2020

Laborbuch

„HS-KL goes RoboCup 2021“

TEAM HS-KL, LEITUNG: PROF. ADRIAN MÜLLER, FB I/MST

Inhalt

Team HS-KL 4

Ansprechpartner 4

Dokumentation 4

Hinweise für Autoren, To-Do 5

Unsere Strategie 5

Hintergrundinformation 5

Spielfeld und Testen in der Praxis 6

Strategie: Optionen für das Vorgehen 6

SimRobot (ein Simulator für den RoboCup) 7

Voraussetzungen für das Arbeiten mit dem Simulator: PC mit Ubuntu 18.04 7

Benötigte Abhängigkeiten auf Deinem PC installieren 8

Installieren der IDE 8

Klonen der Codebase 8

Für unsere IDE *CodeLite* einen Workspace generieren 8

Zugang zum Projektserver zwoogle4 9

Dokumentation erzeugen mit Doxygen, Lesetipps in der Doku 11

Simulator: Neue Befehle *randomize* und *spread* für den gc 13

Simulator: Logging von Daten für Benchmarks und Tests 16

Pipeline: Gitlab-ci Config für Docker 17

Erstellung einer Objekterkennung 18

Erkennung eines einzelnen Nao Roboters 18

Nao: Flashen der Robocup Firmware 21

Vorbereiten des Flashers unter Ubuntu 18.04 21

USB-Stick zum Flashen vorbereiten 21

Flashen des USB-Sticks 22

Flashen der Software auf dem Nao 22

Nao: Deployen der Software auf dem Roboter 23

Den Code zum Deployen bauen 23

Den Nao mit dem PC verbinden 23

Den Nao vorbereiten 25

Das Deployment vorbereiten 27

Den eigentlichen Code deployen 28

NAO-PS4-Gamepad Support 29
Pipeline: Grundgerüst für Regressionstests 30
Papierprototyp 31
Die CSV „SequenzFileOfWorldModelX“ 31
Die Szenarios 32
Die Simulationsdaten 32
Die Daten der Bilderkennung 33
Simulator: Erstellen von Scenes 34
Simulator: Ein Szenario erstellen und einem Team zuweisen 36
Skills & Cards 37
Einen Skill anlegen 40
Troubleshooting 42
Eine Card anlegen 42
Troubleshooting 46
Teach-In Cards 47
Konzeption 47
Der neue Dispatcher (erweiterter PriorityListDealer) 48
Teach-In Cards proof-of-concept 51
Best-practices für den B-Human Code 53
Das Feld: Koordinaten, Aufsetzpunkte, Roboternummer 53
Tipps für den Simulator 54
Konzeption Behaviour, Card, Skill, Sensoren 54
Worldmodel, RobotPose, Odometrie, Vision, Representations 55
Nützliche Codesnippets 56
Deprecated 58
“Erstellung einer Objekterkennung” (Version von Salome Schlemer) 58

Dokumenthistorie

15.6.2020 Erste Fassung Jonas

16.6.2020 Überarbeitung Adrian: Eingefügt Abschnitte Team, Historie, Ziele dieses Laborbuches; Abschnitte Titel erweitert, Seitennummern, Verweise

17.6.2020 Überarbeitung Jonas: Eingefügt Screenshots und Zeilenangaben

29.6.2020 Überarbeitung Felix: Einfügen von Verbesserungsvorschläge für das Laborbuch

04.07.2020 Autor: Salome Schlemer, Ergänzung des Kapitels „Erstellung einer Objekterkennung" Ausformulierung der ersten Versuche

7.7.2020 Autor Adrian: Neues Kapitel Strategie und allgemeine Information

14.7.2020 Autor Jonas: Neues Kapitel zum Flashen des Nao

17.7.2020 Autor Jonas: Neues Kapitel zum Einspielen des Codes auf dem Nao, Erweiterung des Kapitels für die Vorbereitung.

15.10.2020 Autor Adrian: Ergänzungen bei Team Liste, der IDE (wie baue ich SimRobot), Zugang zur zwoogle4 (VPN, TightVNC von Windows aus und von Ubuntu aus).

16.10.2020 Autor Adrian: Stand 2.1 (added) Dokumentation erzeugen mit Doxygen

3.11.2020 Autor Felix Mayer: Stand 2.2 (added) Kapitel Papierprototyp mit Unterkapiteln Die CSV „SequenzFileOfWorldModelX" und Die Simulationsdaten

4.11.2020 Autor Thomas Jäger: Stand 2.3 (added) Kapitel "Simulator: Zwei Teams mit unterschiedlichen Card Decks spielen lassen".

