Hochschule für Technik, Wirtschaft und Kultur (HTWK) Leipzig
Fakultät Informatik, Mathematik und Naturwissenschaften
Studiengang Informatik Bachelor

Bachelorarbeit

zur Erlangung des akademischen Grades eines
Bachelor of Science (B.Sc.)

Austauschbare Module zur Integration von KI in den Nao-Simulator "Spielwiese"

Autor: Manuel Bellersen
Betreuer: Prof. Dr. rer. nat. Karsten Weicker
Gutachter: Prof. Dr. rer. nat. Klaus Bastian
Abgabe:

Ich erkläre hiermit, dass ich diese Bachelorarbeit selbstständig ohne Hilfe dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen sind als solche einzeln kenntlich gemacht.

Diese Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht worden.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Ort, Datum	Unterschrift

Inhaltsverzeichnis

1 Einleitung

In dieser Kapitel werden die Grundlagen für diese Arbeit bereitgestellt. Dabei wird der Grund aufgezeigt, warum diese Arbeit geschaffen wurde und die benutzten Techniken und Technologien aufgezeigt und kurz erläutert.

1.1 Motivation

Während meines Praktikums im Nao-Team der HTWK-Leipzig entwickelte ich den Nao-Roboter-Fußball-Simulator „Spielwiese“. Dabei handelt es sich um eine Anwendung, die ein Fußballfeld, einen Ball und eine Anzahl von Roboter-Spielern für zwei Mannschaften bereitstellt und deren Interaktionen simuliert. Dies läuft auf einem normalen PC und soll später auch auf dem Nao-Roboter funktionsfähig sein. Auf dem PC dient die Simulation der Erstellung und dem Testen von Strategien für Wettkämpfe.Dafür steht unter anderem ein grafisches Benutzerinterface zur Verfügung. Auf dem Roboter laufend, soll es das Spiel mittels aktueller Daten mehrere Schritte voraus berechnen und somit bei der Erstellung eines geeigneten Folgezug beistehen.

1.2 Problemstellung und Zielsetzung

Dem Roboter steht eine für die heutige Zeit nur schwache Rechenmaschine zur Verfügung. Diese verarbeitet die reinkommenden Daten von verschiedenen Sensoren und Kameras und erzeugt daraus und unter Anwendung von unterschiedlichen Algorithmen Signale für die Motoren. Dinge, wie Laufen, Aufstehen, Umsehen und Schießen funktionieren schon ganz gut. Ein großes Problem ist allerdings das Zusammenspiel der Teammitglieder. Es gibt dabei verschiedene Möglichkeiten dies zu Realisieren. Das Ziel dieser Arbeit ist es nun, passende Schnittstellen in den Simulator zu integrieren, die eine einfache Möglichkeit schaffen, künstliche Intelligenzen (KI) auf dem PC zu entwickeln, zu testen und auch auf dem Roboter anzuwenden.

Weiterhin soll es ohne großen Aufwand möglich sein, die künstliche Intelligenz auszutauschen. Für diesen Bereich werden häufig Skriptsprachen bevorzugt.

Dies birgt jedoch das Problem, dass die Skripte in der Ausführung meist langsamer und ressourcenintensiver sind, als native Implementierungen, was auf dem Roboter zu Schwierigkeiten führen könnte. Deswegen soll neben dem Einbau der Schnittstelle zu einer Skriptsprache, auch der zur Nutzung der nativen Sprache vollbracht werden. Weiterhin werden durch das Vorhandensein beider Möglichkeiten Implementierungen entsprechend gleicher Algorithmen an diesen Schnittstellen auf Laufzeit, Ressourcenverbrauch und Codegröße verglichen.

1.3 Abgrenzung

Die Gebiete der künstlichen Intelligenz, der Simulation und der Robotik sind groß. Ebenso vielseitig ist das Feld der nutzbaren Werkzeuge und Wege, die zum Ziel führen. Die meisten Dinge daraus werden hier nicht behandelt oder nur ganz kurz angerissen. Es wird letztendlich keine unbedingt sinnvolle künstliche Intelligenz für die Testzwecke erstellt werden. Für die Vergleiche reichen schon sehr einfache Beispiele. Der Simulator wird im Ganzen nur grob erklärt. Die Stellen, die wichtig für die Integrationen sind, werden jedoch deutlicher beleuchtet. Die Robotik fällt ganz raus, da sie für diese Arbeit nicht weiter von Bedeutung ist. Die genutzten Utensilien werden teilweise ausführlicher vorgestellt, vereinzelt auch Alternativen genannt.

1.4 Überblick

Beginnend erkläre ich den Unterschied zwischen Kompilieren, Interpretieren und dem Bezug zu Skriptsprachen, sowie was ein Nao-Roboter und der RoboCup ist. Darauf aufbauend folgt eine Erläuterung zum Simulator „Spielwiese“. Die genutzten Technologien und die Gründe dafür sind im nachfolgenden Abschnitt aufgezeigt.

Anschließend folgt der Hauptteil dieser Arbeit. Darin gehe ich auf den Aufbau und die Integration der Schnittstellen in den Simulator ein. Weiterhin sind dort Vergleiche zu Laufzeiten und Ressourcenverhalten zwischen den Erweiterungen an den Schnittstellen an Hand von 4 Beispielen zu finden. Dabei wird kurz auf

die Syntax der beiden Sprachen und ein paar Neuerungen der einen Sprache mit dem erst kürzlich verabschiedeten neuen Standard eingegangen. Damit einher geht die Auswertung dieser Ergebnisse. Zum Abschluss folgt das daraus resultierende Fazit, sowie der Ausblick.

1.5 Ergebnisse

Die Angestrebte Integration der Schnittstellen in bzw. um den Simulator, damit die Nutzung von Module ermöglicht wird, ist erfolgreich vollbracht. Zwei Möglichkeiten sind für C++ und eine für Lua geschafften wurden.

Die genutzte Skriptsprache brachte, wie zu erwarten, eine langsamere Ausführungsgeschwindigkeit und einen höheren Ressourcenverbrauch. Der Ressourcenverbrauch hält sich trotz Schwankungen relativ konstant wohingegen die Ausführungsgeschwindigkeit durch schon wenig Quelltext stark verlangsamt werden kann.

Der Quellcode, der für die Beispiele geschrieben wurde, ist annähernd gleich, sowohl in Form, als auch in der Anzahl der geschriebenen Zeichen. Die ist aber auf die Kürze der Beispiele zurückzuführen. Für eine bessere Bewertung währen viel längere Beispiele nötig

Beide Varianten der Modulerstellung, in C++ und Lua, haben ihre Berechtigung. Für das schnelle Ausprobieren ist Lua eindeutig besser geeignet, solange der Code etwas länger ist, bzw. der Code auch während der Laufzeit verändert werden soll. Bei sehr kurzem Quelltext ist die Zeit für das Kompilieren der C++-Module vernachlässigbar.

2 Grundlagen

2.1 Kompilieren, Interpretieren und Skripte

Software wird mit Programmiersprachen entwickelt. Zu Beginn der Computer war dies der berüchtigte Binärcode mit vielen Einsen und Nullen, später entstand zum Beispiel Assembler, C, C++, Fortran, Haskell, Java, Lua, Pascal, und unzählige weitere. Dabei unterscheidet man bei diesen Sprachen zwischen Kompilieren und Interpretieren.

2.1.1 Kompilieren

Beim Kompilieren wird der Quelltext in Maschinensprache übersetzt, welcher dann direkt auf dem Computer ausgeführt werden kann. Der Compiler kann während des Übersetzungsvorganges verschiedene Optimierungen, wie Erkennung und Beseitigung von nicht genutzten Codefragmenten und Umschreiben von Code, vornehmen, um die Effizienz, sowohl der Laufzeit, als auch des Ressourcenverbrauchs, zu erhöhen. Es kann dabei soweit gehen, dass die dabei erstellte ausführbare Datei optimal an die zugrunde liegende Hardware angepasst und dadurch eine maximale Effizienz erreicht wird, die Datei jedoch nur noch genau auf dieser Hardwarekombination läuft.

2.1.2 Interpretieren

Das Interpretieren verläuft parallel zur Ausführung. Der Code wird Schritt für Schritt ohne Übersetzungsphase sofort ausgeführt. Somit sind auch Änderungen während der Laufzeit möglich. Solange ein entsprechender Interpreter auf dem Computer vorhanden ist, kann das Programm lauffähig gemacht werden. Optimierungen sind dabei kaum möglich.

2.1.3 Just-In-Time-Compiler

Ein Zwischending sind Just-In-Time-Compiler (JIT-Compiler). Diese kompilieren den Quelltext, was allerdings genau während des Ausführens passiert. Damit

bekommt man die Optimierungsvorteile des normalen Compilers und die Vorteile von Interpretern, wodurch der Code auf allen Maschinen, für die ein entsprechender JIT-Compiler verfügbar ist, läuft und während der Ausführung der Quellcode geändert werden kann. Als Nachteil ist hier der höhere Speicherverbrauch und die langsamere Startzeit gegenüber normal kompilierten Code zu nennen.

2.1.4 Auswahlkriterien

Je nach Hard- und Softwareumgebung gibt es heutzutage unterschiedliche Anforderungen an die Anwendungen. Auf schwachen PCs und Großcomputern zählt vor allem schnelle Laufzeit und niedriger Ressourcenverbrauch, bei normalen Heim- und Firmen-PCs hingegen ist die Aktualität und schnelle und einfache Anpassung an eigene Bedürfnisse wichtig. Mit „DotNet“ und der Java „Virtual Machine“ (VM) gibt es zwei große Plattformen in der JIT-Kompiler Welt, die genau letzteres bedienen. Für ersteres werden noch Compiler bevorzugt. Jedoch gibt es auch hierbei genügend Szenarien, die Anpassungen an dem Programm unter anderem während der Laufzeit erfordern und auch Erweiterungen ohne Programmcodeänderungen ermöglichen sollen. Bei solchen Fällen kommen deswegen nicht selten Skriptsprachen zum Einsatz.

2.1.5 Skriptsprachen

Skriptsprachen sind, durch zum Beispiel dynamische Variablen- und Funktionsdeklarationen, Vererbung und automatische Speicherverwaltung, einfacher gehalten, als andere Programmiersprachen. Auch wird Quellcode, welcher in diesen Sprachen geschrieben wurde, im Allgemeinen interpretiert. Häufig ist aber auch eine Kompilierung möglich. Einige Skriptsprachen, wie Python, Perl und LISP, besitzen sowohl Interpreter, als auch Compiler, und existieren auch als richtig selbständige Programmiersprachen.

Unter Umständen ist der Kompromiss zur Nutzung einer extra Skriptsprache innerhalb eines kompilierten Programms akzeptabel. Im Heimbereich gibt es dazu unzählige Beispiele. Nahezu jedes größere Spiel enthält Skript-Code. Viele

Anwendungen stellen Schnittstellen für Skript-Dateien zur Verfügung um durch sogenannte „Add-Ons“ die eigene Funktionalität zu erweitern. Hierbei fallen die nötige Belastung von Prozessor und Arbeitsspeicher nicht so sehr ins Gewicht.

Bei schwachen Maschinen und Großrechnern ist man jedoch versucht, auf Skripte bei Berechnungen zu verzichten. Zur Programminitialisierung und für einmalige, kurze und nebensächliche Aufgaben können diese aber dennoch nützlich sein. Für große und intensive Berechnungen lohnt es sich hier auf andere Mittel zurückzugreifen.

2.2 Nao-Roboter

Abbildung 1: Logos von Aldebaran Robotics und Nao

2.2.1 Geschichte

Nao ist der Name eines Robotertyps der Firma Aldebaran. 2004 begann das Projekt Nao und führte 2005 in Frankreich zur Gründung von Aldebaran, bzw. Aldebaran Robotics.[2] Im nachfolgenden Jahr wurde das Projekt erstmals öffentlich vorgestellt. Bereits 2007 gelang der Durchbruch, indem in der Kategorie „Standardplattform“ beim RoboCup der Sony Roboter Aibo durch den Aldebaran Roboter Nao ausgewechselt wurde.[3] Dadurch erlangte sowohl Aldebaran, als auch der Nao Weltruhm.

2.2.2 Technische Details

Der Roboter ist 58cm hoch und hat in der RoboCup-Version 21 und in der akademischen Version 25 Freiheitsgrade. Dieser Unterschied kommt durch Extrafreiheitsgrade je Hand und Arm zum Tragen, da diese für das Fußballspielen nicht essentiell sind. Ein Freiheitsgrad ist ein Systemparameter, wobei das Sys-

tem durch seine Parameter eindeutig bestimmt ist und die Änderung einzelner Parameter keine Auswirkungen auf die anderen hat.[4]

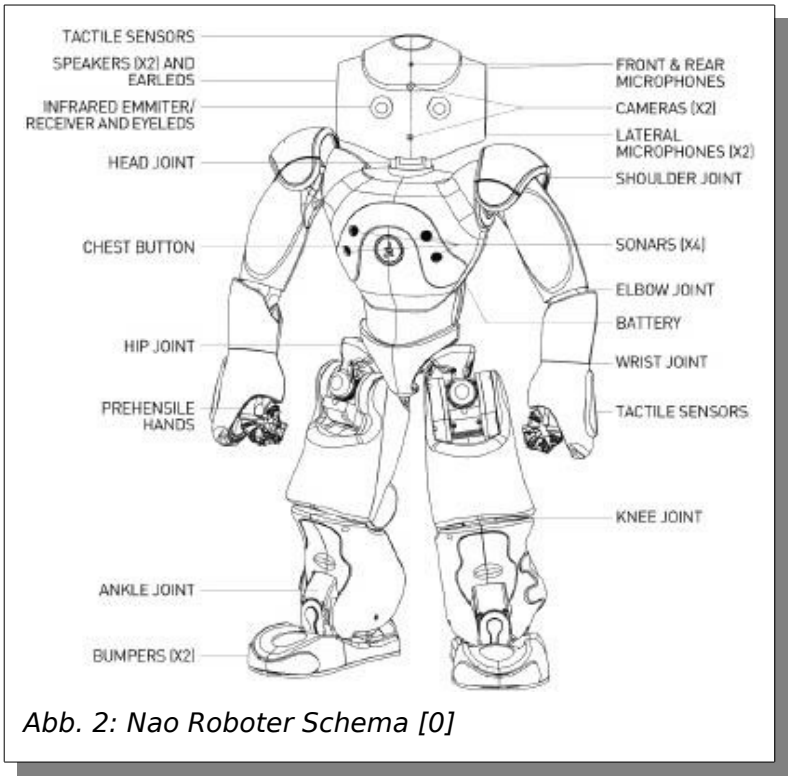

Abb. 2: Nao Roboter Schema [0]

Körperhöhe	~58cm	Körpergewicht	~5.2kg
Freiheitsgrade	2 Kopf, 5 je Arm, 1 Becken, 5 je Bein, 1 je Hand		
Multimedia	2 Lautsprecher, 4 Mikrophone		
Kamera	2 CMOS digital		
Netzwerk	Wi-fi, Ethernet		
Aktoren	Hall-Effekt Sensoren, dsPICS Mikrocontroller, kernlose DC Motoren		
Sensoren	32 Hall-Effekt Sensoren, 1x 2 Achsen Gyrometer, 1x 3 Achsen Beschleunigungsmesser, 2 Stoßfänger pro Fuß, 2 Kanal Sonar, 2 infrarot Sensoren, Taktile Sensoren an Kopf und Händen		
LEDs	Kopf (12), Augen (2x8), Ohren (2x10), Torso (1), Füße (2x1)		
Motherboard	X86 AMD Geode 500 MHz CPU, 256 MB SDRAM, 2 GB Flash Memory		

Tabelle 1: Nao Spezifikation [5]

In Abbildung 2 und Tabelle 1 ist die Spezifikation des Nao Roboter dargestellt. Man kann deutlich sehen, dass die Recheneinheit mit vielen und verschiedenen Bausteinen interagieren muss. Das Auslesen und Setzen der Werte ist indessen noch einfach und schnell erledigt. Hinzu kommt aber noch die ganze Logik, die in Software gefertigt, ebenfalls vom Prozessor bearbeitet werden muss. Dabei kann jener schnell an seine Grenzen geraten. Vor allem muss beachtet werden, dass die ganzen Berechnungen mehrmals je Sekunde ausgeführt werden müssen, um einen flüssigen Ablauf zu gewährleisten.

