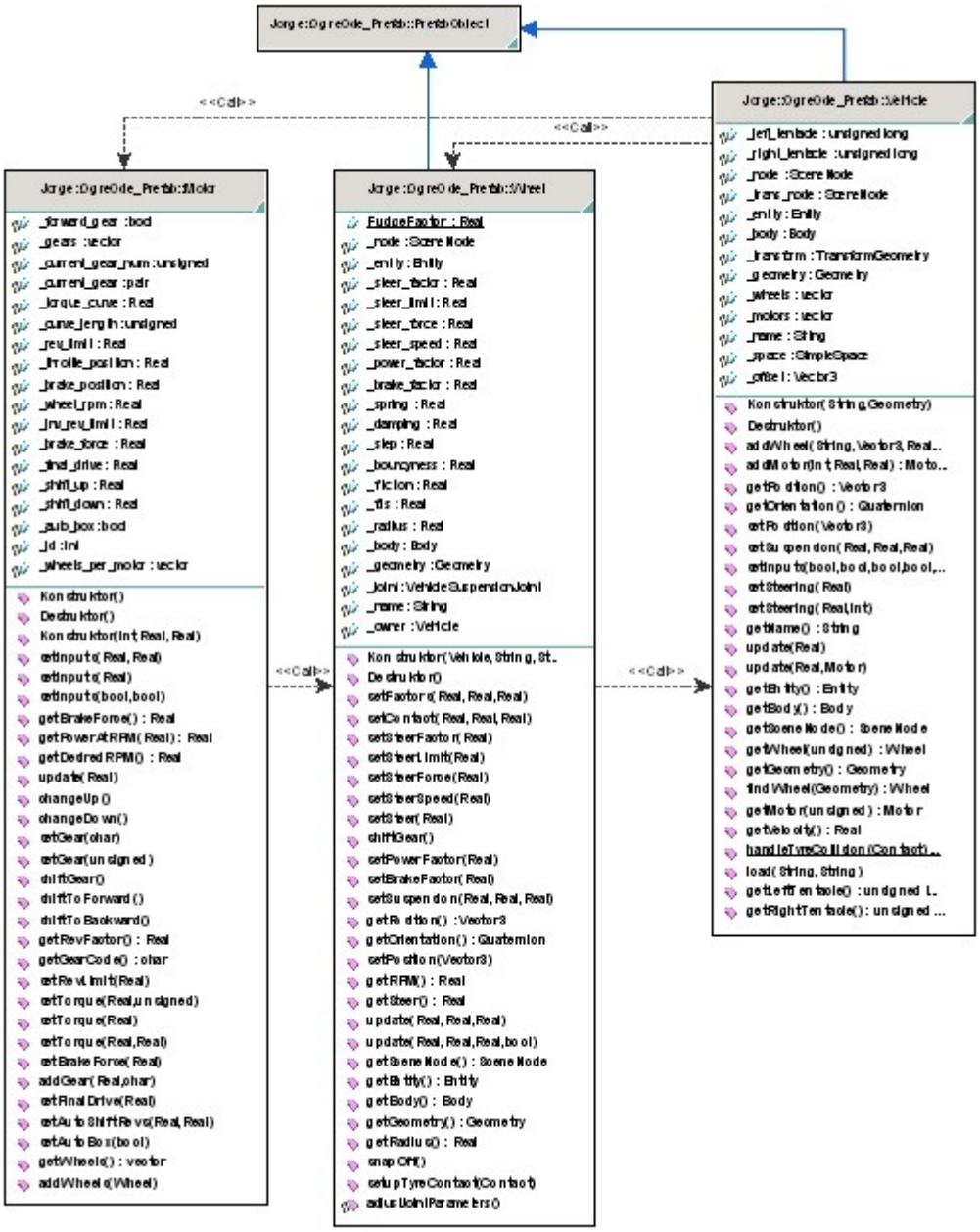
JorgeFrameListener enthält die Methode frameStarted und frameEnded, die bei jedem Neuzeichnen des Bildes aufgerufen werden. Hier werden auch KeyEvents abgefangen.

4.1.2 Namespace Jorge::OgreOde_Prefab

Dieser Namespace enthält die Klassen Vehicle, Motor und Wheel. Hier wird unser virtueller Roboter abgebildet. Physikalische Einstellungen aus der XML Datei werden in diesen Datenstrukturen festgehalten (siehe Punkt “XML Konfiguration des Roboters”).

4.1.3 Namespace Jorge::OgreOde

Jorge::OgreOde stellt die Verbindung von Ogre zu Ode dar. Dies repräsentiert den eigentlichen Wrapper. Hier werden die entsprächenden C-Funktionen vom Ode Interface aufgerufen. Da hier praktisch keine Änderungen vorgenommen wurden, haben wir darauf verzichtet, den Code zu dokumentieren.