5.11.2020 Autor Felix Mayer: Stand 2.4 (added) Kapitel Papierprototyp/Die CSV „SequenzFileOfWorldModelX" um ein Datensatz Beispiel erweitert und auf die neuen Kommando Bezeichnungen angepasst.

06.11.2020 Autor: David Kostka, Ergänzung des Kapitels „Erstellung einer Objekterkennung" mit Unterkapitel "Erkennen eines einzelnen Nao Roboters"

10.11.2020 Autor Thomas Jäger: Stand 2.6 (added) Kapitel "Cards & Skills" mit den Unterkapiteln "Einen Skill anlegen", „Eine Card anlegen" und „Einen Skill in Card verwenden"
11.11.2020 Autor Adrian: Kapitel "Cards & Skills überarbeitet: fachlichen Übersicht ergänzt, Abschnitt Skill eingefügt.
11.11.2020 Autor Jonas: Kapitel „Erstellen von Scenes"

11.11.2020 Autor Thomas: Kapitel „Simulator: Zwei Teams mit unterschiedlichen Card Decks spielen lassen“ überarbeitet. Neue Überschrift: „Simulator: Ein Szenario erstellen und einem Team zuweisen “

17.11.2020 Autor Felix Mayer: Stand 2.9 (testet) Kapitel „Skills anlegen“. Ergänzungen und leichte Korrekturen. Allgemeine Erklärung von Bulletpoints-Stil in Fließtext geändert.

30.11. Autor Adrian. Added Jannis als Team Member. Best-practices mit dem B-Human Code added (4 Abschnitte – alle Inhalte aus Trello und e-mails hierher überführt außer B-Human Code Object Detection)

4.12. Autor Jonas: Kapitel „Nützliche Codesnippets“ angelegt.
21.12. 2020 (v2.13) Autor Adrian: added Markus (neues Team Mitglied). Im Kapitel „Skills & Cards“, Unterkapitel „Teach-In Cards“ angelegt. Konzeption, Integration, erste Ideen zur Data Mining Integration

30.12.2020 (v.2.14) Überarbeitung Markus: Review des gesamten Laborbuchs. In allen Kapiteln Rechtschreibfehler und Style behoben. Kapitel “Dokumentation” etwas erweitert (Erläuterung zum Kapitelaufbau des Dokuments und Leseempfehlungen)

05.01.2021 (v.2.15) Autor David Kostka: Kapitel “Bilderkennung” von Schlemer in neuen Abschnitt Deprecated verschoben. Kleine Änderung in “Unsere Strategie”.

Team HS-KL

Prof. Adrian Müller
David Kostka
Felix Mayer
Jannis Schottler
Jonas Lambing (BBS KL)
Markus Dauth
Thomas Jäger

Unter Mitwirkung von Salome Schlemer

Ansprechpartner

Prof. Adrian Müller - adrian.mueller@hs-kl.de

Dokumentation

Ziele dieses Laborbuches: Dieses Laborbuch dokumentiert die Arbeiten für die Teilnahme an der Qualifikation der deutschen RoboCup SPL 2021 der HS-KL für neue und bestehende Team

Mitglieder. Es ergänzt das GIT Repository https://zwoogle3.informatik.hs-kl.de/root/hskl-robocupgermanopen-2021.git um folgende Punkte:

- Hintergrundinformation, Installationshinweise
- Wie modifiziere bzw. erweitere ich den Simulator *SimRobot*?
- Wie arbeite ich remote mit dem Projekt-Server zwoogle4.ds.fh-kl.de?
- Erläuterungen zur Architektur und zu Modifikationen am Robot Code
- Welche automatisierten Tests gibt es, Performance Benchmarking
- (später): Bilderkennung mit Keras und TensorFlow: Testdaten, Modelle, Evaluierung

Dieses Laborbuch ist linear aufgebaut und die Kapitel sind chronologisch angeordnet. Neuere Kapitel überschreiben die Inhalte von Älteren.

Folgende Dokumente sollten vor dem Lesen dieses Laborbuchs angesehen werden:

- Der Datei CodeRelease2019.pdf von B-Human unter https://github.com/bhuman/BHumanCodeRelease
- Das aktuellste Regelwerk für die SPL unter https://spl.robocup.org/

Die Arbeiten basieren aktuell auf dem Code Drop: https://github.com/bhuman/BHumanCodeRelease

Die Dokumentation des Teams B-Human zum Bauen und anderen Prozessen befindet sich im Root der Repository in *CodeRelease2019.pdf*.
Wir verwenden die IDE CodeLite. Das erforderliche Systemimage für die echten Naos ist das nao-2.8.5.11_ROBOCUP_ONLY.