Features:

• Low power, Full x86 compatibility
• Processor functional blocks:
  • CPU Core, Graphics Processor, Display Controller, Video Processor, Video Input Port
  • GeodeLink Control Processor, Interface Units, Memory Controller, PCI Bridge
  • Security Block (128-Bit Advanced Encryption Standard (AES) – (CBC/ECB), True Random Number Generator)

Specification:
• Processor frequency 500 MHz • 64K Instruction / 64K Data L1 cache and 128K L2 cache • Split Instruction/Data cache/TLB • DDR Memory 400 MHz • Integrated FPU with MMX and 3DNow! • 9 GB/s internal GeodeLink Interface Unit (GLIU) • 481-terminal PBGA (Plastic Ball grid array) • GeodeLink active hardware power management

Tabelle 2: AMD Geode LX800 Spezifikation [6]

Tabelle 2 bietet einen guten Überblick über die Spezifikation des Gehirns, des AMD Geode LX800, des Nao Roboter. Es handelt sich dabei um ein „System-on-

a-Chip“ (SoC). Hierbei befinden sich alle nötigen Rechen- und Speichereinheiten des Systems auf einem integrierten Schaltkreis.

Im Vergleich zu handelsüblichen Heim-PCs und Laptops ist die zur Verfügung stehende Leistung eher gering. Um also möglichst viel aus diesem System heraus zu holen, sollten Programme sowohl möglichst hardwarenah, als auch sehr auf Ressourcenverbrauch und Rechenzeit achtend, erstellt werden.

2.3 RoboCup

Abb. 3: Logo des RoboCup

Das Gebiet, in dem man wohl die meisten Naos antrifft, ist die „Standard Platform League“ des RoboCup.

2.3.1 Geschichte

Bei einem Workshop 1992 kam das Gespräch darauf zu sprechen, Fußball als Vorzeigebild der Wissenschaft zu nehmen. Bereits ein Jahr später wurde daraufhin in Japan die „Robot J-League“ gegründet und schon nach einem Monat auf Grund internationalen Interesses zu „Robot World Cup Initiative“, kurz RoboCup, umgenannt. 1997 wurde in Japan die erste offizielle RoboCup Konferenz abgehalten, während dessen traten über 40 Mannschaften gegeneinander an und über 5000 Zuschauer waren anwesend.[7] Seit dem ist die Anzahl der Mitglieder und Interessenten ständig gestiegen. Das große Ziel bis in etwa zur Mitte des 21. Jahrhunderts ist, dass ein Team autonomer Fußball-Roboter gegen die dann amtierenden FIFA World Cup Champions gewinnt.[8]

2.3.2 Aufteilung

Inzwischen ist die RoboCup-Initiative soweit gewachsen, dass auch das eigentliche Interessenfeld erweitert wurde. So existiert neben dem RoboCupSoccer, bei dem es weiterhin um die ursprüngliche Fußballidee geht, RoboCupRescue für Rettungsroboter, RoboCupJunior um junge Menschen an das Roboterthema heran zu führen, und RoboCup@Home für Haushaltsroboter.[9]

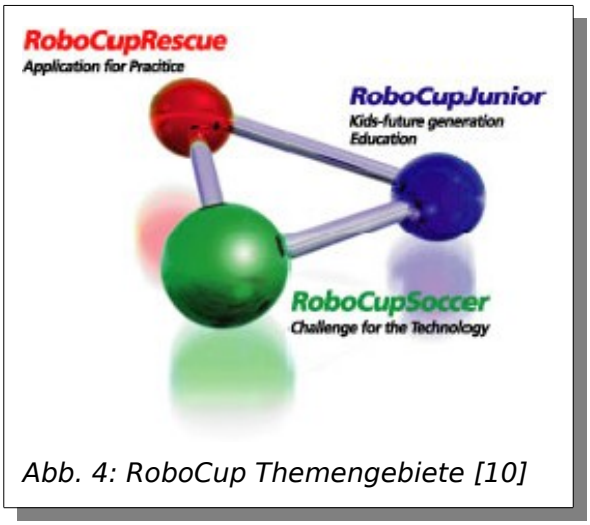

Abb. 4: RoboCup Themengebiete [10]

Die Bereiche sind selber meist auch nochmal unterteilt. Im RoboCup Soccer gibt es 5 Ligen[11]:

- Humanoid
- Middle Size
- Simulation
- Small Size
- Standard Platform

2.3.3 Standard Plattform Liga

Die Naos kommen dabei seit 2008 in der Standard Plattform Liga (SPL) zum tragen. Hierbei stehen sich 2 in der Hardware identische Roboter-Teams gegenüber und versuchen autonom gegeneinander Fußball zu spielen.

Um die mitmachenden Teams auch zu fordern und dem eigentlichen Endziel näher zu kommen, werden des öfteren die Regeln angepasst. Im Gegensatz zum letzten Jahr waren dieses Jahr 4 statt 3 Roboter pro Team auf dem Feld. Für nächstes Jahr ist geplant, dass die Tore, bei denen derzeit eines blau und das andere gelb ist, einheitlich koloriert werden sollen. Dadurch verschwindet ein statisches Merkmal zur Lokalisierung. Die Teilnehmer müssen sich nun also darüber den Kopf zerbrechen, wonach die Roboter sich dann noch lokalisieren können und dieses ihnen beibringen.

Abb. 5: Naos in der SPL [12]

2.4 Simulator „Spielwiese“

2.4.1 Anwendungsgebiete

Während meiner Praktikumsphase habe ich den Simulator „Spielwiese“ erstellt. Dieser soll es ermöglichen, sowohl künstliche Intelligenzen für die Roboter zu erstellen, als auch diese zu verbessern und verschiedene künstliche Intelligenzen gegeneinander zu testen. Für diese Fälle wird eine grafische Benutzeroberfläche angeboten. Dabei kann die Simulation je nach Wunsch, wenn es die zur Verfügung stehende Hardware erlaubt, schneller bzw. langsamer ablaufen als in Wirklichkeit. Da die Realität zu komplex ist, um sie einfach in einer Simulation abzubilden, wurde sie an einigen Stellen vereinfacht. Zum Beispiel läuft die ganze Simulation in einer Ebene statt im 3 dimensionalen Raum ab. Auch sind die Roboter und der Ball nur als Kreis abgebildet. Dadurch, dass die Real-Zeit nicht originalgetreu dargestellt werden kann, wird zwischen zwei Zeitschritten

interpoliert. Die Kollisionserkennung und -behandlung ist ebenfalls gegenüber der Realität vereinfacht wurden. Durch diese ganzen Approximationen kann es passieren, dass das, was unter Umständen in der Simulation geschieht, wie zum Beispiel das Durchdringen des Balls bzw. Roboters durch einen anderen Roboter oder der Torpfosten, nicht in der Wirklichkeit möglich wäre. Jedoch reicht dies für die vorgesehenen Aufgaben aus.

Ein weiteres Anwendungsgebiet des Simulators ist der Einsatz dessen auf dem Roboter. Dieser soll somit in der Lage sein, den Simulator zur Auswertung von Folgezügen zu nutzen. Entsprechend wurde der Simulator so entwickelt, dass es möglich ist, möglichst viele Zeitschritte in einer kleinst möglichen Realzeit durchzugehen.

2.4.2 Aufbau

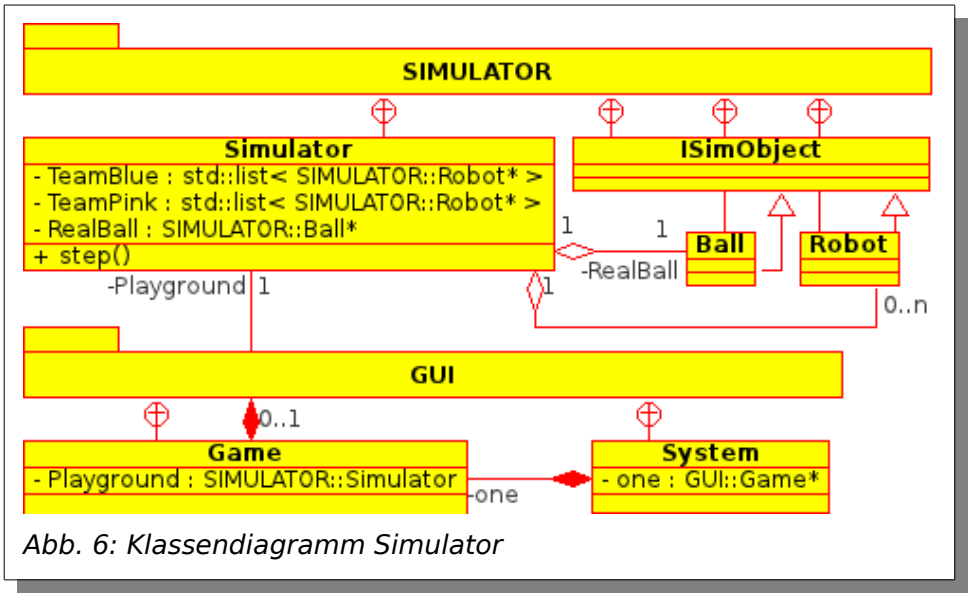

Abb. 6: Klassendiagramm Simulator

In dem Klassendiagramm in Abbildung 6 sind die für diese Arbeit nötigen Klassen und Beziehungen dargestellt. Die Simulator-Klasse stellt die zentrale Funktionalität zur Verfügung. Unter anderem hält sie die Daten für die Roboter und den Ball. Die step()-Methode dieser Klasse führt die entsprechenden Befehle aus um zum nächsten Zeitpunkt zu gelangen. Es werden dabei folgende 4 Abschnitte begangen:

1. Kontrollierung auf Änderung der Spielsituation

2. Ausführung einer Aktion pro ISimObject
3. Anwendung der Regeln und Erkennung und Behandlung von Kollisionen
4. Überprüfung der Torsituation

Die aktiven Simulationsobjekte sind vom Typ „Ball“ bzw. „Robot“ und stammen von der abstrakten Klasse „ISimObject“ ab. Auf diesem Weg ist es möglich, sowohl die Roboter, als auch den Ball identisch anzusprechen und damit teils gleich, aber auch komplett unterschiedliche Aktionen auszuführen.

Diese und die weiteren benötigten Klassen liegen in einem Ordner und sind so konzipiert, dass keine externen Bibliotheken benötigt werden. Sie können somit simpler wiederverwendet und portiert werden. Die Nutzung des Simulators ist denkbar einfach. Man legt sich ein Objekt vom Typ „Simulator“ an. Dabei sind keine Parameter von Nöten. Die Startpositionen des Balls und der Roboter, sowie die Anzahl derer sind bisher noch fest codiert. Als nächstes kann an der gewünschten Stelle die step()-Methode des Simulators aufgerufen und gegebenenfalls dessen Rückgabewert untersucht werden. Dieser gibt an, ob und wenn ja, in welches Tor der Ball gerollt ist. Um Änderungen an dem Ball oder den Robotern zu tätigen, können diese vom Simulator erfragt und über die bereitgestellten Methoden bearbeitet werden.

2.4.3 Benutzerinterface

Auf diese Weise ist deren Nutzung in Kommandozeilenprogrammen (CLI, engl. Command Line Interface) und Programmen mit grafischer Benutzeroberfläche (GUI, engl. Graphical User Interface) relativ problemlos möglich. Beide Varianten stehen zur Nutzung bereit. Für beide GUI-Versionen wird zusätzlich die Bibliothek „Simple DirectMedia Layer“ (SDL) genutzt.

Die erste GUI, „gui/nss-gui“, dient vor allem dem Test des Simulators. Es können per Maus der Ball und die Roboter versetzt, gedreht und die Zielpositionen verändert werden. Per Tastatur kann die Anzahl der Zeitschritte pro Sekunde zwischen 1 und 200 eingestellt werden. Auch ist es möglich, dass jeder Zeitschritt manuell per Hand getätigt werden kann. Außerdem besteht noch die

Möglichkeit, die künstlichen Intelligenzen für jedes Team zu aktivieren und zu deaktivieren, das Spiel wieder auf die Startposition zu stellen und die Zielvisualisierung auszuschalten.

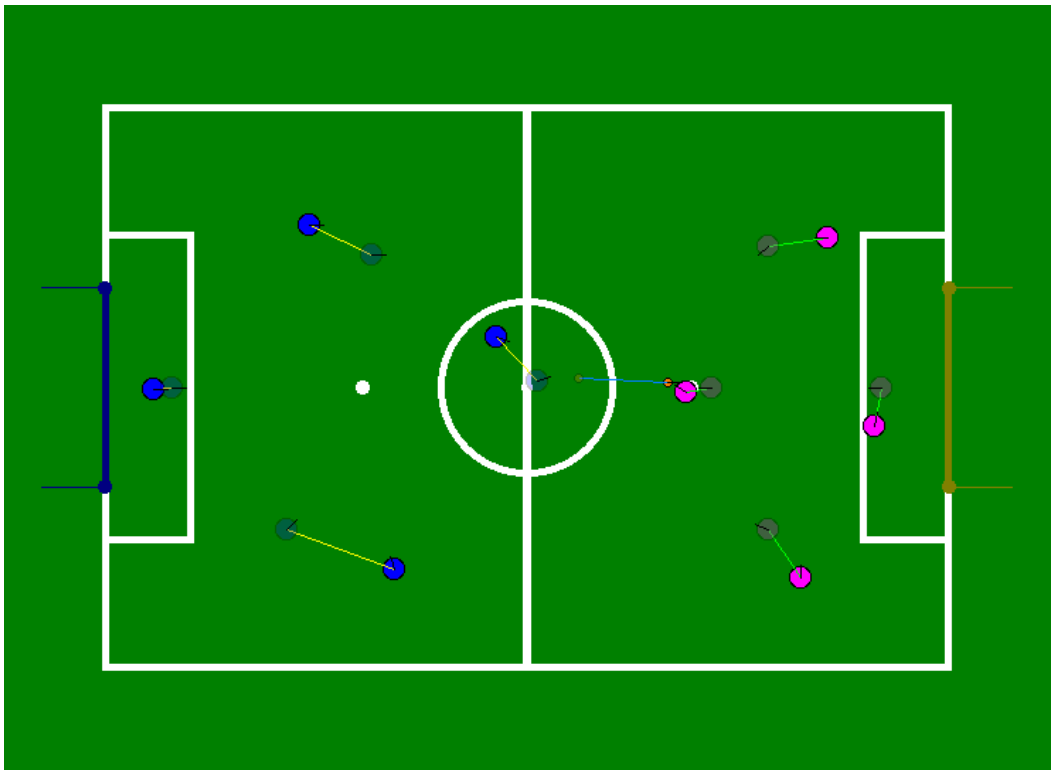

Abb. 7: grafische Benutzeroberfläche des Simulator

Die zweite GUI, „gui-linux-xinput2/nss-gui-lxi2“, ist speziell auf Linux ausgerichtet und ermöglicht es, mit jeweils einer Maus ein Team zu steuern. Dabei kann jedoch nur die Zielposition und Zielrotation der Roboter gesetzt werden. Ein direktes Eingreifen ist nicht möglich. Dadurch, und durch die fehlende Integration der Nutzung der künstlichen Intelligenz, ist diese GUI eher zum Spaß für zwei Spieler gedacht.