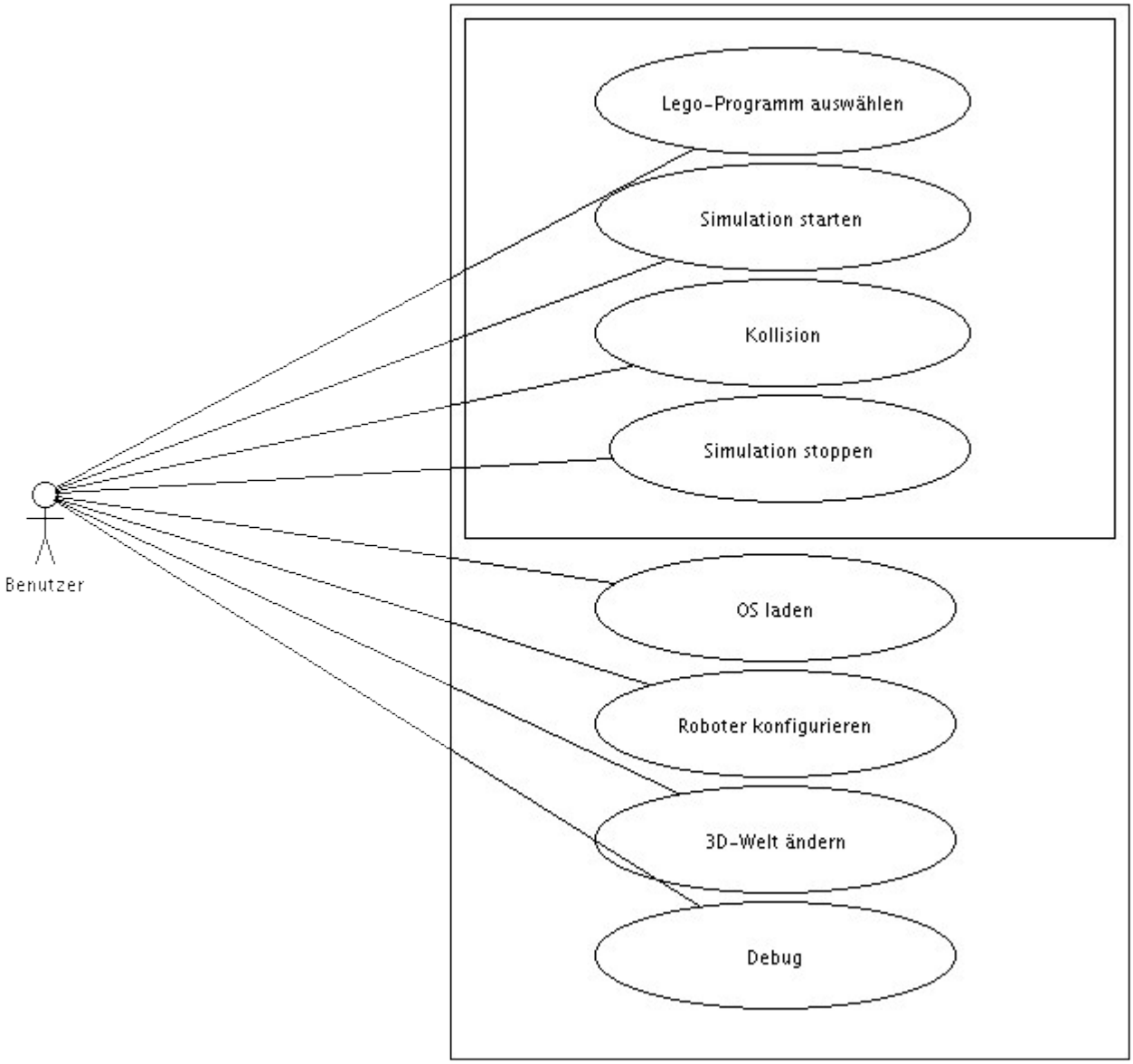


4.1.3.1 Namespace Jorge::Emulators

Jorge::Emulators beschreibt die Schnittstelle zum Emulator. Innerhalb dieses Namespaces werden die Sensor- und Motorzustände verwaltet und der Emulator in einem neuen Thread gestartet.

4.2 Anwendungsfälle (Use Cases)

Arbeitsaufwand: 20 Stunden



Im Folgenden kommentieren wir die von uns definierten Anwendungsfälle zu Beginn der Semesterarbeit.

5 Lego-Programm auswählen

Der Benutzer startet das Program, und wählt auf dem Dateisystem ein von ihm erstelltes LeJOS-Program. Wenn der Import erfolgreich war, kann der Benutzer die Simulation starten.

Kommentar: Da kein getestetes GUI-Framwork zufriedenstellend mit Ogre arbeitet, bieten wir dem Benutzer kein Datei-Öffnen Dialog an, sondern verlangen den Vollständigen Pfad und Dateiname als Argument beim Start. Wir sind zuversichtlich, im Rahmen der Projektarbeit eine geeignete Lösung zu finden.

6 Simulation starten

Nachdem der Benutzer das Programm gestartet und ein LeJOS-Programm ausgewählt hat, kann er das Programm anweisen die Simulation zu starten. Der Benutzer kann nun die Aktionen des Roboters in der 3D-Welt/Landschaft mitverfolgen.

Kommentar: Mit der Taste “O“ wird JORGE angewiesen das LeJos Program zu starten.

7 Kollision

Falls der Roboter an ein Hindernis stösst, wird ein Sensor-Ereignis ausgelöst und an das vom Benutzer geladene Lego-Programm als Sensor-Aktivität weitergeleitet.

Kommentar: Bei einer Kollision wird dem Benutzer innerhalb des JORGE Fensters angezeigt, dass ein Sensor ausgelöst wurde. Falls das Lego-Program darauf mit einer Anweisung an den Roboter reagiert, kann der Benutzer im “Emulator Log“, ein Textfeld innerhalb des JORGE Fensters, diese Anweisungen in Textform lesen und mitverfolgen wie der Roboter reagiert.

8 Simulation stoppen

Der Benutzer kann zu jedem Zeitpunkt die Simulation stoppen, wenn gewünscht ein neues Lego-Programm auswählen und die Simulation neu starten.

Kommentar: Mit der ESC-Taste kann der Benutzer jederzeit JORGE beenden. Um ein neues Program zu wählen muss JORGE neu gestartet werden. (Siehe auch “Simulation starten“)

9 Nice to Have: OS laden

Der Benutzer wählt im Dateisystem ein Lego-Betriebssystem. Das Betriebbssystem wird im Emulator gestartet. Wenn das Starten des Lego-Betriebssystems erfolgreich war, kann der Benutzer für dieses Betriebbssystem geeignete Lego-Programme laden.

Kommentar: Diesen Anwendungsfall haben wir als “Nice to have“ deklariert und konnte nicht realisiert werden. Da wir anstelle eines Emulators eine Virtuelle Maschine(für Java) einsetzen.

10 Nice to Have: Roboter konfigurieren

Der Benutzer kann den gewünschten Roboter laden und gewisse Einstellungen wie die Beschleunigung des Roboters oder die Anschlüsse der Sensoren konfigurieren.

Kommentar: Dieser optionale Anwendungsfall konnten wir nur zur Hälfte Realisieren. In einer XML-Datei kann das Drehmoment der Motoren, die Anzahl angetriebener Räder, die Beschaffenheit der Oberflächen (und somit die Haftung/Reibung der Räder am Boden) und vieles mehr definiert werden. Die Anschlüsse der Sensoren und Motoren (Verkabelung) ist zur Zeit noch nicht konfigurierbar.

11 Nice to Have: 3D-Welt ändern

Der Benutzer wählt im Dateisystem eine Datei, die eine geeignete Beschreibung einer 3D-Welt/Landschaft enthält.

Kommentar: Dieser optionale Anwendungsfall konnten wir nicht realisieren.

12 Nice to Have: Debug

Der Benutzer kann jederzeit über ein Dialogfenster Zustandsinformationen anzeigen lassen (z.B. Register-Inhalte), um so bespielsweise Fehler im Lego-Programm finden zu können.

Kommentar: Diesen Anwendungsfall haben wir als “Nice to have“ deklariert und konnte nicht realisiert werden.

13 Implementation

13.1 Allgemein

13.1.1 CVS Struktur

Der ganze SourceCode, Dokumente, Bilder sowie Tools sind auf dem CVS Server verfügbar. Die Zugangsdaten lauten: cvs.hta-bi.bfh.ch -> /var/cvsreps/projects/jorge.