Hinweise für Autoren, To-Do

Alle Verbesserungsvorschläge beziehen sich auf das Laborbuch, es sei denn es wird explizit etwas Anderes genannt.

- Wenn Dateien modifiziert oder neu erstellt werden, sollte auf den zugehörigen Ordner verwiesen werden, bzw. der Path mit hinterlegt werden, um die Datei schneller zu finden.

- Angabe nicht über Zeile, denn nach Erweiterungen stimmt diese nicht mehr überein. Vorschlag: im Kopf der Datei eine Notiz wie: Änderung bei Datum*Autor*Funktion

Unsere Strategie

Input: unsere Erkenntnisse aus 6/2020 mit dem B-Human Code Release 2019, deren Simulator SimRobot, und ein Telefonat mit Rico Tilgner, Teamleiter Leipzig, Diskussion mit Jonas

Hintergrundinformation

Stand: 07/2020

- Der Code Release 2019 von B-Human ist historisch gewachsener Code. Die einzelnen Module (TeamStrategie, WalkingEngine etc.) sind über Jahre hin optimiert worden
- Die Bewegungen des Naos im *RoboCup* sind 2-3x schneller als im nativen „Kind“ Modus. Kaum ein Team benutzt NaoQi, da der aktuelle Release den Durchgriff auf seine WalkingEngine verbietet.
- Zugang zum Nao erfolgt über *LoLa* (s. LoLa RoboCupper Official Documentation.pdf). Es gibt einen LoLa Connector im GitHub der HTWK Leipzig (https://github.com/NaoHTWK). Es ist unklar, wie *LoLa* in *B-Human* integriert ist (-> Praktikum Jonas)
- Alle Teams testen vorwiegend in der Realität, erzeugen Log-Dateien, speichern alle 1 bis 2 Sekunden full frame Bilder. Das Erkennen von Robotern, und die Freund-Feind Unterscheidung sind schwierig.
- Es werden inoffizielle Testspiele zwischen den Teams organisiert.
- Für die reguläre TensorFlow Laufzeit ist die CPU des Nao zu schwach -> Verwenden von ~~TensorFlow Lite~~ JIT-Compiler “CompiledNN”. Dessen Kompilation ist „eine bitch“ (Zitat Rico), er sendet uns die Library zu.
- Es gibt eine On-Board GPU; deren Einsatz macht aber Probleme (Zitat Rico).
- Dieses Jahr entfiel der RoboCup.

Spielfeld und Testen in der Praxis

- Achtung: wechselnde Lichtverhältnisse. Auto-white balance erforderlich (Neon Licht)
- Unser Ansatz mit dem ¼ Feld könnte klappen, aber:
  - Mind. 0.5m Grün hinter den Außenlinien (oder an der Wand dahinter)
  - Der Anstoßkreis muss vollständig sichtbar sein
- Bezugsquellen
  - Fußball: s. Regelwerk
  - Tor: Plastikrohre (weiß) aus dem Baumarkt, Durchmesser 10cm
  - https://messe-creation.de/kunstrasen-boston/
- Wie starten? Rico sendet uns die Binaries der Weltmeister-Mannschaft (!), und ein Installation-Script
  - Darin ist u.a. die Spielfeldgröße wählbar (6x4, 6x9, andere)
- Ideen zur Testdaten Generierung
  - Ferngesteuerte Spiele
  - 2 gegen 3 Mannschaften (Code Leipzig, unser Code)
  - RC „plastic Naos“

Strategie: Optionen für das Vorgehen

Zitate Rico

„Vorsicht vor dem B-Human Clone“ Effekt“

Wechsel des Frameworks tut weg

„Macht alles anders als B-Human“

Das folgende ist eine Momentaufnahme, Stand 7.7.2020, und wird weiterentwickelt:

Sinnvolle Strategien sind

a) Re-Engineering Framework B-Human oder rRUNSWIFt

b) Eigenes Framework + Module anderer Teams

Eigene Entwicklung bei

- o TeamStrategie
- o Pässe
- o Dribbling, Gegner vermeiden
- o Trade-Off Speed / Überhitzen Motoren / Stress auf Gelenke Annecy

SimRobot (ein Simulator für den RoboCup)