Die Kommandozeilenprogramme sind so gebaut, dass keine Interaktionen vom Benutzer möglich sind. Einzig beim Starten können den Anwendungen als Parameter Angaben über die Anzahl der zu berechnenden Zeitschritte gemacht werden. Die Programme laufen dann mit maximaler Geschwindigkeit, bis sie die gewünschte Anzahl erreicht haben.

2.5 Genutzte Technologien

Während der Arbeit am Simulator konnte ich von einigen Technologien Gebrauch machen. Diese werde ich jetzt kurz vorstellen. Ganz wichtig bei der Auswahl waren sowohl die Verfügbarkeit unter Open Source Lizenzen, als auch eine möglichst hohe Plattformunabhängigkeit. Außerdem musste gewährleistet werden, dass die Laufzeiten und der jeweilige Ressourcenverbrauch sehr niedrig sein würden.

2.5.1 „Simple DirectMedia Layer“ (SDL) 1.3

Abb. 8: SDL Logo

SDL existiert seit ungefähr 1998 und wurde hauptsächlich von Sam Lantinga entwickelt. Dies geschah während dessen Arbeit bei Loki Software, welche Spiele auf verschiedene Systeme portierten.[13] Diese Bibliothek wurde unter der LGPL (GNU Lesser General Public License) als Open Source Produkt für die Öffentlichkeit freigegeben. Dadurch erreichte sie Popularität und freiwillige Mitentwickler. Kommerzielle Produkte, wie Doom3, Quake4, Neverwinter Nights, und Penumbra [14], halfen bei der Vergrößerung des Bekanntheitsgrades. SDL dient als Schicht zwischen der Anwendung und dem System und kümmert sich um die Anbindung an die passenden Subsysteme, wie Audio, Video, Netzwerk, Zeitgeber und Eingabe per Tastatur, Maus, Gamepad, Joystick. Es existieren unter anderem Erweiterungen zur Benutzung von Bildern in verschiedenen Formaten, des Audiomixers, komplexerer Netzwerksupport und Textdarstellung. Die aktuell als stabil gekennzeichnete, jedoch etwas in die Jahre gekommene Version, ist die 1.2. Die Nachfolgeversion 1.3 ist bereits so gut, wie fertig und enthält einiges an Veränderungen, wie zum Beispiel die Umstellung auf die zlib-Li-

zenz, volle 3D Hardwarebeschleunigung und Unterstützung von OpenGL3, mehreren Fenstern, Monitoren und Mäusen, weswegen ich mich auch für diese Arbeit für diese Version entschieden hatte. Allerdings spielt diese Bibliothek nur bei der Nutzung der GUI eine Rolle.

2.5.2 CMake

Abb. 9: CMake Logo

CMake wurde von Bill Hoffman bei der Firma Kitware angefangen zu entwickeln und ist dank Verfügbarkeit unter der „New BSD License“ und Mitarbeit der Gemeinschaft seither im Wachstum. Es handelt sich hierbei um ein System, das bei der Erstellung von Programmen aus Quellcode behilflich ist. Dabei generiert es aus gegebenen Konfigurationsdateien unter anderem Makefiles. Diese können anschließend von dem Programm „make“ verarbeitet werden. Weiterhin ist es aber auch möglich Projektdateien für zum Beispiel Visual Studio und Eclipse zu erstellen. Die Konfigurationsdateien werden dabei größtenteils plattformunabhängig gehalten, was der große Vorteil dieses Systems ist. Darüber hinaus unterstützt CMake unter anderem die Kompilierung für andere Systeme und die Erstellung von Paketen aus erstellten Anwendungen als gepackte Archive und Installer. Ebenfalls ist es möglich, Bibliotheken zu erstellen. Der Hauptgrund zur Nutzung dieses Build-Systems war die Plattformunabhängigkeit, sowie die einfache Handhabung.

2.5.3 C++

C++ ist eine höhere Programmiersprache. Angefangen hat sie 1979 als C mit Klassen, entwickelt von Bjarne Stroustrup bei AT&T Bell Laboratories.[15] Einflüsse kamen vor allem von C, Simula, Ada, ALGOL 68, CLU und ML. Inzwischen kann C++ verschiedene Paradigmen, und zwar funktional, generisch, impera-

tiv, objektorientiert, prozedural und strukturiert. Die Typisierung ist statisch, wobei auch dynamisch möglich ist, sowie explizit und stark. Für die meisten Systeme sind Implementierungen vorhanden. Die Sprache wurde 1998, 2003 und 2011 unter ISO/IEC standardisiert. Selber beeinflusst C++ verschiedene Sprachen, wie C#, Java, PHP, Perl, D, Go und Vala.[16]

Seit Jahren ist C++ weltweit eine der meist genutzten Sprachen, sowohl für kleine, als auch für große Anwendungen. Diese Sprache ist im Prinzip das Schweizer Taschenmesser unter den Programmiersprachen. Dank der Tatsache, dass die Simulation sowohl auf normalen PCs, als auch auf den Nao-Robotern laufen sowie performant sein sollte, war die Entscheidung dafür schnell gefallen.

Der 12. August 2011 war der Tag, an dem C++ in die dritte Runde ging.[17] C++0x wurde seit Jahren diskutiert und sollte eigentlich bis Ende 2009 herausgebracht sein. Mit etwas Verzögerung ist es nun gelungen, C++ mit C++11 zum neuen internationalen Standard zu heben. Damit gehen seit 1998 neue Features einher, die teilweise schon im Technical Report 1 (TR1) oder in der Boost-Bibliothek inkludiert waren. Laut einer Übersicht von Scott Meyers[18] bzw. von Apache[19] stellt der GCC(siehe Seite 18), im Gegensatz zu seinen Kontrahenten, bereits die meisten neuen Funktionen bereit.

2.5.4 Lua

Lua wurde 1993 von Roberto Ierusalimschy, Luiz Henrique de Figueiredo und Waldemar Celes an der päpstlich katholischen Universität von Rio de Janeiro entworfen.[20] Es handelt sich hierbei um eine Programmiersprache, die als Skriptsprache entwickelt wurde. Sie hat sich im Laufe der Zeit als robust und schnell herausgestellt. Durch ihre Verfügbarkeit unter der MIT Lizenz, die geringe Größe und hohe Portabilität, sowie ihre hohe Ausdruckskraft, konnte sie sich in vielen Bereichen, wie Industrieapplikationen und Spielen, einnisten und ein hohes Ansehen erlangen. Dies führte dann auch zur Nutzung in diesem Projekt.

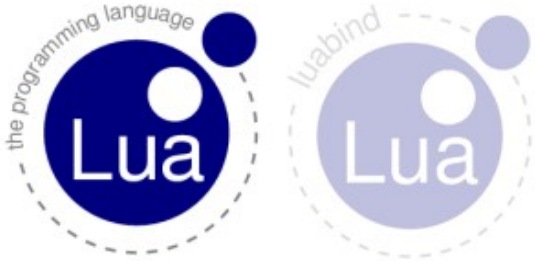

Abb. 10: Logos von Lua und LuaBind

2.5.5 LuaBind

LuaBind wurde 2003 von Daniel Wallin und Arvid Norberg von der Firma „Rasterbar Software“ erstellt.[21] Es handelt sich hierbei um eine Bibliothek, die mittels Template Metaprogrammierung gebaut wurde. Mit ihrer Hilfe können Funktionen und Klassen von C++ zu Lua gereicht und Klassen selber in Lua geschrieben werden. Durch die Nutzung von C++ musste ein Weg gefunden werden, die entsprechenden Funktionalitäten an Lua weiter zu reichen. Dies war der Entscheidende Grund für diese Bibliothek.

2.5.6 Weitere Komponenten

Die weiteren genutzten Komponenten sind allgemeinerer Natur und wurden hauptsächlich aus privater Zuneigung ausgewählt.

Abb. 11: Logos von GCC, Debian, Kdevelop und Mercurial

2.5.6.1 GCC

Die „GNU Compiler Collection“ (GCC) wurde von Richard Stallman 1985 als „GNU C Compiler“ gestartet.[22] Mit der Zeit kamen Unterstützungen für weitere Sprachen, wie C++, Ada, Java und Objectiv-C, womit der Name auf „GNU

Compiler Collection“ umgeändert wurde. Ebenso, wie inzwischen etliche Sprachen unterstützt werden, läuft die GCC auf vielen verschiedenen Plattformen und Architekturen. Das Projekt steht unter der freien „GNU General Public License 3“ (GPL3). Auf Nicht-Windows-Systemen ist die GCC quasi der Standard. Ein weiterer Grund für dessen Nutzung ist die Tatsache, dass bereits die meisten Funktionen aus dem neuen C++11 Standard bereits enthalten sind. [23]

2.5.6.2 Debian GNU/Linux

Debian GNU/Linux ist ein freies Betriebssystem auf Basis von GNU und dem Linux-Kernel. Erstellt wurde es 1993 von Ian Murdock.[24] Über die Jahre wuchs das Projekt und zählt heute mit zu den größten Linux-Distributionen mit sehr vielen Nutzern und erhältlicher Software. Das Paketmanagement ist unter anderem neben der großen Anzahl an frei verfügbaren Paketen eines der Vorteile von Debian. Es ermöglicht eine einfache Installation und Wartung von Softwarepaketen von Anwendungen aller Art, als auch von Entwicklungsbibliotheken und -werkzeugen.

2.5.6.3 KDevelop

KDevelop ist eine grafische Entwicklungsumgebung (IDE, engl. Integrated Development Environment), die seit 1998 existiert.[25] Sie setzt auf die Qt-Bibliothek auf. Die aktuelle Version ist dabei vor allem für C++- und PHP-Entwickler und bietet unter anderem Support für den neuen C++11-Standard. Weitere Vorteile sind unter anderem Syntaxeinfärbung, automatische Codevervollständigung und -einrückung, Integration von Vi, CMake, Debugger und Versionskontrollsystemen.

2.5.6.4 Mercurial

Bei Mercurial handelt es sich um ein freies verteiltes Versionskontrollsystem (VCS, engl. Version Control System). Die Arbeiten daran begannen 2005 von Matt Mackall als Alternative für das damalige für den Linux-Kernel genutzte

VCS, wofür jedoch später Git eingesetzt wurde.[26] Der Umstieg von SVN ist sehr leicht. Die Befehle ähneln sich. Es ist im Prinzip nur ein weiterer Schritt nötig. Zum Beispiel packt ein „commit“ Daten in das lokale Archiv. So können mehrere Veränderungen lokal gemacht werden. Um die Änderungen auf einen externen Server zu bringen, nutzt man „push“. So etwas ist nicht nur im Team sehr nützlich, sondern fördert auch das Experimentieren im alleinigen lokalen Bereich. Aktuell ist Mercurial in der Version 1.9.2 verfügbar und findet bei vielen, vor allem freien Projekten, wie Mozilla, Netbeans und OpenOffice.org, Anwendung.

2.6 Benchmarks

Benchmarks bei Programmiersprachen sind immer so eine Sache. Sie sind wie Werkzeuge. Mit unterschiedlichen Werkzeugen kann man unterschiedliche Dinge tun. Manche besser, andere schlechter. Aber an sich ist jedes Werkzeug auf etwas spezialisiert. Bei den Programmiersprachen ist es genauso. Dies ermöglicht das Schreiben von spezifischem Code extra für bzw. in jener Sprache. Entsprechend kann nun die Implementation von einem Algorithmus in verschiedenen Sprachen oder aber unterschiedliche Algorithmen zur Lösung eines Problems in diversen Sprachen verglichen werden. Üblicherweise werden dabei 3 Größen gemessen:

- Die Laufzeit des Programms,
- Der Speicherverbrauch während der Laufzeit und
- Die Größe des geschriebenen Quellcodes.

2.6.1 „The Computer Language Benchmarks Game“

In Tabelle 3 ist eine Auswahl aus solch einem Vergleich aus dem „The Computer Language Benchmarks Game“[27] aufgeführt. Die Benchmarks wurden auf einem Intel Q6600 Einkern 32Bit Ubuntu Linux System ausgeführt. Weiter stehen noch Tests zu Einkern 64Bit, sowie 4Kern 32- und 64Bit jedes Mal mit Intel-Prozessoren auf Ubuntu-Linux zur Verfügung. Die Werte sind jeweils

das gewichtete geometrische Mittel aus der besten Messung des gesamten Benchmarks, auch der hier nicht aufgeführten Werte, durch die beste Messung der Implementation in der jeweiligen Sprache. Die Spalten entsprechen folgenden Vergleichen:

1. Laufzeit in Sekunden
2. Speicherverbrauch während der Laufzeit in Kilobyte
3. Größe des minimal mit GZip gepackten Quellcodes in Byte
4. Diese Spalte ist eine Zusammenfassung der ersten drei und zeigt mit Spalte 5 die Gesamtplatzierung innerhalb der Tests an.

	Time (secs)	Memory (KB)	Code (B)	Time + Memory + Code	
C++ GNU g++	1.21	21.05	2.97	4.23	3
C GNU gcc	1.29	14.43	2.99	3.81	2
Java 7 averaged	1.53	483.45	2.62	12.48	9
Pascal Free Pascal	2.56	1.50	2.07	1.99	1
Haskell GHC	2.86	17.50	2.74	5.16	4
C# Mono	3.28	65.84	2.25	7.86	5
JavaScript V8	8.19	57.15	1.33	8.54	6
Lua	21.15	26.78	1.37	9.19	7
Python 3	29.20	71.37	1.30	13.94	11
PHP	29.68	41.73	1.66	12.70	10
Perl	30.04	27.51	1.20	9.96	8
C CINT	311.19	65.33	1.53	31.46	12

Tabelle 3: Vergleich von Laufzeit[28], Speicherverbrauch[29] und Codegröße[30] von ausgewählten Programmiersprachen [31]

Die für die Programmierung des Projektes gewählte Sprache C++ liegt insgesamt auf dem 3. Platz. Vor allem im Bereich der geringen Laufzeit konnte sie punkten. Aber auch beim Ressourcenverbrauch liegt sie im vorderen Feld. Schlechter sieht es jedoch bei der Menge des geschriebenen Quellcodes aus. Dort befindet sie sich mit auf den letzten Plätzen. Lua hingegen ist insgesamt im mittleren Feld platziert. Dabei dreht sich die Einzelplatzierung im Vergleich

zu C++ um. Eine geringe Größe an Quellcode und ein Speicherverbrauch, der ähnlich dem von C++ ist, zeigen die Vorteile. Dies geht allerdings auf Kosten der Laufzeit. Als Alternative wäre hier sicherlich noch JavaScript V8 anzusehen. Sie liegt in einem ähnlichen Bereich wie Lua und hat im Vergleich zu Lua eine halbierte Laufzeit und einen doppelten Ressourcenverbrauch.

2.6.2 „Game Scripting Languages“

Die beiden Grafiken in den Abbildungen 12 und 13 sind aus „Game Scripting Languages“[32] von Lewis Van Winkle. Hierbei handelt es sich um Vergleiche zwischen den 6 Skriptsprachen AngelScript, GameMonkey, Lua, Pawn, Squirrel und TinyScheme.