Hier ein paar Worte zur Struktur:

- doc enthält alle von uns erstellten Dokumente, speziell erwähnt seien hier die Projektführungsdokumente sowie das Pflichtenheft und der Technische Report
- images enthält Dateien wie unser Logo. Diese Daten werden nicht von der Applikation selber genutzt.
- jorgeMain ist unsere eigentliche Applikation.
  - Hier befindet sich der ganze Source Code.
  - Wir haben die Include und CPP Dateien getrennt um eine bessere Übersicht zu schaffen.
  - Im bin Verzeichnis findet man Konfigurationsdateien sowie vorkompilierte Libraries für Windows.
  - Scripts enthält Projekt Dateien für Visual Studio sowie Xcode. Auf die CodeForge Projektdatei haben wir verzichtet. Da diese Datei absolute Pfadnamen enthält.
- Media enthält alle unsere Ressourcen, welche wir für die Applikation benötigen. Auch sind hier die Blender Dateien vorhanden.
- Im Samples Ordner kann man verschiedene Beispiele finden. Da in diesen Ordner relativ viel getestet wurde, kann hier nicht garantiert werden, dass diese Samples laufen. Dieser Ordner dient mehr zu Testzwecken für das Team.
- Tools enthält alle unsere Libraries, die für den Einsatz von Jorge benötigen wie Ogre, Ode etc.

13.1.2 Libraries

Arbeitsaufwand: 25 Stunden

Um überhaupt mit der Entwicklung beginnen zu können, mussten wir erstmals die verschiedenen Libraries installieren und zum Laufen bringen. Wie so oft stellte dies unter Windows keine grösseren Probleme dar. Unter Linux hatten wir mehr Probleme.

Um die ganzen Libraries unter Linux zu installieren und alle Abhängigkeiten zu lösen hatten wir ungefähr 3 Tage. Zeit kosteten uns:

- Kennenlernen von Tools wie Automake
- Patches, welche manchmal mühsam zusammen gesucht werden mussten

- Versionsprobleme zwischen einzelnen Libraries, z.B. zwischen Wrapper und der neusten Version von Ogre

13.1.3 Main: Einstiegspunkt der Applikation

Die Main Funktion findet man unter *jorge/jorgeMain/src/JorgeMain.cpp.* Dort wird überprüft, ob das Übergebene Argument auf eine gültiges LeJos Program zeigt. Wenn die Datei ein gültiges Program enthält wird eine Instanz von JorgeApplication kreiert, und die Applikation gestartet.

13.1.4 Boost

Arbeitsaufwand: 10 Stunden

Um Boost zu testen haben wir ein Beispiel Projekt angelegt, und es auf Windows und Linux getestet. Boost wurde fast komplet in Header-Dateien realisiert, so ist es für viele Anwendungen nicht nötig, Boost zu kompilieren, sonder es reicht aus die Header-Dateien einzubinden. Dieses Verfahren haben wir für Boost-Mutex nutzen können. Um Boost-Threads einsetzen zu können, muss man aber vorab Boost kompilieren.
Der Source Code von Boost ist entpackt ca 23 MB gross, und es dauert ungefähr eine halbe Stunde, um es zu kompilieren. Zum Kompilieren setzt Boost nicht auf Platform-Spezifische Tools sondern nutz das Platform-Unabhängige jam (http://www.perforce.com/jam/jam.html). Auf Linux-Systemen bieten die meisten Distributoren Boost als Binär-Packet an. Unter Windows ist die Installation/Kompilation aber aufwändig. Deshalb haben wir Boost für Windows kompiliert und die benötigten Dateien (Binär und Sourcen) zum CVS hinzugefügt (jorge/tools/boost/), so dass man Boost ohne weiteres einsetzen kann (wenn man unseren ganzen CVS-Baum heruntergeladen hat).
Im Verzeichnis *samples/sampleEmulator/* hat es zwei Beispiele wie man Boost einsetzen kann.

13.2 Emulator

Arbeitsaufwand: 100 Stunden

Grundsätzlich wollten wir den bestehenden Source-Code möglichst wenig verändern, damit allfällige Änderungen (vom LeJos-Projekt) einfach übernommen werden könnten. Deshalb haben wir auch die Struktur so beibehalten wie sie von LeJos

vorgegeben wurde, obwohl damit einige Schwierigkeiten im Zusammenhang mit automake enstanden sind. Dennoch haben wir alle Ausgabe-Statements (printf) durch einen Aufruf auf die in der Emulator Klasse definierten Log-Funktion ersetzt, damit wir die Ausgabe im Ogre-Fenster anzeigen können und nicht in der Konsole.

13.2.1 Datenaustausch

Wie im Abschnitt Architektur erwähnt, erfolgt der Datenaustausch (Motoren und Sensoren Zustände und Debug-Ausgabe vom Emulator) über die C++ Klasse Emulator, welche die in C geschriebene Laufzeitumgebung in einem neuen Thread startet. Da mehrere Threads auf die Sensor- und Motordaten zugreifen müssen, konnte der Laufzeitumgebung nicht einfache ein Pointer auf die entsprechende Datenstruktur übergeben werden.
Wir haben je zwei Funktionen in der Emulator Klasse definiert, die für's lesen und schreiben der Motoren bzw. Sensordaten zuständig sind:

```
virtual int emulator_set_sensor_state(int sensor,int value);
virtual int emulator_get_sensor_state(int sensor);

virtual int emulator_set_motor_state(int sensor, int value);
virtual int emulator_get_motor_state(int sensor);
```

Diese Funktionen haben je einen gemainsamen Boost-Mutex, der den gleichzeitigen Zugriff verhindert. Nun musste ein Weg gefunden werden, um vom C-Code aus diese Funktionen (C++ Objekt!) aufzurufen. Dies haben wir mit Funktionspointern realisiert: Man übergibt dem C-Code einen Pointer auf das C++ Objekt und für jede benötigte Funktion einen zusätzlichen Pointer.

Ausserdem benötigt man statische C++ Wrapper-Funktionen, welche aus dem C-Code aufgerufen werden können, und den Funktions-Aufruf auf das C++ Objekt machen, ein Beispiel:

```
static int
Wrapper_To_Call_emulator_get_sensor_state(void* pt2Object, int sensor){
    Emulator* mySelf = (Emulator*) pt2Object;
    return  mySelf->emulator_get_sensor_state(sensor);
}
```

Die Laufzeitumgebung braucht also ein Pointer auf das Emulator-Objekt und einen weiteren Pointer auf jede benötigte Funktion. Hier ein Auszug der C-Funktion der Laufzeitumgebung die diese Pointer entgegennimmt:

```
int init_emulator(char *file,int verbose,
                void* pt2Object,
                int (*pt2sensorFunction)(void* pt2Object, int sensor),
                             int (*pt2motorFunction)(void* pt2Object, int motor,
int value),
                void (*pt2logFunction)(void* pt2Object, int msg_type, char *entry)
                ){
    pt2EmulatorObject = pt2Object;
    pt2Emulator_get_sensor_state_function = pt2sensorFunction;
    pt2Emulator_set_motor_state_function = pt2motorFunction;
    pt2Emulator_log = pt2logFunction;
...
```

Im C-Code kann dann die C++ Funktion wie folgt aufgerufen werden:

```
pt2Emulator_set_motor_state_function(pt2EmulatorObject, motor_nr, value);
```

13.2.2 Compilieren: Linux

Unter Linux war das Kompilieren kein Problem, die mitgelieferten Makefiles funktionierten ohne Probleme mit unseren Änderungen.