Voraussetzungen für das Arbeiten mit dem Simulator: PC mit Ubuntu 18.04

Hinweis: das folgende ist eine Kurfassung des Handbuch B-Human „CodeRelease2019.pdf“, Kap 2. GettingStarted“

Ein durchschnittlicher PC genügt. Der Projekt Server zwoogle4.ds.fh-kl.de (im VPN der HS-KL zugreifbar) verfügt über ausreichend Performanz für flüssige Simulationen. Darauf sind folgende Pakete erforderlich

- Clang
- Make
- Qtbase5-dev
- Libqt5opengl5-dev
- Libqt5svg5-dev
- Libglew-dev
- Libasound-dev
- Ocl-icd-opencl-dev
- Net-tools
- Graphviz
- Xterm

Das Arbeiten unter Windows ist nicht empfohlen, da eine veraltete Version von Windows 10 benötigt wird, welche nicht geupdatet werden darf.

Benötigte Abhängigkeiten auf Deinem PC installieren

Den Befehl

"sudo apt install clang make qtbase5-dev libqt5opengl5-dev libqt5svg5-dev libglew-dev libasound-dev ocl-icd-opencl-dev net-tools graphviz xterm"

im Terminal ausführen. Achtung: keine White-Spaces um die "-" herum!

Installieren der IDE

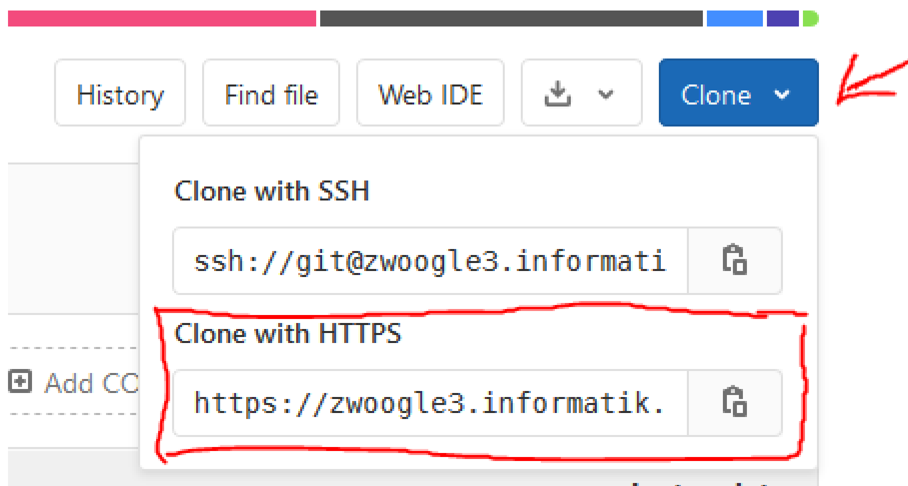
Die IDE kann mit dem Befehl *apt install CodeLite* über das Terminal installiert werden. Danach ist sie direkt startklar.

Klonen der Codebase

Das Git-Repository für das Projekt ist unter folgendem Link zu finden:
https://zwoogle3.informatik.hs-kl.de/root/hskl-robocupgermanopen-2021

Vor dem Klonen muss Git mit dem Befehl *apt install git* auf der Maschine installiert werden.

Danach den Link zum Klonen über den oben genannten Repository-Link kopieren



Nun kann man einen beliebigen Ordner im Terminal öffnen (z.B. den home-Ordner) und die Codebase mit dem Befehl *git clone <URL>* klonen (Git erstellt automatisch einen Unterordner für die Repo).