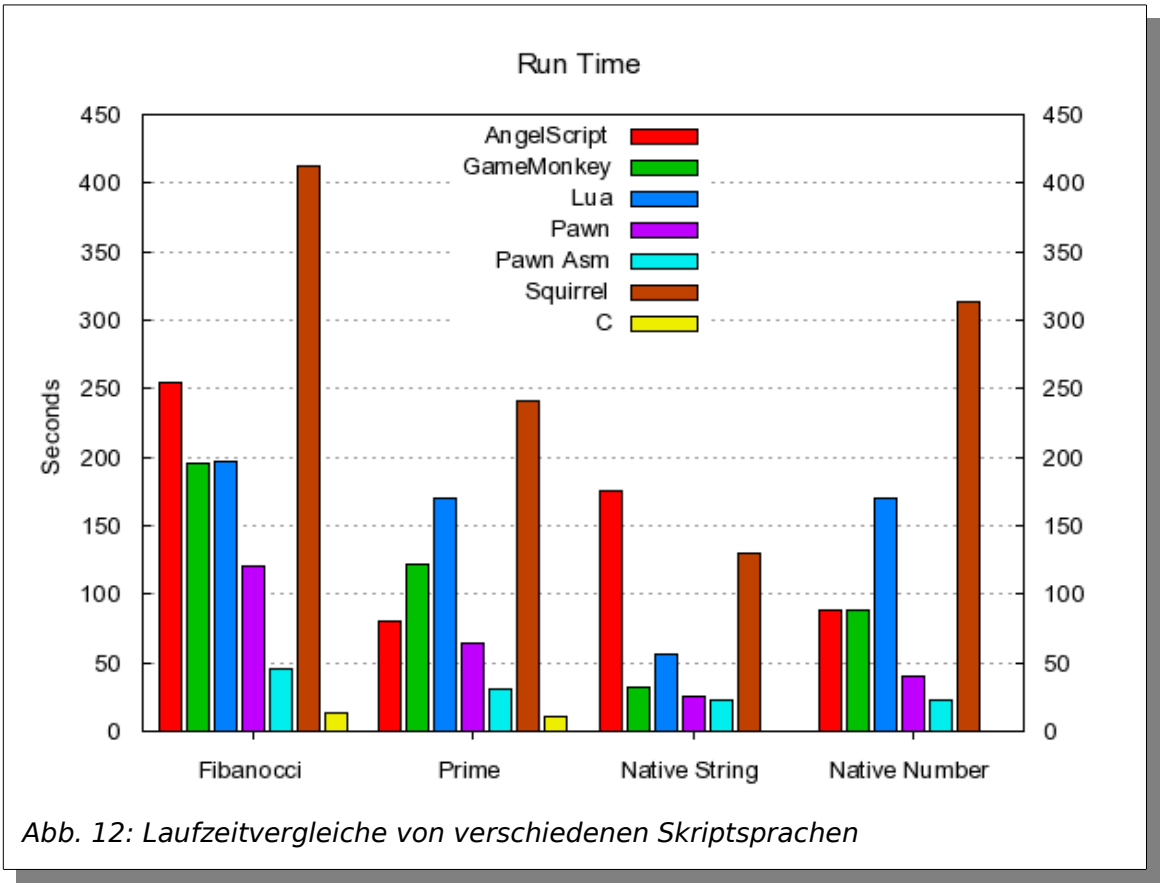

Abb. 12: Laufzeitvergleiche von verschiedenen Skriptsprachen

Abbildung 12 zeigt Vergleiche zur Laufzeit von Programmen in Skriptsprachen zu 4 Problemstellungen. Zusätzlich wurde die Laufzeit eines entsprechenden C-Programms mit angegeben. Der Quellcode der Skriptsprachen wurde vor dem Test zu Bytecode kompiliert. Dadurch entfällt die Zeit, die dafür vor dem Start

benötigt würde. Somit ist in dem Test die reine Laufzeit zur Problemlösung aufgeführt. Die Implementation in C ist jeweils am schnellsten. Lua ist im hinteren Mittelfeld zu finden.

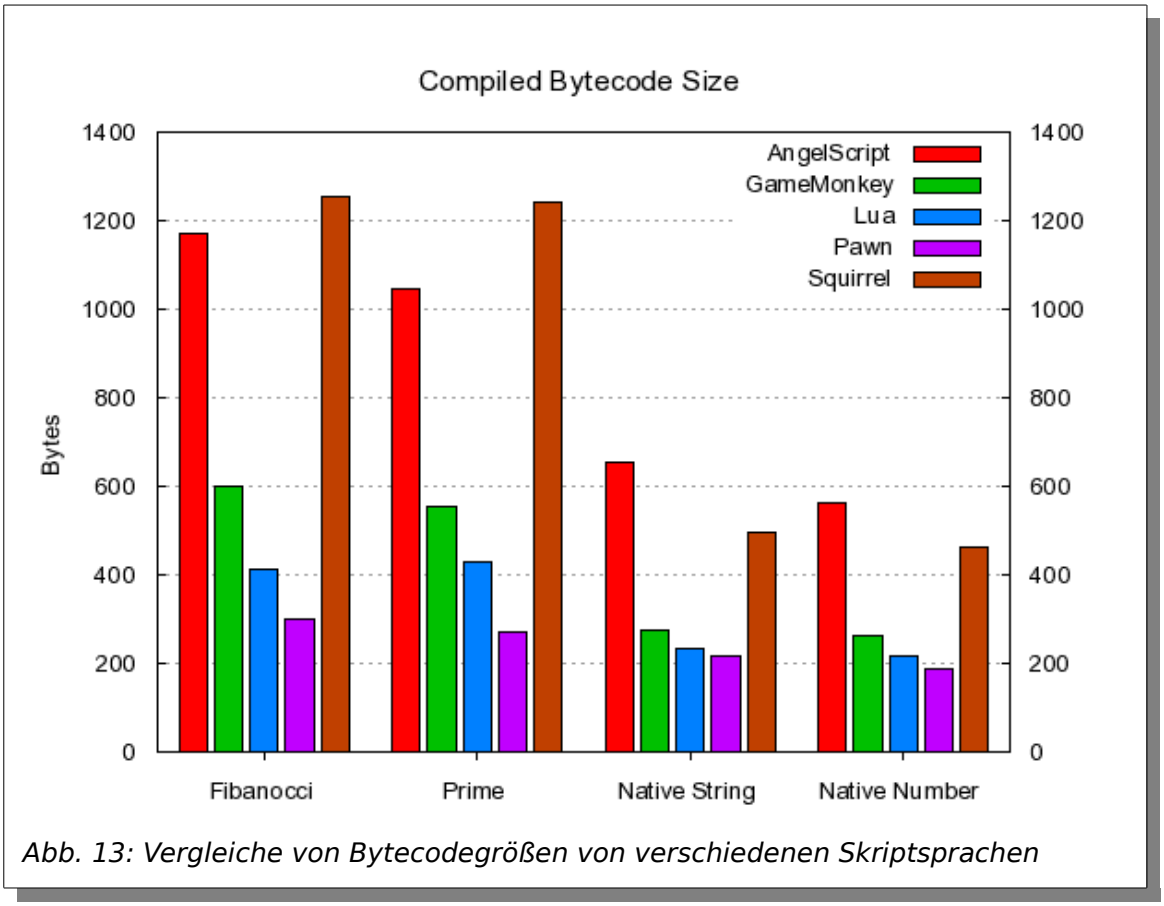

Abb. 13: Vergleiche von Bytecodegrößen von verschiedenen Skriptsprachen

In Abbildung 13 wird die Größe des jeweiligen Programms in Bytecode dargestellt. Lua befindet sich deutlich auf dem zweiten Platz.

Wie bereits im vorherigen Benchmark ist auch hier deutlich zu sehen, dass Lua-Programme nicht mit geringer Laufzeit punkten können, dafür aber mit einer geringen Bytecodegröße und somit auch mit wenig Quellcode auskommen. Aus diesem Benchmark geht Pawn als eindeutiger Sieger heraus. Die Laufzeit beträgt maximal die Hälfte von Lua. Selbst die Bytegröße ist niedriger. Bezogen auf den ersten Benchmark müsste dies mit einem höheren Ressourcenverbrauch erklärbar sein. Leider liegen dazu keine Daten vor. Damit dürften Pawn und JavaScript V8 fast gleich auf liegen. Aufgrund der höheren Popularität von JavaScript, denke ich, dass die Bevorzugung dessen vor Pawn angebracht ist.

Für dieses Projekt wird weiterhin auf Lua gesetzt, da dies ebenfalls gut abschneidet und gefragt ist.

3 Aufbau und Integration der Schnittstellen

Der Hautteil dieser Arbeit beschäftigt sich mit der Integration der Schnittstellen der beiden gewählten Programmiersprachen in den Simulator, sowie dem technischen Vergleich zwischen diesen.

3.1 Aufbau der Schnittstellen

3.1.1 Kapselung der Klassen

3.1.1.1 Theorie

Die künstliche Intelligenz benötigt Zugriff auf verschiedene Werte des Simulators. Die meisten davon sollten aber nur lesenden Zugriff gestatten. Nur die Roboter des eigenen Teams dürfen verändert werden können. Dies aber auch nicht direkt, sondern nur deren Zielpositionen und -rotationen. Es wäre nun möglich, einfach alle Daten der künstlichen Intelligenz bereit zu stellen und sich nicht über die Zugriffskontrolle Gedanken zu machen. Die Implementation dürfte dadurch schneller von statten gehen. Allerdings besteht hier das Problem, dass man schnell auch Werte ändern könnte, die nicht geändert werden sollen. Das kann unabsichtlich passieren, aber auch mutwillig. Da beide Teams von jeweils einer eigenen KI gesteuert werden soll, könnten bei Vollzugriff die künstlichen Intelligenzen jeweils das andere Team manipulieren. Des Weiteren ist es auch für Entwickler, die nur an der künstlichen Intelligenz arbeiten, einfacher, wenn sie genaue Vorgaben haben, was sie nur lesen bzw. auch schreiben können. Ein einfaches Umsetzten der Roboter, oder gar des Balles kann damit unterbunden werden. Es muss also um jede Klasse, die an die KI übergeben werden, bzw. auf die die KI Zugriff erhalten soll, eine Hülle gelegt werden.

3.1.1.2 Praxis: C++

In C++ kann dies umgesetzt werden, indem neue Klassen erstellt werden, die im Konstruktor das zu kapselnde Objekt übergeben bekommen und Methoden

für den Zugriff darauf bereitstellen. Die einzelnen Hüllklassen möchte ich hier kurz betrachten. Dazu als erstes ein Ausschnitt der Hülle für die Klasse „Ball“:

```
class Ball_Connector {
public:     Ball_Connector( Ball* ball);
            Ball_Connector( const Ball_Connector& BC);
            float getCurrentSpeed() const;
            const Vector& getPosition() const;
private:    const Ball_Connector& operator=( const Ball_Connector&);
            Ball* TheBall;
};
```

Tabelle 4: Ausschnitt aus der Klasse Ball_Connector

„Ball“ wird als Zeiger dem Konstruktor übergeben und privat abgespeichert. Damit kann auf den normalen Ball des Simulators zugegriffen werden. Nach außen steht der Ball allerdings nicht zur Verfügung. Da der Ball nicht von der KI verändert werden darf, sind auch sonst keine weiteren Methoden dafür vorhanden. Wichtig sind noch der Kopierkonstruktor und die Methode operator=(). Die erste muss implementiert werden, da die Klasse ein Zeiger-Objekt enthält. Bei Kopieroperationen muss dafür gesorgt werden, dass der Zeiger mit den richtigen Werten gefüllt wird. Die operator=() Methode dient der Beschreibung, was getan werden soll, wenn ein Objekt von diesem Typ eine Zuweisung per = bekommt. In diesem Fall wird die Methode nur deklariert und nicht definiert. Damit wird eine unabsichtliche Nutzung unterbunden. Falls nun jemand versucht per = dem Objekt etwas zuzuweisen, gibt es einen Fehler beim Kompilieren.

Insgesamt folgen die Methoden dem folgenden Schema:

1) elementarer Datentyp Methodenname() const;
2) const abstrakter Datentyp& Methodenname() const;

Das const am Ende der Deklaration bedeutet, dass das, was in der Methode passiert, keine Veränderungen birgt. Da die Methoden nur zur Weiterleitung dienen und somit nichts verändern sollten, ist dieser Zusatz ganz nützlich. Der

Compiler würde passende Fehlermeldungen schmeißen, falls die Regel nicht eingehalten wurde.

Elementare Datentypen werden immer, falls nicht anders angegeben, als Wert zurückgegeben. Ein Zugriff auf das Innere der Methode bzw. die Variable, die zurückgegeben wurde, ist somit nicht möglich. Kaum anders ist es bei abstrakten Datentypen. Allerdings bestehen diese meist intern aus verschiedenen elementaren und auch abstrakten Datentypen. Eine Übergabe als Wert ist möglich, zieht aber entsprechende Kopieroperationen mit sich. Um dies zu vermeiden, ist die Rückgabe als Referenz bzw. Zeiger vorzuziehen. Ein Problem, was dabei jedoch entsteht, ist der Zugriff auf das Objekt. Die internen Daten können, falls es entsprechende Methoden oder gar Variablen gibt, verändert werden. Da dieses Objekt aber eine Referenz bzw. Zeiger ist, handelt es sich um das gleiche Objekt, das in der Methode genutzt wurde. Die Änderungen stecken also auch da mit drin. Um zu verhindern, dass Veränderungen gemacht werden können, wird das Objekt als const Referenz bzw. Zeiger zurückgegeben.Die Bearbeitungen des Objektes ist damit nicht mehr möglich. Es kann nur noch auf Methoden zugegriffen werden, die vor ihrem Rückgabetyp ebenfalls ein const stehen haben.

```
class Robot_Connector {
public:     Robot_Connector( Robot& robot);
            Robot_Connector( const Robot_Connector& RC);
            virtual ~Robot_Connector();
protected:  const Robot_Connector& operator=( const Robot_Connector&);
            Robot& me;
};
```

```
class Robot_Own_Connector : public Robot_Connector {
public:     void shoot( );
            void shoot( Vector direction, float energy);
};
```

Tabelle 5: Ausschnitte aus den Klassen Robot_Connector und Robot_Own_Connector

Für die Roboter sind zwei Hüllklassen nötig, wobei die eine die Ableitung der anderen ist. Dazu stehen Auszüge aus den beiden Klassen in Tabelle 5.

Die erste Hülle ist als Basis für sowohl die gegnerischen, als auch die eigenen Roboter geplant. Dabei stehen nur lesende Methoden zur Verfügung. Dem Konstruktor wird diesmal das einzuhüllende Objekt als Referenz übergeben und intern im Bereich protected abgelegt. Dadurch können auch abgeleitete Klassen darauf zugreifen. Der Destruktor ist virtual. Dies ist nötig, da abgeleitete Klassen geplant sind. Für die eigenen Roboter wird eine abgeleitete Klasse bereit gestellt. Somit enthält sie alles, was die andere Klasse hat und kann diese um Funktionalitäten erweitern. Vor allem sind hierbei Methoden zum Setzen von Werten notwendig. Jene sind wie folgt aufgebaut:

```
void Methodenname( mögliche Parameter);
```

Beim Setzen der Werte mittels dieser Methoden sind keine Fehler zu erwarten, weshalb auf Rückgabewerte verzichtet wurde. Es tritt weiterhin kein const auf, da der Grund des Methodenaufrufs eine Wertänderung ist und dies gegen const spricht. Als Parameter, falls benötigt, sind Wertübergaben vorgesehen. Hier wären für die abstrakten Datentypen ebenfalls Referenzen bzw. Zeiger angebracht, da dadurch Kopieroperationen wegfallen würden. Allerdings ist im allgemeinen der Compiler in der Lage diese Art von Weiterleitung zu erkennen und zu eliminieren.

Anders Schau Knatten hat dazu einen kleinen Test aufgebaut[33], den ich hier in Tabelle 6 etwas zusammengekürzt habe.