13.2.3 Compilieren: Windows

Der grösste Nachteil der gewählten Lösung ist, dass unter Windows Cygwin zum Compilieren benötigt wird. Dies deshalb, weil gewisse Funktionen wie z.B. *gettimeofday* oder *signal* in der Windows-API nicht existieren.
Natürlich haben wir versucht den Code in Visual-Studio zu importieren, was aber zu unzähligen Fehlern führte, so dass eine Portierung sehr wahrscheindlich den zeitlichen Rahmen gesprengt hätte. Deshalb haben wir einen Weg gesucht, den Code mit Cygwin zu kompilieren und erst danach mit Visual Studio einzubinden.

13.2.3.1 Ansatz 1: Erzeugen einer DLL mit Cygwin

Mit Cygwin kann man DLL (Dynamic Linked Libraries) erzeugen, die dann von einem Program zur Laufzeit eingebunden werden können.

Grundsätzlich kann relativ einfach eine Dll mit Cygwin erzeugt werden, was wir hier anhand eines simplen Beispiels zeigen, haben wir auch mit unserem Code gemacht:

1. Schreibe den Dll Code:
   ```
   #include <stdio.h>
   int hello(){
       printf(„Hello World!\n");
   }
   ```
2. Kompiliere an der Cygwin Kommandozeile:
   gcc -c mydll.c
3. Generiere daraus eine Shared Library:
   gcc -shared -o mydll.dll mydll.o
4. Schreibe ein Program das diesen Code laden kann:
   ```
   ...
   HMODULE h = LoadLibrary("mydll.dll");
   void (*hello)() = GetProcAddress(h, "hello");
   hello();
   ..
   ```
5. Compiliere dieses Program mit Cygwin und führe es aus.

Dieser Ansatz funktioniert, wenn man das Program, das die DLL einbindet, auch mit Cygwin kompiliert. Wenn man diese DLL aber in ein mit Visual Studio erstellte Applikation einbinden will, braucht man zusätzlich eine .lib-Datei (Library), welche die Funktions-Definitionen der in der Dll enthaltenen Funktionen enthält. Hierfür sind folgende Schritte nötig:

1. Download impdef.zip von http://ftp.berlios.de/pub/unimatrix-fulda/win-devel/impdef.zip
2. entpacke impdef.zip
3. Führe folgenden Befehl in einer Konsole aus:
   impdef.exe mydll.dll > mydll.def
4. Nutze den Visual Studio Linker zum generieren der .lib datei:
   ```
   lib.exe /def:mydll.def /out:mydll.lib
   ```
   -> lib.exe findet man gewöhnlich unter: c:\Programs\Microsoft Visual Studio .Net 2003\Vc7\bin\lib.exe
5. In Visual Studio kann man unter den Linker-Optionen mydll.lib angeben.

Leider konnten wir die so generierte DLL zwar in ein mit Visual Studio geschriebenes Program einbinden und compilieren, beim ausführen gab es aber schwerwiegende Fehler (Absturz!).

13.2.3.2 Ansatz 2: Erzeugen des Objekt-Codes mit Cygwin

Da der DLL Ansatz offensichtlich nicht funktionierte, versuchten wir nun die Objekt -Dateien (erstellt mit: gcc -o test.o test.c) direckt in Visual Studio zu linken. Da aber diese Objekt-Dateien von Cygwin abhängen, mussten wir die Cygwin-Dll zur Laufzeit laden. Zum Compilieren brauchten wir dieses mal die Library (.lib) der in cygwin1.dll definierten Funktionen (siehe 5.3.3.1).

Beim Linken mit Visual Studio stellte sich aber heraus(Fehlermeldungen), dass viele Funktionen nun doppelt definiert waren: Von einer Windows-DLL und von cygwin1.dll. Wir interpretierten dieses Problem als ein reines Linker-Problem und suchten deshalb einen Weg, Visual Studio anzugeben aus welcher DLL welche Funktion zu gebrauchen ist:

1. Generiere cygwin1.def mit dem Komando:
   impdef.exe cygwin1.dll > cygwin1.def
2. Oeffne cygwin1.def mit einem Text-Editor
3. Entferne all Funktionen ausser:
   getitimer, setitimer, gettimeofday, _gettimeofday
4. Generiere die Library:
   `lib.exe /def:cygwin1.def /out:cygwin1.lib`
6. Füge cygwin1.lib in Visual Studio zu Importierte Libraries hinzu.

So konnte nun problemlos kompiliert werden, beim ausführen blockierten(ewiges Warten) aber einige Funktionen ohne eine Fehlermeldunge zu generieren. So konnten wir das Problem so nicht lösen, da wir keine Erklärung für dieses Phänomen hatten (ausser dass der Linker wohl etwas komplett unerwartetes machte), musste weiter nach einer Lösung gesucht werden.

13.2.3.3 Ansatz 3: Funktionierende Lösung

Gcc hat eine Option *-mno-cygwin*. Mit dieser Option kompiliert gcc nicht mit den Cygwin Include-Dateien sondern mit dem Windows-Include Dateien. Es zeigte sich, dass nur wenige gebrauchte Funktionen nicht vom Windows-API definiert werden: getitimer, setitimer, gettimeofday und signal. Für diese Funktionen konnte ein Work-Around gefunden werden: Der Timer wird zum Beispiel nur gebraucht um von Zeit zu Zeit die Sensor-Daten auf zufälliger Basis zu verändern, was für unsere Zwecke sowieso nicht erwünscht ist, somit konnten wir diese Funktions-Aufrufe deaktivieren. Die Funktion gettimeofday wird aber z.B. für die Java-Funktion *System.getCurrentTimeMillies()*, folglich konnte dieser Aufruf nicht einfach gestrichen werden, sondern musste durch die entsprechende Windows-Funktion ersetzt werden.

Durch Anpassungen im Code konnte also emulejos-run mit gcc kompiliert werden. Die so enstandenen Objekt-Dateien haben wir in Visual Studio unter Linker-Optionen als zusätzliche Libraries angegeben und konnten JORGE so kompilieren und schliesslich auch ausführen.

13.2.3.4 Fazit

Obwohl wir zu Beginn das Portieren von emulejos-run als zu Aufwändig deklariert haben, haben wir nun zumindest Teile davon portiert. Und sehen so nun die Fehler die Visual Studio gemeldet hat unter einem anderen Licht, so dass wir in unserer Diplomarbeit nochmals über eine Portierung diskutieren können.
Wir haben sicher die Komplexität im Zusammenspiel von Cygwin mit Visual Studio unterschätzt, müssen aber auch festhalten, dass alle getesteten Emulatoren unter Windows nur mit Cygwin funktionieren.

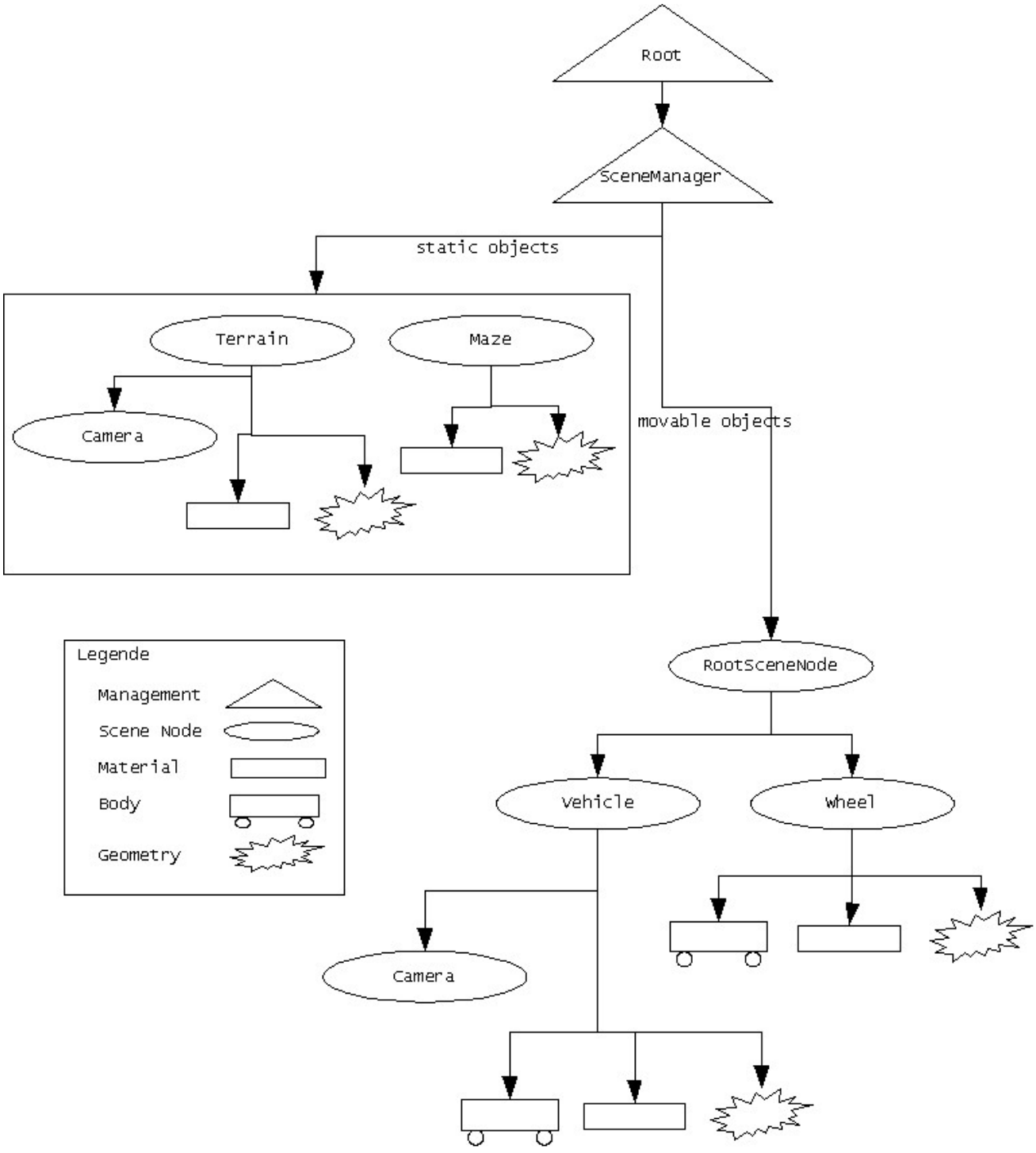
13.2.4 Compilieren: MacOsX

Auch unter MacOsX kann man emulejos-run kompilieren. Leider hatten wir Schwierigkeiten mit Boost, und aus Zeitmangel und Prioritätsgründen haben wir nur kurz versucht, dieses Problem zu lösen.

13.3 Virtual Reality

13.3.1 SceneGraph

Arbeitsaufwand: 5 Stunden



13.3.2 XML Konfiguration des Roboters

Arbeitsaufwand: 20 Stunden

Damit ein schnelles und einfaches Setzen der physikalischen Komponenten möglich ist, haben wir uns entschlossen, die Settings in eine XML Datei zu packen. Die Datei ist im Grossen und Ganzen selbsterklärend. Hier aber noch einige Bemerkungen:

- Über die XML Datei können die Werte für den Roboter, die zwei Motoren sowie die Räder gesetzt werden
- Es ist möglich ein anderes Mesh für den Roboter zu wählen. Allerdings müssen dann alle andere Werten, wie beispielsweise die Räder, die relativ zum Roboter angeordnet sind, angepasst werden
- Die Motoren sind mit einer ID versehen um die Räder einem bestimmten Motor zuzuordnen. Hier können zusätzlich Angaben zum Drehmoment, zur Motorenbremse etc. angegeben werden
- Wichtige Komponenten der Räder sind vor allem die Reibung. Je mehr Reibung desto mehr Halt.
- Ebenfalls können Angaben zur Aufhängung gemacht werden. Diese Angaben werden allerdings beim Robo nicht gebraucht