Es wird die Anzahl der Konstruktor- (ctors) und Kopierkonstruktor-Aufrufe (copy) bei unterschiedlichen Szenarien (1-6) gemessen. Die Ergebnisse sind mit dem GCC in Version 4.5.2 entstanden. Laut Logik müsste zum Beispiel bei Nummer 5 der Kopierkonstruktor 2 mal aufgerufen werden. Der Compiler kann jedoch diesen Bereich optimieren und beide Extraaufrufe entfernen.

```
C getTemporaryC() {
        return C();  }
C getLocalC() {
        C c;
        return c;     }
C getDelegatedC() {
        return getLocalC();}
vector<C> getVectorOfC() {
        vector<C> v;
        v.push_back(C());
        return v;             }
```

```
int main() {
        C c1;                                   //("1: ");
        C c2(c1);                               //("2: ");
        C c4 = getTemporaryC();         //("3: ");
        C c5 = getLocalC();                     //("4: ");
        C c6 = getDelegatedC();         //("5: ");
        vector<C> v = getVectorOfC();//("6: ");
}
```

	1:	2:	3:	4:	5:	6:
copy	0	1	0	0	0	1
ctors	1	0	1	1	1	1

Tabelle 6: Zusammengekürzter Test von A. S. Knatten mit Ergebnissen

3.1.1.3 Praxis: Lua

Lua muss mitgeteilt bekommen, welche Klassen existieren und was sie beinhalten. Dies entspricht der Umhüllung der Klassen, die im vorherigen Teil besprochen wurden. Zur Vereinfachung wurde die Bibliothek LuaBind hinzugezogen. Es wäre nun möglich gewesen, die eigentlichen Klassen zu umhüllen. Da aber schon die Kapselklassen aus C++ vorhanden waren, konnte nahezu 1 zu 1 der public Part übergeben werden. Dank der Compileroptimierungen dürfte es dabei auch zu keinen negativen Auswirkungen kommen. Im folgenden Ausschnitt ist ein Teil der Ball-Klassenumhüllung zu sehen:

```
1:  luabind::module(luaState)[
2:          luabind::class_<Ball_Connector>("Ball_Connector")
3:          .def( luabind::constructor<const Ball_Connector&>())
4:          .def( "getCurrentSpeed", &Ball_Connector::getCurrentSpeed)
5:  ];
```

Tabelle 7: Ausschnitt aus der Funktion zum Kapsel des Ball_Connector

Mit der zweiten Zeile wird der Zeichenkette „Ball_Connector“ die Klasse „Ball_Connector“ zugewiesen. Durch die folgenden Zeilen werden die Methoden angebunden. Als erstes kommt der Kopierkonstruktor in Zeile 2. Der eigentliche

Konstruktor ist ausgelassen wurden, da er nicht benötigt wird. Es soll mittels Lua keine eigenständige Ball-Hülle existieren. Anschließend stehen weitere Methoden, wobei dabei wieder eine Zeichenkette als Name der Methode innerhalb Luas mit angegeben werden muss. Die Zeichenketten müssen nicht mit dem Namen der Klasse bzw. der Methode übereinstimmen. Sie können frei gewählt werden. Zur einfachen Identifizierung wurden aber die gleichen Namen beibehalten. Der Typ des Rückgabewertes muss nicht mit angegeben werden. Dieser wird automatisch ermittelt.

Für die beiden Roboter-Klassen gilt das gleiche:

```
1: luabind::module(luaState)[
2:         luabind::class_<Robot_Connector>("Robot_Connector")
3:         .def(luabind::constructor<const Robot_Connector&>())
4: ];
```

```
1: luabind::module(luaState)[
2:         luabind::class_<Robot_Own_Connector,
                                Robot_Connector>("Robot_Own_Connector")
3:         .def(luabind::constructor<const Robot_Own_Connector&>())
4:         .def("shoot", (void(Robot_Own_Connector::*)())
                        &Robot_Own_Connector::shoot)
5:         .def("shoot", (void(Robot_Own_Connector::*)(Vector, float))
                        &Robot_Own_Connector::shoot)
6: ];
```

Tabelle 8: Ausschnitte aus den Kapselungen für Lua von Robot_Connector und Robot_Own_Connector

Sie enthalten ebenfalls wieder nicht den normalen Konstruktor, weil keine eigenen Hüll-Objekte erstellt werden sollen. Mittels der Kopierkonstruktoren können die Hüllen der Spieler übernommen werden. In der abgeleiteten Klasse muss, damit auch die geerbten Methoden genutzt werden können, die Basisklasse, wie in Zeile 2 zu sehen, mit angegeben werden. Die shoot()-Methode existiert zweimal in der Klasse. Sie ist überladen. Obwohl bei der Beschreibung der Hülle normalerweise sowohl Rückgabetyp als auch Parameter nicht mit angegeben

werden, da sie automatisch bezogen werden, muss hier der Typ der Methode mit angegeben werden.

3.1.1.4 Praxis: Probleme mit Lua

Im Prinzip ist die Kapselung für Lua und C++ recht simpel. Der zu schreibende Quelltext hält sich soweit gleich. Würde man dies jetzt schon versuchen zu verwenden, würde man feststellen, dass in Lua Methoden, die abstrakte Datentypen zurückgeben, nicht funktionieren bzw. man nicht auf diese Rückgabewerte zugreifen kann. Dies liegt daran, dass Lua diese Klassen noch nicht kennt. Sie müssen also ebenfalls noch bekannt gemacht werden. Das geschieht analog zu den bereits angelegten Hüll-Klassen. In C++ tritt dieses Problem nicht auf, da die bereits vorhandenen Klassen weiter genutzt werden können.

Ein weiteres Problem für Lua sind die Zeiger. Im Lua-Script-Code kann nur mit normalen Werten und nicht mit Zeigern gearbeitet werden. Bei der Übergabe sollten diese demnach nicht benutzt werden. In Container, wie Listen, Vektoren oder Arrays, werden Objekte mit abstrakten Datentypen jedoch meist mittels deren Zeiger gespeichert. Für die Übergabe an Lua muss man sich dabei etwas anderes einfallen lassen.

LuaBind nutzt Templates. An sich ist dies sehr gut, denn Templates erlauben Meta-Programmierung. Das heißt, dass Code, der so geschrieben wurde, zu großen Teilen bereits während des Compiliervorganges umgearbeitet und angepasst wird. Dabei können auch komplexe Berechnungen stattfinden. Dadurch, dass das alles während des Kompilierens geschieht, entfallen jene Schritte zur Laufzeit. Die Anwendung läuft somit schneller. Genau hier liegt jedoch auch ein Nachteil. Das Compilieren dauert dadurch länger. Als Lösung wurde der LuaBind-Code in einer Datei gehalten. Ist der Code einmal zu Objektcode kompiliert wurden, kann bei Änderungen des Programms an anderen Stellen der Objektcode zur Linkzeit einfach hinzugefügt und genutzt werden.

3.1.2 Die Zugriffsklasse

3.1.2.1 Theorie

Von der künstlichen Intelligenz aus soll es genau ein Objekt geben, über das man die nötigen Informationen über die Objekte des Spielfelds bekommt und seine Spieler Befehle erteilen kann. Durch die Benutzung von nur einem Objekt bleibt die Übersicht über die durch die KI möglichen Zugriffe relativ hoch. Durch die Verwendung der Objektorientierung reicht es, auf die bereits erstellen gekapselten Klassen zurückzugreifen. Die Klasse stellt zur künstlichen Intelligenz hin die Ball-Hülle, sowie zwei Container mit den Hüllen von einmal den Spielern des eigenen, sowie einmal den Spielern des anderen Teams zur Verfügung.

3.1.2.2 Praxis: C++

Nach außen sieht die Klasse in C++ wie folgt aus:

```
1:  class Robot_Team_Connector {
2:  public:
3:          Robot_Team_Connector( std::list< Robot* >& own_team,
                    std::list< Robot* >& other_team, Ball* ball);
4:          Robot_Team_Connector( const Robot_Team_Connector& RTC);
5:          std::vector< Robot_Own_Connector* >& getOwnRobotTeam();
6:          std::vector< Robot_Connector* >& getOtherRobotTeam();
7:          Ball_Connector& getBall();
8:          Robot_Own_Connector& getOneOwnRobotTeam(
                    short unsigned int i);
9:          unsigned short int getNumberOfOwnRobotTeam();
10:         Robot_Connector& getOneOtherRobotTeam( short unsigned int i);
11:         unsigned short int getNumberOfOtherRobotTeam();
12: };
```

Tabelle 9: Ausschnitt aus der Klasse Robot_Team_Connector

Der Konstruktor nimmt Listen mit den eigenen Spielern und Gegenspielern, sowie den Ball und speichert diese als entsprechende Hüllobjekte intern ab. Von der KI aus kann nach den Spielern des eigenen bzw. denen des anderen Teams gefragt werden. Dabei kann man entweder die gesamte Gruppe erhalten oder nur einen bestimmten Spieler. Des Weiteren ist der Zugriff auf den Ball vorhanden.

3.1.2.3 Praxis: Lua

Für Lua wird wieder mittels LuaBind gearbeitet. Als Grundlage dient wieder die entsprechende C++-Klasse. Hierbei tritt allerdings das bereits oben erwähnte Problem mit den Zeigern auf. Die Teams werden als Vektoren von Zeigern zurückgegeben. Dies kann nicht in Lua verwendet werden. Durch den Zugriff auf die einzelnen Spieler dürfte das Erfragen des gesamten Teams durch Iteration nicht so kompliziert sein. Die beiden Methoden wurden deswegen nicht mit auf anderem Wege eingebaut.

3.1.3 Klassen der künstlichen Intelligenz

Die Klasse der künstlichen Intelligenz muss an sich nur mindestens eine Methode bereitstellen, die das Zugriffsobjekt entgegen nimmt und Aktionen auf die eigenen Spieler anwendet. Zur besseren Übersicht wurde dies in zwei getrennten Methoden, init() und step(), aufgeteilt. Es wird auch noch eine weitere Methode erwartet, die den frei wählbaren Namen der KI als Zeichenkette zurückgibt. Diese Kombination steht als Basisklasse KI zur Verfügung. Sie sollte nicht verändert werden. Eigene künstliche Intelligenzen müssen bei Benutzung der C++-Version von dieser Klasse abgeleitet sein. Als Beispiele sind die Klassen KI_Blue und KI_Pink vorhanden. Die Lua-Variante setzt ebenfalls auf die Basisklasse KI auf und stellt dadurch die gleichen Funktionalitäten zur Verfügung. Sie sollte nicht verändert werden, da durch sie auf das Lua-Script zugegriffen wird. Das Lua-Script dient der Definition der beiden Methoden step() und getName(). Eine Beispiel-Lua-Datei ist vorhanden.

3.2 Die Integration

Die künstliche Intelligenz muss an einem bestimmten Punkt angreifen. Ausgegangen wird dabei von Abbildung 6. Die Simulator-Klasse stellt die step()-Methode bereit, die alle nötigen Berechnungen ausführt, um zum nächsten Zeitschritt zu gelangen. Im vorherigen Augenblick muss also, wenn gewünscht, die künstliche Intelligenz auf die Spieler angewandt werden. Die KI soll an sich vom Simulator unabhängig sein. Die Zugriffsklasse benötigt den Ball und die Spieler. Dies kann vom Simulator erfragt werden.

3.2.1 Das Bindeglied

Eine weitere Klasse zur Verbindung der künstlichen Intelligenz und des Simulators ist nötig. Sie ist nicht vom Simulator abhängig, übernimmt aber per Konstruktor die wichtigen Objekte und ruft mittels ihrer step()-Methode die step()-Methode der KI- bzw. der abgeleiteten Klassen auf.

```
KI_Connector( std::array<std::string,2> library,
        std::list<Robot*>& team_own, std::list<Robot*>& team_enemy,
        Ball* ball);
```

Das erste Argument stellt den Namen der zu ladenden KI dar. Es gibt dabei drei verschiedene Varianten:

1. Der Name der Lua-Script-Datei
2. Der Name der Klasse, die von der KI-Klasse abgeleitet und im KI_Chooser registriert ist
3. Der Name der Bibliothek, die die von der KI-Klasse abgeleitete Klasse enthält, und der Name der Methode, die einen Zeiger auf ein neues Objekt der Klasse zurückgibt

Die weiteren Argumente sollten mit den Daten aus dem Simulator befüllt werden. Im Konstruktor wird unter anderem die Zugriffsklasse initialisiert und die

eigentlich KI mittels KI_Chooser geladen und anschließend mit der Zugriffsklasse initialisiert.

3.2.2 Das Laden der KI

1:	class KI_Chooser {
2:	public: KI_Chooser();
3:	KI* getKIObject(std::array<std::string,2>);
4:	private: std::map< std::string, KI_t> KIMap;
5:	};
KIMap.insert(std::pair< std::string, KI_t>("ki_blue", CTR::construct<KI_Blue>));	

Tabelle 10: Ausschnitt aus der Klasse KI_Chooser und Funktion zum Einfügen von Paaren in die KIMap

Das Herzstück bei der Bereitstellung der KI ist der KI_Chooser. In dieser Klasse existiert eine Map, die Paare aus einer Zeichenkette und einem Zeiger auf eine Funktion zur Erstellung eines KI-Objektes beinhalten kann. Der Konstruktor füllt diese Map. Ein Beispiel dazu ist in der zweiten Tabellenzeile zu sehen. Klassen, die von der KI-Klasse abgeleitet sind, und entweder in den Simulator mit hinein kompiliert oder dynamisch gelinkt werden sollen, müssen hier angegeben werden. Nur so kann die Methode getKIObject() durch Übergabe einer Zeichenkette einen passenden Zeiger auf ein neues Objekt zurückgeben. Die Methode kann aber auch künstliche Intelligenz auf andere Wege laden. Zum einen ist es möglich durch Übergabe eines Dateinamen, unter anderem mit Pfadangabe, in Form einer Zeichenkette, die auf „.lua“ endet, eine Lua-Script-Datei zu laden und zu nutzen. Zum anderen kann eine Bibliothek dynamisch zur Laufzeit hinzugeladen werden, woraus ein entsprechend passendes, von der KI-Klasse abgeleitetes Objekt bezogen wird. Dazu muss zum einen der Name der zu ladenden Bibliothek, gegebenenfalls mit Pfad, und der Name der Funktion, die als Rückgabe das Objekt hat, als Zeichenketten übergeben werden. Für die Funktion steht in dem ki.h-Header die Methodendeklaration createKI() zur Verfügung, die in der Ableitung implementiert werden kann.

3.2.3 Erleichterungen durch CMake

CMake als Buildsystem einzusetzen, hat einige Vorteile gebracht. Die Quellcodedateien, werden jeweils einzeln zu Objektcode kompiliert. Falls dabei ein Fehler auftritt, kann dieser behoben werden. Beim nächsten Kompiliervorgang werden die bereits vorkompilierten Dateien, falls sich an deren Quellcode nichts geändert hat, übersprungen und erneut mit der nun hoffentlich nicht mehr fehlerhaften Datei fortgefahren. Die verschiedenen Objektcodedateien werden abschließend zu fertigen Programmen bzw. Bibliotheken verlinkt. Ebenso wirken sich Änderungen auf einzelne Dateien beim Kompilieren nur auf die davon abhängigen Dateien aus. Die Restlichen werden übersprungen. Die Zeit für Kompilierungen, vor allem häufiges Neukompilieren nach Änderungen, wird dadurch relativ gering gehalten.