14 Die Datei: jorge/media/app/model/Robo.ogreode

```
<ogreode>
	<!-- Create the vehicle using the supplied name and body mesh, make it 1.5 units heavy
		e.g. tonnes and offset the center of gravity by half (a metre) to make it more stable
		This puts the CoG somewhere around the wheel hubs, which is probably reasonably
		realistic for a sports-car
	-->

	<vehicle name="Robo">

		<!-- Body tag -->
		<body mesh="Robo.mesh">
			<mass value="0.1" x="0" y="-0.5" z="0" />
		</body>

		<!-- Set the motors (and other drivetrain) parameters, lots of work still to do here!
			0.0225 would be a realistic torque..
		-->

		<!-- Motor 1 -->
		<motor id="0" redline="10" brake="1000">
```

```
        <torque min="0.25" max="0.4" />
    </motor>

    <!-- Motor 2 -->
    <motor id="1" redline="10" brake="1000">
        <torque min="0.25" max="0.4" />
    </motor>


    <!-- Create all the wheels, using the supplied mesh and with the specified offset
         relative to the car. Every wheel has got an id of an engine.
        <steer factor="0" force="8.0" speed="2.0" limit="0.75" />
    -->

    <!-- Wheel options:
        - fds -> force-dependent-slip
    -->

    <!-- Wheel 1 -->
    <wheel mesh="Wheel.mesh" x="6.5" y="-2.5" z="2.5" mass="0.01" engineId="0" >
        <power factor="1" />
        <brake factor="0.001" />
        <contact bouncyness="0" friction="100" fds="1" />
    </wheel>
    <!-- Wheel 2 -->
    <wheel mesh="Wheel.mesh" x="6.5" y="-2.5" z="-6" mass="0.01" engineId="0">
        <power factor="1" />
        <brake factor="0.001" />
        <contact bouncyness="0" friction="100" fds="1" />
    </wheel>

    <!-- Wheel 3 -->
    <wheel mesh="Wheel.mesh" x="-6.5" y="-2.5" z="2.5" mass="0.01" engineId="1">
        <power factor="1" />
        <brake factor="0.001" />
        <contact bouncyness="0" friction="100" fds="1" />
    </wheel>
    <!-- Wheel 4 -->
    <wheel mesh="Wheel.mesh" x="-6.5" y="-2.5" z="-6" mass="0.01" engineId="1">
        <power factor="1" />
        <brake factor="0.001" />
        <contact bouncyness="0" friction="100" fds="1" />
    </wheel>

    <!--

     Set up the suspension spring and damping constants, passing the rate we're going to
         be stepping the world so it can work out the forces needed each step
         We could do this per-wheel, like the other factors, but it's easier to do it this way.
         N.B. You must create the wheels before you can do this!

    -->

    <suspension spring="0.01" damping="0.1" step="0.01" />

</vehicle>

</ogreode>
```

14.1.1 Physik

Arbeitsaufwand: 50 Stunden

Neben der Evaluation der einzelnen Komponenten hat die Physik bestimmt am meisten Zeit in Anspruch genommen. Nicht nur das Verstehen von Ode und dem Wrapper war zeitaufwändig, auch das Zusammenspiel der physikalischen Parametern. Um dies ein wenig zu erläutern, hier ein paar Probleme auf die wir gestossen sind:

- Wir haben versucht das Modell des Roboters so realistisch wie möglich abzubilden. So haben wir auch versucht die physikalischen Werte realistisch zu setzen. Zu Beginn setzten wir das Gewicht des Robos auf 2 Kilogramm. Wir fanden aber schnell heraus das die Simulation unstabil wird. Also landeten wir bei einem Gewicht von 100 Kilogramm.
- Um eine realistische Simulation zu erzeugen, mussten wir allgemein mit grösseren Werten arbeiten, da tiefe Werte weniger Fehlerspielraum lassen.

Es ist eine delikate Angelegenheit das richtige Verhältnis der einzelnen Physikparameter zu finden. Hier ein paar konkrete Abhängigkeiten:

15 Reibung

Um den Roboter zu steuern muss man einen der Motoren aktivieren und den anderen deaktivieren. Bei dieser Variante sind nun die Räder auf der einen Seite blockiert und auf der anderen aktiv. Dadurch ensteht ein Wiederstand an der Seite die blockiert. Dieser Wiederstand müssen wir nun durch Setzen der Reibungskonstante so optimieren, dass genug Reibung ensteht und der Roboter nicht ins Schleudern gerät. Andererseits führt eine hohe Reibungskonstante dazu, dass keine Steuerung mehr möglich ist. Dieses Mittelmass zu finden ist nicht eifach und hat uns viel Zeit und Nerven gekostet.

16 Masse

Bekanntlich spielt die Masse eine ziehmlich essentielle Rolle in der Physik und im speziellen in der Bewegung von Körpern:

Kraft = Masse * Beschleunigung

Die Masse musste nun so gesetzt werden, dass eine vernünftiges Verhältnis entsteht. Wir versuchten durch Erhöhung der Anfangsmasse von 2 Kilogramm, wie oben erwähnt, auf 100 Kilogramm ein etwas stabileres Verhalten zu etablieren. Diese Veränderung der Masse zieht dann aber gründsätzliche Änderungen der restlichen Parameter mit.

17 Kraft

Die Kraft die sich in unserem Fall des Roboters auf die Drehachsen der Räder auswirkten und so in einem Drehmoment resultierten, waren die fundamentale Antriebskraft um den Roboter in der Landschaft zu bewegen. Da diese Kraft wieder direkt im Zusammenhang mit der Masse unserer Räder und des eigentlichen Körpers des Roboters stand, war Vorsicht geboten. Bei zu hoheren Krafteinwirkung auf die Achsen konnten wir einen speziellen Effekt der Räder beobachten, die durch diese hohe Beslastung instabil wurden und die Richtung willkürklich änderten, was es dann natürlich unmöglich machte die Simulation vernünftig zu navigieren.

17.1 Rückwärtsgang

Arbeitsaufwand: 10 Stunden

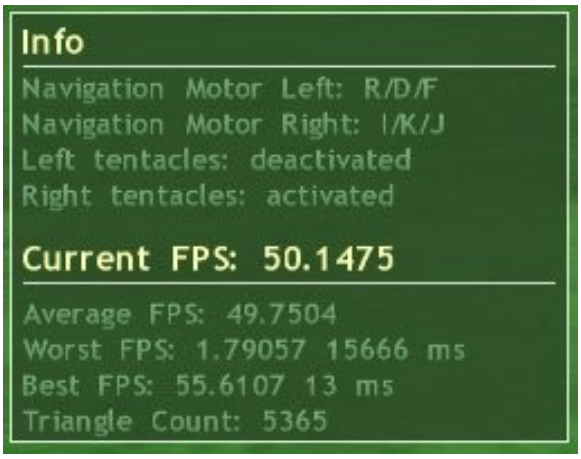
Die bestehende Implementation des Vehicle, die wir als Sample zum Wrapper gefunden haben, konnten wir als gutes Fundament für unseren Roboter gebrauchen. Einer der Punkte, der noch fehlte war der Rückwärtsgang. In dem Physikframework OgreOde, das wir gebracht hatten, wurde der Antrieb wie schon im Punkt Physik erwähnt wurde, durch die Krafteinwirkung auf die Achsen der Räder bewerkstelligt. Für dieses Verbindung des Körpers und der Räder, genannt Joints, sind zwei Achsen definiert. Eine Achse für die Steuerung, nennen wir sie die Y-Achse, und die ander für den Antrieb, hier als X-Achse bezeichnet. Im Fall der Richtungsänderung mussten wir die X-Achse ändern, genauer gesagt wurde die Richtung in welche der Roboter fuhr durch einen Richtungsvektor definiert. Dieser Richtungsvektor musste man nun um 180 Grad drehen, dies musste auf allen Achsen der vier Rädern gemacht werden.