Die Erstellung und Nutzung von dynamischen Bibliotheken funktioniert in CMake recht einfach. Der Befehl zur Erstellung der dynamischen Bibliothek des Simulators sieht wie folgt aus:

```
add_library( Simulator SHARED ${SOURCE_FILES})
```

Der erste Parameter ist der Name, wie die Bibliothek heißen soll. Unter diesem Namen kann sie im gesamten Projekt genutzt werden. Der zweite Parameter gibt an, ob es sich um eine statische oder dynamische Bibliothek oder ein Modul handeln soll. Letztendlich folgt eine Liste mit Quellcode-Dateien, die zu der Bibliothek gehören sollen.Im Beispiel soll eine dynamische Bibliothek mit dem Namen „Simulator“ erstellt werden. Die Liste der Dateien ist hier in einer Variablen versteckt. Eine ausführbare Datei wird ähnlich mittels zum Beispiel

```
add_executable(simulator_ki main_ki.cpp)
```

erstellt. Die Parameter sind der Name der ausführbaren Datei und die Quellcode-Datei mit der main()-Funktion drin. Um weitere Abhängigkeiten aufzulösen, fehlen entweder die anderen nötigen Klassendefinitionen in Form von Quellcodedateien oder diese als Bibliothek. Die zweite Variante wird über folgenden Befehl getätigt:

```
target_link_libraries( simulator_ki ${LINK_LIBS})
```

Als Parameter dienen der Name der ausführbaren Datei, welche die Bibliotheken gelinkt bekommen soll, sowie die nötigen Bibliotheken. In diesem Fall sind diese wieder in einer Variablen versteckt. Dabei handelt es sich um:

```
set( LINK_LIBS Simulator ki_blue ki_pink ki_chooser )
```

Hier werden also 4 Bibliotheken hinzu gelinkt, die an anderer Stelle definiert wurden.

Ein Vorteil der Nutzung von dynamischen Bibliotheken ist das Einsparen von Zeit bei der Kompilierung. Wie CMake schon einzelne Dateien zu Objekten kompiliert und bei Nichtveränderung nicht neu kompiliert, so ist die dynamische Bibliothek im Prinzip eine Zusammenfassung von Dateien und somit eine Vergrößerung des Konzeptes. Zusammengehörige Funktionalitäten, die sich über mehrere Dateien erstrecken, können so in einer Datei zusammengefasst werden. Des weiteren lässt sich solch eine Datei unter Umständen ohne Probleme durch eine neuere Version ersetzten. Das Programm muss dazu nicht im Ganzen neu erstellt werden. Unter Linux-Distributionen hat dieses System einen sehr hohen Stellenwert.

Eine besonders hohe Gewichtung hat dieses System auch bei der Integration der künstlichen Intelligenz in den Simulator. Bei der Erstellung und Anpassung von künstlicher Intelligenz steht häufiges Ausprobieren an. Code wird geschrieben, getestet, verändert und erneut getestet. Bei der Verwendung von C++ muss der Quellcode vor dem Testen immer neu kompiliert werden. Je größer das Projekt ist, desto mehr Zeit wird dafür benötigt. Wird die KI in eine eigene Bibliothek gekapselt, ist sie vom Rest der Anwendung unabhängig. Ebenso ist die Anwendung dann nur noch an die Bibliothek gebunden. Bei Änderungen in der Bibliothek muss das Programm nicht extra neu kompiliert werden. Es ist somit bei Editierungen nur noch die Zeit für die Erstellung der Bibliothek und nicht mehr des gesamten Projektes notwendig. Die beiden von KI abgeleiteten Beispiel-KI-Klassen sind klein gehalten. Dadurch beträgt die Zeit für deren Erstellung nur wenige Sekunden. Komplexe Strukturen und Algorithmen können hier zur Verlängerung der dafür benötigten Zeit führen.

3.3 Ergebnis

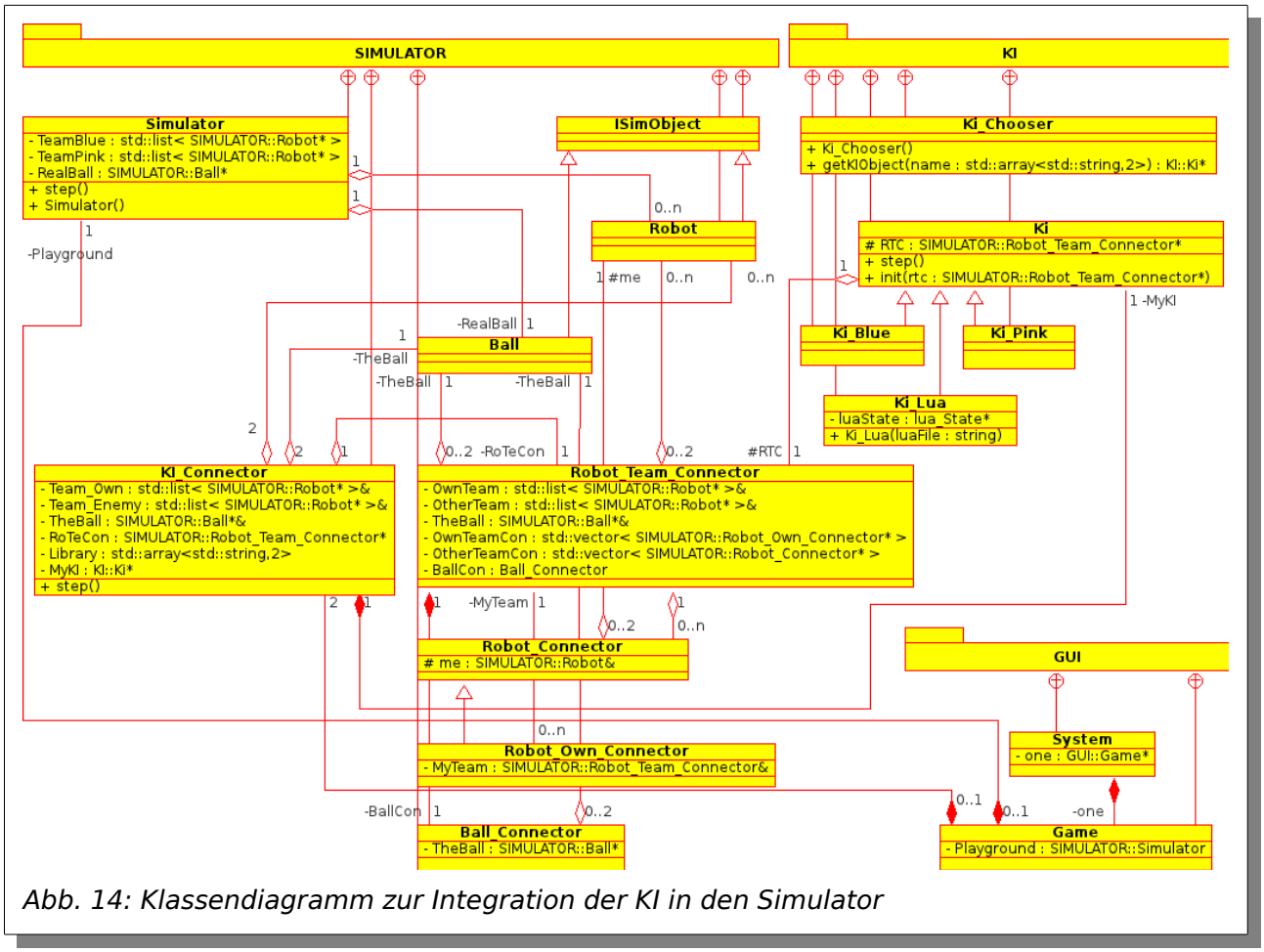

Abb. 14: Klassendiagramm zur Integration der KI in den Simulator

In Abbildung 14 ist die Integration der Klassen, die für die Nutzung der künstlichen Intelligenz nötig sind, in einem Klassendiagramm dargestellt. Die Klassen, die der Umhüllung und Verbindung dienen, sind im Bereich des Simulators integriert. Für die eigentliche KI, inklusive des KI_Chooser und der Beispiel-KI, sowie für die GUI-Komponenten sind eigene Abschnitt erstellt worden. Dies wurde durch die Nutzung entsprechender Ordnerstrukturen versucht zu verdeutlichen.

Die KI wird durch den KI_Connector bereitgestellt und hat Zugriff auf den Robot_Team_Connector, durch den sie Zugang zu den Informationen von dem Ball und den Spielern erhält. Geladen wird sie im KI_Connector mittels KI_Chooser, der die entsprechenden Zeichenketten per Konstruktor übergeben bekommen muss. Des Weiteren müssen bei der Übergabe auch zwei Listen mit den Roboter-Teams, sowie der Ball übermittelt werden. Jenes wird für die Erstellung des Robot_Team_Connector benötigt, welcher beim Aufruf der init()-Methode der KI

übergeben wird. Die KI kann somit auf die Objekte der Simulation zugreifen, ohne dass der Simulator davon etwas mitbekommt.

3.4 Einbau in eine Anwendung

Damit eine Applikation nun die künstliche Intelligenz nutzen kann, muss diese neben dem Simulator angelegt werden.

```
1: Simulator sim;
2: KI_Connector KIC_TeamBlue( "ki/sample.lua",
       sim.getTeamBlue(), sim.getTeamPink(), sim.getBall());
3: KI_Connector KIC_TeamPink( "ki_pink",
       sim.getTeamPink(), sim.getTeamBlue(), sim.getBall());
4: for( int i = 0; i < 3600; ++i ) {
5:     KIC_TeamBlue.step();
6:     KIC_TeamPink.step();
7:     sim.step();
8: }
```

Tabelle 11: Beispiel zum Zusammenspiel KI und Simulator

Als erstes muss der Simulator, in dem der Ball, die Spieler und weitere Objekte erstellt und initialisiert werden, erstellt werden. Direkt anschließend kann man zwei Objekte vom Typ KI_Connector unter Angabe der gewünschten KI als Zeichenkette und der nötigen Daten aus dem Simulator kreieren. Da die künstliche Intelligenz vor dem Simulationsschritt angewandt werden sollte, sind diese entsprechend vorher aufzurufen. Um ein Spiel über einen bestimmten Zeitraum darzustellen, sind die step()-Methoden innerhalb einer Schleife mehrmals hintereinander auszuführen. Auf diesem Weg wäre der Einsatz der KI immer noch etwas fest. Die Änderung der zu benutzenden KI erfordert ein Neukompilieren. Um dem entgegenzuwirken, können zum Beispiel die Zeichenketten als Parameter beim Programmaufruf übergeben werden. Eine weitere Möglichkeit wäre die Abfrage der Namen nach Programmstart. Als Beispiele stehen sowohl die Konsolenanwendungen, als auch die GUI-Programme zur Verfügung.

4 Vergleiche zwischen Lua und C++

In diesem Kapitel soll Quellcode, welcher in beiden Sprachen geschrieben wurde, miteinander verglichen werden. Im ersten Abschnitt wird dabei direkt auf den Quellcode eingegangen, wohingegen es sich im zweiten Abschnitt vor allem um deren Laufzeiten und im dritten Abschnitt um den Ressourcenverbrauch während der Ausführung geht.

4.1 Quellcode

4.1.1 Beispiel 1: ohne Inhalt

C++: ki_empty.cpp	Lua: empty.lua

std::string KI_Empty::getName() {
return "Empty-C++";
}
void KI_Empty::step() {
}
void KI_Empty::init(Robot_Team_Connector* rtc) {
RTC = rtc;
}
extern "C" {
KI* createKI() {
return new KI_Empty();
}
}
	function getName ( )
	return "Empty-Lua"
	end
	function step (RTC)
	end

Tabelle 12: Beispiel 1: ohne Inhalt

Im ersten Beispiel wird die notwendige Grundstruktur verglichen. Der Lua-Code erstreckt sich über 5 Zeilen, wohingegen auf der C++-Seite bereits 13 Zeilen Quellcode stehen. Dabei fehlt hier noch die Klassendeklaration und etwaige include-Befehle. Diese Grundstruktur kann man sich jedoch schnell bereitstellen und damit steht der vollen Konzentration auf die Entwicklung innerhalb der

step()-Methode nichts mehr im Wege. Da Lua sich stark an C anlehnt, sei angemerkt, dass die Erstellung von Methoden sehr ähnlich ist. Man darf auch nicht vergessen, dass für die C++-Klasse die CMake-Konfigurationsdatei angepasst werden muss. In diesem Projekt sind das zwei Zeilen, die hinzugefügt und eine, die angepasst wird. Damit sich die Lua-Datei besser in das Projekt eingliedert, kamen dafür ebenfalls zwei Zeilen in der Konfigurationsdatei hinzu.

4.1.2 Beispiel 2: Aufruf zweier Methoden in der step()-Methode

C++: ki_onefun.cpp	Lua: onefun.lua
std::cout << RTC->getNumberOfOwnRobotTeam() << ":" << RTC->getNumberOfOtherRobotTeam() << std::endl;	io.write(RTC:getNumberOfOwnRobotTeam() , ":", RTC:getNumberOfOtherRobotTeam() , "\n")

Tabelle 13: Beispiel 2: Aufruf zweier Methoden in der step()-Methode

Im Gegensatz zum ersten Beispiel passiert hier im zweiten Beispiel etwas in der step()-Methode. Es werden zwei Methoden aufgerufen, die jeweils die Anzahl der Roboter des eigenen bzw. anderen Teams erfragt und anschließend gemeinsam auf die Standardausgabe ausgibt. Die Anzahl der zu tippenden Zeichen sind nur gering unterschiedlich mit leichtem Vorteil für Lua. Ein Problem, was die Nutzung von Lua etwas erschwert, ergibt sich allerdings daraus, dass die von mir genutzte Entwicklungsumgebung die automatische Vervollständigung in der Lua-Datei nicht unterstützt. Dadurch bin ich gezwungen gewissermaßen alles per Hand zu schreiben und muss notgedrungen, falls ich nicht weiß, wie zum Beispiel die Methodennamen lauten, diese aus der entsprechenden Datei ersuchen. In C++ wird mir dagegen nach dem Tippen der ersten Buchstaben zum Beispiel einer Variablen eine Liste mit möglichen Variablen gezeigt, aus denen ich dann per Mausklick bzw. Pfeiltasten und Entertaste meine gesuchte auswählen kann. Ebenso einfach funktioniert das Auswählen und Einfügen nach Punkt bzw. Pfeil. Dadurch wird das Arbeiten deutlich vereinfacht.

4.1.3 Beispiel 3: Benutzung von For-Schleifen und Variablen

In Beispiel drei geht es um die Benutzung von Schleifen und Variablen. Dazu solle die Positionen aller Roboter ausgegeben werden.