17.2 Fühlerkollision

Arbeitsaufwand: 30 Stunden

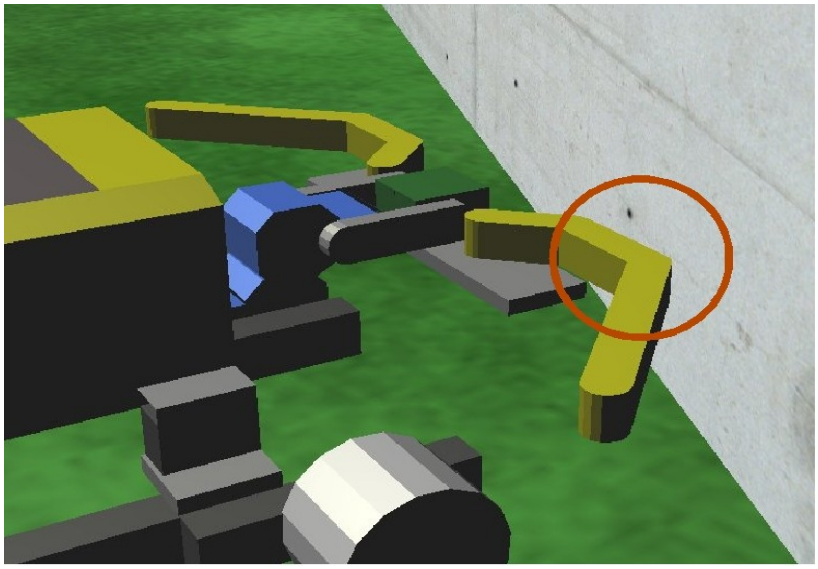
17.2.1 Problemstellung

Um unserem Emulator die Rückmeldung einer Kollision mit einem der Fühler zu geben, mussten wir den beiden Fühlern eine separate Boundingbox zuweisen. Dadurch konnten wir dann in der laufenden Simulation die Kollsionspunkte der Fühler kontrollieren und so feststellen, ob einer der Fühler in Kontakt mit einem Gegenstand, in diesem Fall das Labyrinth, steht. In der Grafik 1 sehen wir die Informationen über den momentanen Zustand beider Fühler. In diesem Fall ist der linke Fühler nicht aktiv und der rechte Fühler aktiv.



Grafik 1: Fühler Infos

Um eine solche Situation herzustellen, könnte die Position des Roboters aussehen wie in der Grafik 2 gezeigt.

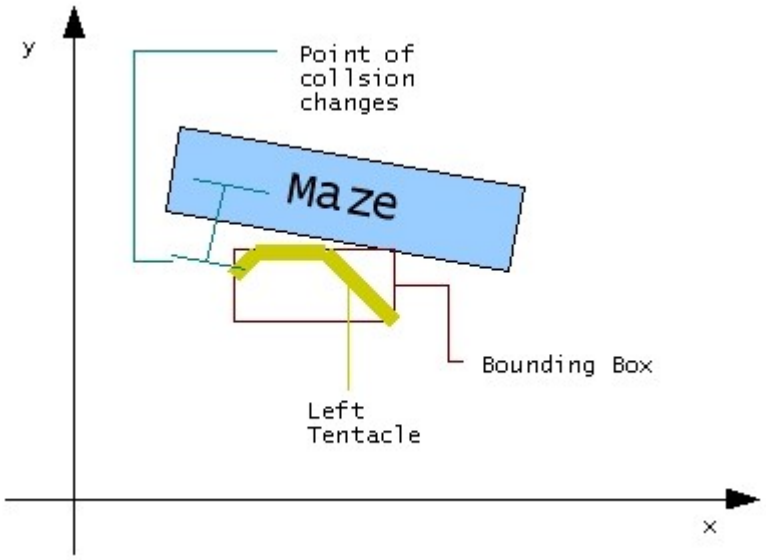


Grafik 2: Kollision mit rechtem Fühler

Der rote Kreis kennzeichnet die Kollisionsstelle des rechten Fühlers mit dem Labyrinth. Mit folgenden Problemen hatten wir bei unserer ersten Implementation zu kämpfen:

- Eindringen der Fühler in das Labyrinth durch Physik
- Fehlerhafte Kollisionserkennung durch Float Abweichung bei der Berechnung

In der nächsten Grafik wird gezeigt, dass bei jeder Berechnung der Szene die Positionen minim aber doch spürbar ändern. Dies Veränderung haben zu den oben genannten Problemen geführt.



Grafik 3: Veränderung der Kontaktpunkte

17.2.2 Lösung

Die Probleme sind enstanden da die ganze Berechnung der Szene bei jeder Neuberechnung von der Vorherigen abweichen kann. Dies ist möglich da mit Float Zahlen gerechnet wird und so bei jeder Berechnung eine gewisse Ungenauigkeit besteht, die dann in Abweichungen resultierten.

Um diese Ungenauigkeit zu igrnorieren mussten wir ein Mechanismus einbauen, welcher es ermöglicht zu unterscheiden wann eine Kollision wichtig für unseren Emulator ist und wann sie ignoriert werden kann. Dies konnten wir dank einem Zeitfaktor, den wir einbauten realisieren. Dieser Zeitfaktor überprüfte wie lange es her ist seitdem eine Kollision statt gefunden hat und reagiert nur wenn die gesetzte Zeit abgelaufen ist.

17.3 Rendering System Probleme

Durch die objektorientierte modulare Implementation des Oger3D Framework ist es möglich je nach Bedarf DirectX und OpenGL als Schnittstelle zu der Grafikkarte zu benützen. Diese Option ist natürlich nur auf Windows verfügbar, auf Linux besteht keine DirectX Implementation. Diese Auswahl ist während der Laufzeit des Progamms zu treffen. Auf Windows haben wir in Bezug auf das Rendering System folgende Probleme erkennen müssen:

- Mit DirectX Version 7 treten Probleme mit der Texture auf
- Mit DirectX Version 9, das wir auf Windows als Rendering System empfehlen, ist die Funktion für das Anzeigen der Boundingboxen von dem Roboter nicht möglich

18 Libraries & Tools

18.1 Libraries

18.1.1 Ogre

Arbeitsaufwand: 100 Stunden

Für die Umsetzung der Grafik haben wir Ogre gewählt. Um Erfahrung mit C++ zu sammeln, haben wir bewusst nach einem Framework gesucht, welches in dieser Sprache realisiert wurde. Da wir bis anhin keine Erfahrung mit 3D Libraries hatten, konnten wir nicht recht einschätzen, ob uns dieses Framework genügen wird. Punkte die dafür gesprochen haben:

- Entwicklung von Ogre erstreckt sich bereits über mehrere Jahre
- Objekt-orientierter Ansatz
- Grosse und aktive Community
- Viele, öffentlich zugängliche Beispiele

So entschlossen wir uns ins kalte Wasser zu springen und setzen auf Ogre. Wir sind froh, diese Entscheidung getroffen zu haben.

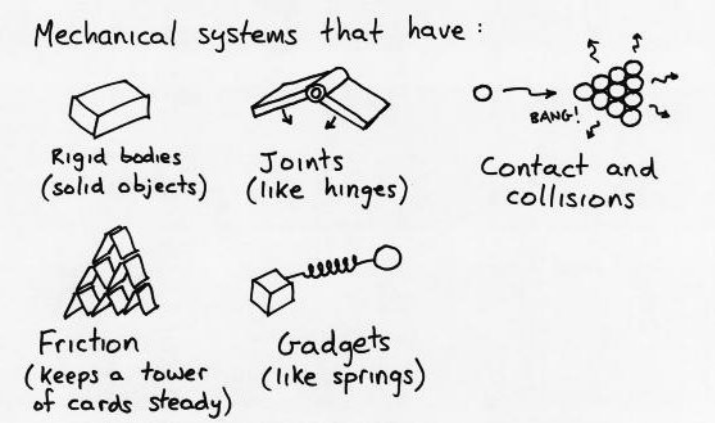
Weitere Informationen zu Ogre, wie Manual und Tutorials, findet man auf der Webpage, welche sehr sorgfältig verwaltet wird. Die URL lautet: http://www.ogre3d.org.

18.1.2 Ode

Arbeitsaufwand: 40 Stunden

Ode in der Version 0.5 ist das von uns gewählte Physics Framework. Entscheidend für diese Wahl war hauptsächlich das bereits vorhandene Zusammenspiel zwischen Ogre und Ode. Dieses wird über einen Wrapper realisiert, der sich bereits bewährt hat.

Sinn und Zweck von Ode ist Simulation von mechanischen Systemen. Folgende Features werden von Ode abgedeckt:



Weiter Informationen zu Ode findet man unter folgender URL: http://www.ode.org/

18.1.3 OgreOde Wrapper

Arbeitsaufwand: 80 Stunden

Ein wichtiger Bestandteil unserer Applikation war der Austausch von Informationen zwischen der graphischen und der physikalischen Welt, sprich zwischen Ogre und Ode.

Wir haben hierzu bereits eine Lösung gefunden, welche uns sehr viel Arbeit abgenommen hat (mehr Infos hier http://www.green-eyed-

monster.com/xoops/modules/news/, Version 1.0.1). Für JORGE konnten wir einen Grossteil dieses Codes übernehmen. Allerdings waren einige Modifikationen notwendig.

18.1.4 TinyXML

TinyXML (http://sourceforge.net/projects/tinyxml, Version 2.3.4) ist eine sehr simple und kleine „Library“ um XML Files zu parsen. Die Library wurde gebraucht um die Konfigurationen für den Roboter einzulesen.

18.1.5 Boost

Arbeitsaufwand: 10 Stunden

Boost (http://www.boost.org) ist eine freie C++-Bibliothek, die auf der C++ Standard-Bibliothek aufsetzt und auf verschiedensten Platformen eingesetzt werden kann. In JORGE haben wir Threads aus Boost Version 1.3.2 benutzt, um den Emulator unabhängig von der 3D-Welt auszuführen. Die gemeinsamen Daten für die Motoren und die Sensoren werden mittels Boost-Mutexes vor gleichzeitigem Zugriff geschüzt.

18.1.6 Cygwin

Arbeitsaufwand: 10 Stunden

Cygwin (http://www.cygwin.com) ist eine “Linux-Umgebung“ für Windows. Wir setzten Cygwin in der Version 1.5.17-1 ein, um diverse Emulatoren zu testen, und schlussendlich auch, um die gewählte Lösung zu kompilieren.

18.1.7 LeJos

Arbeitsaufwand: 70 Stunden

LeJos (http://lejos.sourceforge.net) ist eine kleine Virtuelle Maschine. Sie funktioniert als Ersatz für die offizielle Lego-Firmware des Lego Mindstorm RCX. LeJos definiert eine Java-API, die benutzt werden kann um Programme für den RCX zu schreiben, und bietet auf der Homepage Kompiler und Tool, um diese Programme auf den RCX übertragen. Wir nutzen LeJos in der Version 2.1.0.

18.1.8 Doxygen

Doxygen (http://www.doxygen.org) benutzen wir um aus den Kommentaren im Code eine Dokumentation zu generieren.

18.2 Tools

18.2.1 Blender

Arbeitsaufwand: 50 Stunden

Für das Object Modeling haben wir auf OpenSource gesetzt. Vorteile von Blender sind:

1. kostet nix
2. plattform unabhängig
3. grosse und aktive Community
4. viele nützliche Tutorials (auch Video)
5. Exporter für das Ogre Mesh XML Format vorhanden

Es hat sich gezeigt, dass die Bedienung von Blender zu Beginn eher mühsam ist, nicht zuletzt wegen dem GUI und natürlech wegen den vielen Funktionen. Doch sobald man die eine oder andere Tastenkombination kennt und die Philosophie des Tools verstanden hat, kann die Arbeit sehr viel Spass machen.
Wir haben Blender in der Version 2.36 benutzt, aktuelle Versionen gibt es auf der Homepage des Projekts: http://www.blender3d.org

18.2.2 Terragen & Freeworld3D

Arbeitsaufwand: 20 Stunden

Um ein Terrain zu erstellen haben wir die Kombination Terragen & Freeworld3D gewählt.
Für das Kreieren eines realistischen und doch für unsere Bedürfnisse geeigneten Terrains war Terragen für unsere Bedürfnisse zu mühsam. Das Tool scheint nicht für die Generierung einer benutzerdefinierten Heightmap gemacht. Deshalb haben wir uns entschlossen die kommerzielle Lösung Freeworld3D für diesen Task zu gebrauchen. Die generierte Heightmap aus Freeworld3D konnten wir dann in Terragen importieren umso eine passende Texture zu erzeugen.

18.2.3 IDE

CodeForge schien sehr schnell und ausgereift, gerade in Bezug auf CodeCompletion und CodeBrowsing. Zu unserem Bedauern fanden wir jedoch einen Bug, der das Arbeiten mit CodeForge verunmöglichte. Bei bestimmten Files verursachte CodeForge eine OutOfMemory Exception. Wir fanden dann Tricks dieses Problem zu umgehen und setzten parrarell Debug Libraries von CodeForge ein. Nach zwei Wochen Zusammenarbeit mit einem Entwickler von CodeForge konnte das Problem gelöst werden.

Wir hoffen nun, dass wir gut gewappnet sind für die kommende Diplomarbeit.