C++: ki_printplaces1.cpp

```
1: for( auto r : RTC->getOwnRobotTeam() ){
2:         std::cout << r->getPosition().toString() << "\n";
3: }
4: for( auto r : RTC->getOtherRobotTeam() ){
5:         std::cout << r->getPosition().toString() << "\n";
6: }
```

C++: ki_printplaces2.cpp

```
1: for( auto i = 0; i < RTC->getNumberOfOwnRobotTeam(); ++i){
2:         std::cout << RTC->getOneOwnRobotTeam(i).getPosition().toString()
                << "\n";
3: }
4: for( auto i = 0; i < RTC->getNumberOfOtherRobotTeam(); ++i){
5:         std::cout << RTC->getOneOtherRobotTeam(i).getPosition().toString()
                << "\n";
6: }
```

Lua: printplaces.lua

```
1: for i = 0, RTC:getNumberOfOwnRobotTeam() - 1 do
2:         io.write( RTC:getOneOwnRobotTeam(i):getPosition():toString(), "\n")
3: end
4: for i = 0, RTC:getNumberOfOtherRobotTeam() - 1 do
5:         io.write( RTC:getOneOtherRobotTeam(i):getPosition():toString(), "\n")
6: end
```

Tabelle 14: Beispiel 3: Benutzung von For-Schleifen und Variablen

In Lua wird der Typ einer Variablen nicht explizit mit angegeben. An jeder beliebigen Stelle im Quellcode kann man durch Zuweisung eines Wertes an ein nicht Schlüsselwort eine neue Variable erstellen. Dabei kann es durch Schreibfehler passieren, dass neue Variablen entstehen anstatt alte zu verändern. Für Varia-

blen in C++ hingegen muss der Typ bei ihrer Deklaration angegeben werden. Mit C++11 wurde das Schlüsselwort auto angepasst. Es ist damit nun möglich, den Typ der Variable implizit durch den Compiler erkennen zu lassen. Zum Beispiel wird aus auto in Zeile 1 in ki_printplaces1.cpp Robot_Own_Connector* und aus auto in Zeile 1 in ki_printplaces2.cpp wird int. Die normalen For-Schleifen in C++ (ki_printplaces2.cpp) und Lua (printplaces.lua) sehen sehr ähnlich aus. Als erstes kommt der Ausdruck, welcher vor dem Eintritt in die Schleife ausgeführt wird. Dort stehen meist Initialisierungen und Zuweisungen. Der zweite Ausdruck stellt die Endbedingung da, die nach jedem Durchlauf überprüft wird. Der dritte Ausdruck wird vor jedem außer dem ersten Schleifeneinstieg ausgeführt und wird meist für Incrementierungen bzw. Decrementierungen genutzt. Ist dieser Ausdruck in Lua weggelassen wurden, wird die im ersten Ausdruck stehende Variable incrementiert. Der Quellcode hält sich annähernd die Waage. Durch C++11 erfuhr die For-Schleife eine Neuerung. Diese erlaubt nun den einfachen Umgang mit Containern. In Kombination mit auto kann viel Schreibarbeit eingespart werden. Vor dem Doppelpunkt steht der Variablenname und dessen Typ, welcher die einzelnen Werte des Containers übergeben bekommt. Anschließend folgt das Containerobjekt.

So komfortabel die Benutzung von auto auch ist, führt dies ab und an zu Problemen in meiner Entwicklungsumgebung. Diese ist dann nicht in der Lage den Typ der Variablen zu bestimmen. Entsprechend gelingt nachfolgend die Code Komplettierung nicht. Dann müssen entweder die Methoden per Hand getippt oder auto zum eigentlichen Typ umgewandelt werden.

4.1.4 Beispiel 4: Benutzung von for_each mit Lambda und if

Das vierte Beispiel ist etwas komplexer. Im Prinzip soll die Zielposition des Spielers, der am dichtesten am Ball dran ist, auf die des Balls gesetzt werden. Dazu wird als erstes der Roboter herausgesucht, der am dichtesten am Ball ist. Dies geschieht mit Hilfe einer For-Schleife über alle Spieler und einer If-Abfrage. Anschließend wird dessen Zielposition gesetzt.

C++: ki_nearesttoball1.cpp

```
1: Ball_Connector& b = RTC->getBall();
2: Robot_Own_Connector* r1 = RTC->getOwnRobotTeam()[0];
3: for( Robot_Own_Connector* r : RTC->getOwnRobotTeam()){
4:     if( b.getPosition() - r->getPosition() < b.getPosition() - r1->getPosition()){
5:         r1 = r;
6:     }
7: }
8: r1->setTarget( b.getPosition() - r1->getPosition());
```

C++: ki_nearesttoball2.cpp

```
1: Ball_Connector& b = RTC->getBall();
2: Robot_Own_Connector* r1 = RTC->getOwnRobotTeam()[0];
3: std::for_each( ++(RTC->getOwnRobotTeam().begin()),
   RTC->getOwnRobotTeam().end(), [&b,&r1](Robot_Own_Connector*& r){
4:     if( b.getPosition() - r->getPosition() < b.getPosition() - r1->getPosition()){
5:         r1 = r;
6:     };
7: } );
8: r1->setTarget( b.getPosition() - r1->getPosition());
```

Lua: nearesttoball.lua

```
1: b = RTC:getBall()
2: r0 = RTC:getOneOwnRobotTeam(0)
3: for i = 1, RTC:getNumberOfOwnRobotTeam() - 1 do
4:     if (Vector(0,0) + b:getPosition() - RTC:getOneOwnRobotTeam(i):
       getPosition()) < (Vector(0,0) + b:getPosition() - r0:getPosition()) then
5:         r0 = RTC:getOneOwnRobotTeam(i)
6:     end
7: end
8: robo0:setTarget( Vector(0,0) + b:getPosition() - r0:getPosition())
```

Tabelle 15: Beispiel 4: Benutzung von for_each mit Lambda und if

Von der Zeilenanzahl her sind mit jeweils 8 Zeilen, wenn man die Zeilenumbrüche wegen Überlänge vernachlässigt, alle drei Versionen gleich lang. Die erste C++-Version ist an die kurze For-Schleife in Beispiel 3 angelehnt. Ebenso entspricht die For-Schleife in der Lua-Version der aus Beispiel 3. In ki_nearesttoball2.cpp wird die Funktion for_each aus den Standard Template Library (STL) Algorithmen benutzt. Dadurch kann die mit C++11 neu hinzugekommene anonyme Funktion, die sogenannte Lambda-Funktion, gezeigt werden. Die einfachste anonyme Funktion ist diese:

```
[](){};
```

Man kann sie in drei Bereiche unterteilen, wobei jeder Bereich ein Klammernpaar ist. Die ersten Klammern dienen der Angabe, welche Variablen von außerhalb in der Funktion genutzt werden können. Dabei ist es möglich die Übergabe als Wert bzw. als Referenz zu tätigen. Der zweite Bereich ist funktionstypisch für die Parameter vorgesehen. Zum Abschluss folgt der eigentliche Inhalt der Funktion. Im Beispiel 4 in Datei ki_nearesttoball2.cpp sollen zwei Variablen von außen intern als Referenz genutzt werden. Auch ist die Übergabe eines Parameters notwendig. Der Funktionsrumpf enthält eine If-Kontrollstruktur, die der aus der anderen C++-Version und Lua-Version gleicht.

4.2 Die Testumgebung

Für die folgenden Tests wurde ein ThinkPad Edge 13 von Lenovo genutzt. Er ist mit einem AMD Turion II Neo K685 mit 2 Prozessorkernen zu je 1.8GHz und 6GB RAM ausgestattet. Als Betriebssystem läuft Debian Testing mit einem Kernel der Version 3.0.0 und der Desktopoberfläche KDE4. Der GCC liegt in Version 4.6.1 und Lua in Version 5.1.4 vor. Die Simulationstestprogramme wurden mit der Optimierungsstufe 3 erstellt.

Zur Benutzung der Beispiele stehen drei ausführbare Dateien bereit:

a) simulator_ki

b) sim_ki_dyn

c) simu_ki_all

Die Werte zu Lua (1) und C++ (1) wurden mittels simulator_ki ermittelt. Bei C++ (2) kam sim_ki_dyn zum Einsatz. Diese Datei kann, im Gegensatz zu den beiden anderen Anwendungen, extra Bibliotheken öffnen und KI-Objekte von dort bekommen. Für Lua (2) und C++ (3) wurde simu_ki_all genutzt. Im Unterschied zu simulator_ki sind die meiste Klassen, von denen sie abhängig ist, direkt in simu_ki_all hinein kompiliert. Jede der drei Anwendungen ist intern eine Schleife 100000 mal durchgegangen, in der die step()-Mehtoden des Simulator und der Beispiel-KI aufgerufen wurde.

4.3 Laufzeit

Aus den Benchmarks aus Kapitel 2.6 steht die Erwartung, dass die Laufzeiten der Programme mit Lua langsamer sind. Jeweils 10 einzelne Ergebnisse sind zu einem Durchschnittswert zusammengefasst und in der Tabelle 16 aufgeführt. Die Einzelergebnisse sind in Anhang (1) zu finden. Alle Werte sind in Sekunden.

	Beispiel 1	Beispiel 2	Beispiel 3		Beispiel 4	
Lua (1)	2.414	2.744	10.508		12.615	
Lua (2)	2.020	2.314	9.312		11.160	
C++ (1)	2.277	2.377	5.658	5.640	4.522	4.433
C++ (2)	2.398	2.339	5.638	5.638	4.369	4.527
C++ (3)	1.867	2.011	4.979	4.934	3.856	3.771

Tabelle 16: Durchschnittliche Laufzeiten der Beispiele

4.4 Ressourcenverbrauch

Laut den bereits im vorhergehenden Kapitel erwähnten Benchmarks müsste der Ressourcenverbrauch bei der Nutzung von Lua nur leicht höher sein, als wenn nur C++ genutzt würde. Um dies zu Betrachten, wurde Tabelle 17 erstellt. Die Programme sind dazu wie bei der Untersuchung der Laufzeit gestartet wurden. Die Werte entstammen einem Ressourcenmonitor, der die laufenden Programme überwacht hat. Es ist jeweils ein Paar aus genutztem Arbeitsspeicher und gemeinsamen genutztem Speicher aufgelistet. Alle Werte sind in Kilobyte.

	Beispiel 1	Beispiel 2	Beispiel 3		Beispiel 4	
Lua (1)	484/1744	432/1796	560/1864		448/1808	
Lua (2)	472/1740	420/1792	544/1860		440/1804	
C++ (1)	292/1520	288/1524	284/1528	288/1528	288/1528	284/1528
C++ (2)	284/1528	288/1524	288/1524	288/1524	284/1528	284/1528
C++ (3)	280/1504	280/1504	280/1504	280/1504	280/1504	280/1504

Tabelle 17: Ressourcenverbrauch der Beispiele

4.5 Auswertung

4.5.1 Quellcode schreiben und nutzbar machen

Das Schreiben des Quellcodes ist ein wichtiger Punkt bei der Entwicklung von Anwendungen. In der derzeit sehr schnelllebigen Welt müssen Programme schnell entwickelt und noch schneller angepasst werden können. Die Benutzung eine Skriptsprache, wie zum Beispiel Lua, kann ungemein Zeit für Kompilierungen sparen. Wurde in diesem Projekt eine der Beispiel-KI-Klassen umgeändert, betrug der Kompiliervorgang mindestens 2 Sekunden. Das ist an sich noch verkraftbar. Die Lua-Beispieldatei dagegen war nach dem Ändern und Speichern sofort nutzbar. Die Beispiele waren noch sehr klein und einfach. Blickt man aber auf die Bereitstellung der Klassen für Lua und betrachtet die Zeit, die allein diese Klasse zum erstellen braucht, was ungefähr 12 Sekunden sind, so wird klar, dass zum Beispiel durch die exzessive Nutzung von Templates die Zeit für das Compilieren deutlich steigen kann.

Das eigentliche Schreiben des Quellcodes hält sich bei Sprachen, wie C++ und Lua, durch die Anlehnung von Lua an C, die Waage. Verschiedene Editoren und Entwicklungsumgebungen erleichtern das Arbeiten. Durch Dinge, wie die Angabe des Typs einer Funktion, Methode, Variablen, und Aufteilung Deklarationen und Definitionen, sowie Fallunterscheidungen für verschiedenen Plattformen, kann durchaus mehr Quelltext in C++ als in Lua anfallen. Gerade bei ausführlichem und sauberen Arbeiten kommt einiges zusammen. Dank des Vorhandenseins von Hilfsmitteln kann hierbei aber vieles automatisiert werden.

4.5.2 Laufzeit

Wie bereits in anderen Benchmarks in Kapitel 2.6 zu sehen war, konnte auch hier wieder aufgezeigt werden, dass die Benutzung von Skriptsprachen die Ausführungsgeschwindigkeit der Anwendung negativ beeinträchtigen kann. Simple Abfragen können schon zu, wie in Beispiel 3 dreimal höherer, zusätzlicher Laufzeit führen.

Ob ein Objekt dynamisch aus einer Bibliothek, die extra geladen werden muss, oder aus einer Bibliothek, die bereits dynamisch gelinkt ist, in die Anwendung um den Simulator geladen und benutzt wird, scheint kaum eine Rolle zu spielen. Die Messwerte zu C++ (1) und C++ (2) unterscheiden sich nur marginal. Einen messbaren Unterschied gibt es allerdings bei dem Vergleich von C++ (1) und C++ (3). Bei der Näheren Betrachtung fällt allerdings auf, dass bei allen 4 bzw. 6 Beispielen eine Differenz von ungefähr 0.5 Sekunden existiert. Dieser Vorteil zielt also eher auf das Hineinkompilieren der Abhängigkeiten hin.

Vergleicht man jedoch Lua (1) und Lua (2) miteinander, müsste man annehmen, dass hier ebenfalls nur eine konstante Laufzeitverkürzung durch das Hineinkompilieren möglich ist. Das scheint in den ersten beiden Beispielen auch noch zu stimmen. Die letzten beiden Beispiele zeigen jedoch ein anderes Bild. Die Differenz steigert sich von ungefähr 0.4 Sekunden über 1.2 Sekunden auf ca. 1.45 Sekunden. Es kann also doch auch eine Auswirkung auf die Interaktionen mit den Modulen zu haben.

Auffällig sind die Werte von Beispiel 3 und 4. Lua benötigt für Beispiel 4 mehr Zeit, als für Nummer 3. In C++ ist dies umgedreht. Für die Erklärung liegen drei Faktoren berei: die If-Kontrollstruktur, die Wertbeschaffung und die Wertberechnung. Die If-Kontrollstuktur ist dabei nur unwesentlich beteiligt. Hauptsächlich kommen die anderen beiden Faktoren in Frage. In Lua werden die Werte zusätzlich zu einem Null-Vektor addiert. Dies musste angefügt werden, da ansonsten Fehler entstanden. Berechnungen und Vergleiche scheinen somit in Lua etwas an Laufzeit zu kosten.

4.5.3 Ressourcenverbrauch

Bei der Betrachtung der Werte in Tabelle 17 stellt man fest, dass bei der Benutzung von Lua in beiden Varianten der Speicherverbrauch schwankt. Das liegt daran, dass das nicht nur Speicher für die Variablen, sondern auch für Methoden und weitere Ausdrücke nötig ist. Im Gegensatz dazu sind in der dritten C++-Variante keinerlei und bei den restlichen beiden nur ganz leichte Schwankungen festzustellen. Diese sind somit auf fehlenden Optimierungsmöglichkeiten zwischen den Bibliotheken, die zum Simulator gehören, zurückzuführen. Auch besitzt die dritte Variante den niedrigsten Verbrauch. Vergleicht man die Werte von Lua mit denen von C++, stellt man fest, dass mit Lua ca. 150% bis 200% so viel Arbeitsspeicher und knapp 115% bis 120% so viel gemeinsamer Speicher wie mit C++ genutzt wird.

Innerhalb der Ergebnisse zu Lua lieg die Differenz der verbrauchten Ressourcen bei maximal rund 190 bis 200 Kilobyte zwischen den Beispielen 2 und 3. Da der Unterschied zwischen Beispiel 2 und 4 recht niedrig ist, liegt es nahe, dass der hohe Verbrauch in Beispiel 3 an der Ausgabefunktion io.write liegt. Die restlichen drei Werte liegen relativ nah beieinander. Der Mehrverbrauch im Vergleich zu C++ geht somit hauptsächlich zu Lasten der Bereitstellung der Lua-Schnittstelle.

Eine Besonderheit, die weiterhin ins Auge sticht, ist der sichtlich höhere Wert des Arbeitsspeichers und niedrigere Wert des gemeinsamen Speichers in Beispiel 1 im Vergleich zu Beispiel 2 und 4 von Lua (1) und (2) und C++ (1). Würde es am C++-Unterbau liegen, müssten sich die Werte von C++ (3) in Lua (2) widerspiegeln. Dies ist aber nicht der Fall, da bei C++ (3) die Werte zu allen drei Beispielen konstant ist. Eine passende Erklärung dafür kann ich nicht finden.

5 Fazit

Es sind drei verschiedene Schnittstellen für Module in den Simulator integriert worden. Eine dient zum laden von Modulen, die dynamisch gelinkt wurden. Die andere kann Module aus extra angegebenen Dateien laden. Und die dritte bietet die Möglichkeit zur Scripterstellung mittels Lua an. Dazu wurden die Beispielprogramme angepasst. Zur Analyse des Speicherverbrauchs und der Laufzeit sind 4 Beispiel implementiert worden.

Das Hauptziel, die Bereitstellung der Schnittstellen und das Ermöglichen von austauschbaren Modulen an dem Nao-Simulator „Spielwiese“, ist damit erreicht.

Wie bereits vorher schon bekannt, konnte auch hier aufgezeigt werden, dass durch die Benutzung von Skriptsprachen die Laufzeit und der Ressourcenverbrauch erhöht wird. Für den Simulator heißt das, dass zum schnellen Testen von Ideen am PC die Nutzung des Skriptes von Vorteil ist, da ohne langwieriges Neukompilieren gearbeitet werden kann. Für die Nutzung auf den Robotern ist es allerdings angebracht, die entwickelten Algorithmen in eine entsprechende C++-KI-Klasse zu portieren.

Bevor man eine Skriptsprache einsetzt, sollte man sich über dessen Vor- und Nachteile informieren und im Klaren sein. Die Möglichkeit den Code, den man gescriptet hat, wieder zum Beispiel in C++-Quellcode zu wandeln und etwas fester in das System zu integrieren, wie es hier in dem Simulator möglich ist, wäre eine Option. Dabei fällt jedoch Arbeit für die Umwandlung und Integration des Codes an. Entsprechend sollte man sich dabei für eine in diesem Fall C++-ähnliche Skriptsprache entscheiden.

Besonders Lohnenswert ist die Nutzung von Skriptsprachen bei dynamischer Entwicklung, bei der sich der Quelltext ständig ändern kann. Vor allem im Bereich der Spieleindustrie und im Internet werden diese sehr häufig eingesetzt. Genauso werden aber auch Programmiersprachen wie C++ in den gleichen Sektoren für vor allem Grundlagen genutzt. Beide Sprachformen haben somit

ihre Berechtigung und sollten auch genau so, entsprechend ihrer Qualitäten und Anwendungsgebiete, ausgewählt werden.

Die neuen Sprachfeatures von C++11 sind eine feine Sache und erleichtern das Arbeiten. Dadurch, dass sie aber noch frisch sind, ist deren Benutzung in Editoren teilweise noch etwas umständlich. Die Erstellung von Bibliothek unter Zuhilfenahme von CMake war letztendlich sehr einfach. Die Einarbeitung in Lua gelang durch ihre C-Nähe flüssig. Den Code darin zu schreiben war jedoch etwas mühsam, da der genutzte Editor verschiedene Unterstützungen nicht mitbrachte.

6 Ausblick

An dieser Arbeit gibt es unter anderem Folgende Möglichkeiten der weiteren Bearbeitung:

Die Boost-Bibliothek bietet mit Boost.Extension[34] eine einfache Möglichkeit zur plattformunabhängigen Plugin- und Erweiterungsentwicklung. Die Benutzung von Lua im Simulator-Projekt dürfte auf verschiedenen Plattformen nicht zu Problemen führen. Die Funktion zum Laden von Extrabibliotheken funktioniert jedoch nur unter Linux. Boost.Extension würde dieses Problem beseitigen können.

Im Prinzip wäre es auch möglich, die KI-Module während der Laufzeit auszutauschen. Passende Funktionalitäten sind aber noch nicht eingebaut. Gerade im Bezug auf Skripte, die während der Laufzeit verändert werden können, wäre diese Erweiterung wünschenswert.

Ein netter Zusatz wäre ebenso, wenn, nach der Änderung eines C++-KI-Modulcodes, dieses, ohne Zutun des Anwenders, sich selbst neu kompilieren könnte. Als Beispiel kann hier das Scripting in der HPL3 von Frictional Games[35] angebracht werden.

Aus den Benchmarks aus Kapitel 2.6 geht hervor, dass JavaScript deutlich schneller als Lua sein soll. Gerade bei der Verwendung des Simulators ist die Ausführungsgeschwindigkeit gegenüber des Ressourcenverbrauchs zu bevorzugen. Es wäre also noch zu prüfen, ob diese Skriptsprache hier im Simulator sinnvoller wäre und diese entsprechen vorzuziehen und zu integrieren.

Anhang (1)

empty	onefun	printplaces	nearest
2,41	2,747	10,518	12,449
2,417	2,725	10,454	11,862
2,437	2,827	10,561	13,035
2,395	2,724	10,702	12,783
2,465	2,763	10,611	13,344
2,399	2,704	10,765	12,174
2,374	2,754	10,324	12,705
2,451	2,731	10,406	12,458
2,404	2,695	10,428	12,831
2,391	2,767	10,315	12,513
2,4143	2,7437	10,5084	12,6154

Tabelle 18: Lua (1)

empty	onefun	PrintPI 1	PrintPI 2	Nearest 1	Nearest 2
2,349	2,412	5,641	5,616	4,359	4,411
2,293	2,336	5,594	5,614	4,348	4,493
2,246	2,349	5,585	5,614	4,403	4,549
2,252	2,351	5,528	5,723	4,789	4,462
2,259	2,321	5,535	5,611	4,544	4,803
2,282	2,513	5,567	5,695	4,673	4,286
2,23	2,325	6,331	5,575	4,235	4,715
2,242	2,455	5,552	5,571	4,957	4,341
2,281	2,329	5,681	5,774	4,506	4,208
2,335	2,378	5,565	5,609	4,401	4,066
2,2769	2,3769	5,6579	5,6402	4,5215	4,4334

Tabelle 19: C++ (1)

empty	onefun	printplaces	nearest
2,031	2,331	9,465	10,938
1,996	2,315	9,291	11,442
2,012	2,381	9,188	10,755
2,015	2,263	9,198	11,098
1,987	2,321	9,291	10,61
2,007	2,331	9,386	11,65
2,033	2,266	9,356	11,508
2,075	2,297	9,262	11,39
1,998	2,287	9,295	10,973
2,041	2,347	9,39	11,237
2,0195	2,3139	9,3122	11,1601

Tabelle 20: Lua (2)

empty	onefun	PrintPI 1	PrintPI 2	Nearest 1	Nearest 2
2,275	2,371	5,665	5,623	4,646	4,427
2,205	2,264	5,596	5,733	4,095	4,86
2,267	2,317	5,619	5,641	4,414	4,412
2,456	2,32	5,675	5,627	4,424	4,334
2,356	2,304	5,791	5,619	4,698	4,593
2,226	2,351	5,573	5,608	3,849	4,329
2,22	2,354	5,688	5,695	4,744	4,811
2,785	2,333	5,543	5,517	4,085	4,775
2,802	2,415	5,629	5,647	4,715	4,41
2,388	2,361	5,603	5,666	4,016	4,318
2,398	2,339	5,6382	5,6376	4,3686	4,5269

Tabelle 21: C++ (2)

empty	onefun	PrintPI 1	PrintPI 2	Nearest 1	Nearest 2
1,902	1,944	4,925	4,859	3,439	4,134
1,828	1,97	5,106	4,931	3,486	3,981
1,83	1,95	4,986	4,97	4,122	3,326
1,869	1,978	4,868	4,925	3,872	3,588
1,879	2,092	4,949	4,915	3,842	3,488
1,84	2,44	5,035	4,964	3,872	3,524
1,872	1,906	5,005	4,826	3,909	4,202
1,873	1,977	5,041	4,887	4,232	3,85
1,913	1,898	4,93	5,038	4,271	3,417
1,86	1,959	4,946	5,02	3,51	4,195
1,8666	2,0114	4,9791	4,9335	3,8555	3,7705

Tabelle 22: C++ (3)

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

1: Creative Commons, Namensnennung-Nicht-kommerziell-Weitergabe unter gleichen Bedingungen 3.0 Deutschland (CC BY-NC-SA 3.0), 2011, http://creativecommons.org/licenses/by-nc-sa/3.0/de/ (04.07.2011)

2: Aldebaran Robotics, History of the company Aldebaran Robotics, 2011, http://www.aldebaran-robotics.com/en/Company/history.html (12.09.2011)

3: Aldebaran Robotics, History of Nao robot at RoboCup, 2011, http://www.aldebaran-robotics.com/en/Solutions/For-Robocup/history.html (12.09.2011)

4: Wikipedia, Freiheitsgrad, 2011, http://de.wikipedia.org/wiki/Freiheitsgrad (12.09.2011)

0: Aldebaran Robotics, Schema of Nao robot, 2011, http://www.aldebaran-robotics.com/images/site/nao_schema_zoom.jpg (12.09.2011)

5: Aldebaran Robotics, Specification of Nao Robot, 2011, http://developer.aldebaran-robotics.com/join/#Specs (12.09.2011)

6: Advanced Micro Devices, AMD Geode LX Processor Family, 2007, http://www.amd.com/us/Documents/33358e_lx_900_productb.pdf (12.09.2011)

7: RoboCup, A Brief History of RoboCup, 2011, http://www.robocup.org/about-robocup/a-brief-history-of-robocup/ (13.09.2011)

8: RoboCup, Objective - The Dream, 2011, http://www.robocup.org/about-robocup/objective/ (13.09.2011)

9: RoboCup, About RoboCup, 2011, http://www.robocup.org/about-robocup/ (13.09.2011)

10: RoboCup, RFC Overview, 2011, http://www.robocup.org/wp-content/uploads/2010/02/RCF-Overview.png (13.09.2011)

11: RoboCup, RoboCup Soccer, 2011, http://www.robocup.org/robocup-soccer/ (13.09.2011)

12: TU Dortmund, RoboCup 2010 - Best Of, 2010, http://www.irf.tu-dortmund.de/cms/de/IT/Forschung/Robotics/RoboCup/RoboCupCinema/ndd-robocup2010-bestof-poster.jpg (13.09.2011)

13: Wikipedia, Simple DirectMedia Layer, 2011, http://en.wikipedia.org/wiki/Simple_DirectMedia_Layer (15.09.2011)

14: Wikipedia, List of games using SDL, 2011, http://en.wikipedia.org/wiki/List_of_games_using_SDL (15.09.2011)
15: Dr. Bjarne Stroustrup, Evolving a language in and for the real world: C++ 1991-2006, 2007, http://www.research.att.com/~bs/hopl-almost-final.pdf (15.09.2011)
16: Wikipedia, C++, 2011, http://de.wikipedia.org/wiki/C%2B%2B (15.09.2011)
17: Herb Sutter, We have an international standard: C++0x is unanimously approved, 2011, http://herbsutter.com/2011/08/12/we-have-an-international-standard-c0x-is-unanimously-approved/ (15.09.2011)
18: Scott Meyers, Summary of C++11 Feature Availability in gcc and MSVC, 2011, http://www.aristeia.com/C++0x/C++0xFeatureAvailability.htm (15.09.2011)
19: Apache, C++0xCompilerSupport, 2011, http://wiki.apache.org/stdcxx/C++0xCompilerSupport (15.09.2011)
20: Lua, About, 2011, http://www.lua.org/about.html (15.09.2011)
21: Rasterbar Software, Introduction, 2005, http://www.rasterbar.com/products/luabind/docs.html (15.09.2011)
22: Free Software Foundation, A Brief History of GCC, 2008, http://gcc.gnu.org/wiki/History (15.09.2011)
23: Free Software Foundation, C++0x Support in GCC, 2011, http://gcc.gnu.org/projects/cxx0x.html (15.09.2011)
24: Debian, How'd it all get started?, http://www.debian.org, http://www.debian.org/intro/about#history (15.09.2011)
25: KDevelop, Welcome to KDevelop.org, 2011, http://kdevelop.org (15.09.2011)
26: Matt Mackall , Mercurial v0.1 - a minimal scalable distributed SCM, 2005, http://lkml.indiana.edu/hypermail/linux/kernel/0504.2/0670.html (15.09.2011)
27: Brent Fulgham, The Computer Language Benchmarks Game, 2011, http://shootout.alioth.debian.org (15.09.2011)
28: Brent Fulgham, Which programming language is best?, 2011, http://shootout.alioth.debian.org/u32/which-language-is-best.php?calc=chart&fpascal=on&gcc=on&gpp=on&ghc=on&csharp=on&v8=on&lua=o

n&perl=on&javasteady=on&php=on&python3=on&cint=on&xfullcpu=1&xmem=0&xloc=0&nbody=1&fannkuchredux=1&meteor=0&fasta=1&fastaredux=1&spectralnorm=1&revcomp=1&mandelbrot=1&knucleotide=1&regexdna=1&pidigits=1&chameneosredux=0&threadring=0&binarytrees=1 (15.09.2011)
29: Brent Fulgham, Which programming language is best?, 2011, http://shootout.alioth.debian.org/u32/which-language-is-best.php?calc=chart&gpp=on&gcc=on&javasteady=on&fpascal=on&ghc=on&csharp=on&v8=on&lua=on&python3=on&php=on&perl=on&cint=on&xfullcpu=0&xmem=1&xloc=0&nbody=1&fannkuchredux=1&meteor=0&fasta=1&fastaredux=1&spectralnorm=1&revcomp=1&mandelbrot=1&knucleotide=1&regexdna=1&pidigits=1&chameneosredux=0&threadring=0&binarytrees=1 (15.09.2011)
30: Brent Fulgham, Which programming language is best?, 2011, http://shootout.alioth.debian.org/u32/which-language-is-best.php?calc=chart&perl=on&python3=on&v8=on&lua=on&cint=on&php=on&fpascal=on&csharp=on&javasteady=on&ghc=on&gpp=on&gcc=on&xfullcpu=0&xmem=0&xloc=1&nbody=1&fannkuchredux=1&meteor=0&fasta=1&fastaredux=1&spectralnorm=1&revcomp=1&mandelbrot=1&knucleotide=1&regexdna=1&pidigits=1&chameneosredux=0&threadring=0&binarytrees=1 (15.09.2011)
31: Brent Fulgham, Which programming language is best?, 2011, http://shootout.alioth.debian.org/u32/which-language-is-best.php?fpascal=on&gcc=on&gpp=on&ghc=on&csharp=on&v8=on&lua=on&perl=on&javasteady=on&php=on&python3=on&cint=on&xfullcpu=1&xmem=1&xloc=1&nbody=1&fannkuchredux=1&meteor=0&fasta=1&fastaredux=1&spectral-norm=1&revcomp=1&mandelbrot=1&knucleotide=1&regexdna=1&pidigits=1&chameneosredux=0&threadring=0&binarytrees=1&calc=chart (15.09.2011)
32: Lewis Van Winkle, Game Scripting Languages, 2009, http://codeplea.com/game-scripting-languages (08.06.2011)
33: Anders Schau Knatten , Don’t be Afraid of Returning by Value, Know the Return Value Optimization, 2011, http://blog.knatten.org/2011/08/26/dont-be-afraid-of-returning-by-value-know-the-return-value-optimization/ (20.09.2011)
34: Boost, Boost.Extension, 2008, http://boost-

extension.redshoelace.com/docs/boost/extension/index.html (23.09.2011)
35: Frictional Games, Tech feature: Script Overview #1, 2011, http://frictional-games.blogspot.com/2011/07/tech-feature-script-overview-1.html (23.09.2011)