Für unsere IDE *CodeLite* einen Workspace generieren

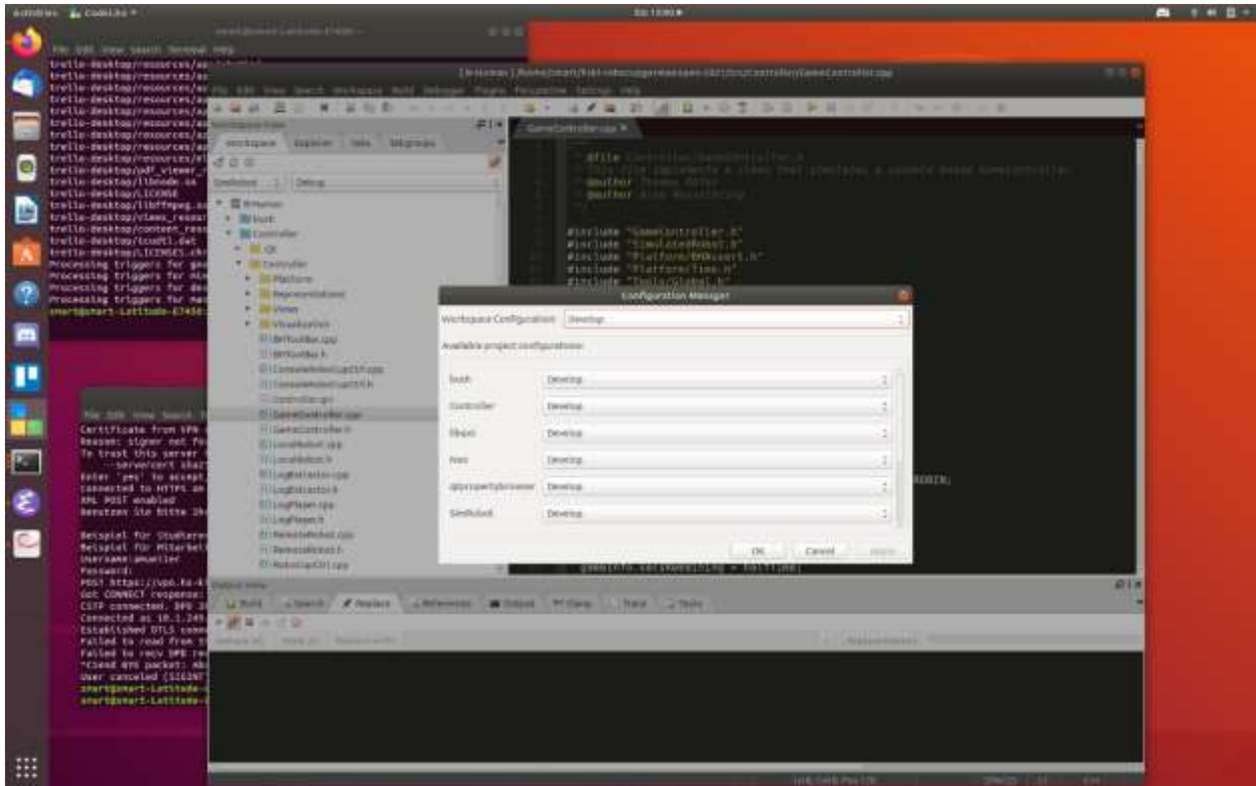
Im Terminal nach *[Repo]/Make/LinuxCodeLite* navigieren („cd") und lokal das Skript *generate* ausführen (./generate).
Danach kann der generierte Workspace mit *CodeLite* geöffnet und verändert werden:

- Starten von „codelite"

- Menu “Workspace/Open Existing Workspace”
- Pfad: [Repo]/hskl-robocupgermanopen_2021/ *Make/LinuxCodeLite*/B-Human.workspace
- Tipp: Perspective auf „default“ zurücksetzen

Hinweis: alle Projekte sind unter „B-Human“ eingeordnet. Beim „build“ werden initial alle Module aus CodeLite heraus gebaut. Wichtig: vorher den Build-Modus wählen (s. Grafik“Configuration Manager“).
Alternatives Bauen: Im Ordner Make/Linux der Codebase den Befehl „make SimRobot“.

Die ausführbare Binary befindet ist dann in [Repo]/**Build/Linux/SimRobot/Develop/SimRobot**


CodeLite auf Ubuntu 18.04

- Simulation auswählen: File:Open Config/Scenes “BH Testing.ros2”
- Doppelclick auf <robocup> -> Simulator öffnet
- Doppelclick auf <console> -> Console öffnet. Dort „help“ tippen

Mehr zur Bedienung Simulator: s. Handbuch B-Human „CodeRelease2019.pdf“, Kap. 2.6

Zugang zum Projektserver zwoogle4

Diese Maschine ist sehr leistungsstark, d.h., hier kann man bspw.

- Im SimRobot mehrere Roboter flüssig animieren
- TensorFlow Trainingsläufe effizient durchführen

Der Zugang erfolgt über VNC, und wurde getestet unter Windows10 und Ubuntu 18.04

Windows10

Mit Cisco Anyconnect ins VPN ins Hochschulnetz einwählen

a) tightVNC Viewer Version 2.8. für 10, oder besser: https://www.tightvnc.com/?f=va
   Connection: zwoogle4.ds.fh-kl.de::5900, pass: <erfragen>
b) Einloggen auf Ubuntu Desktop zwoogle4, pass: <erfrange)
c) Kann sein, dass die Verbindung am Anfang ein paar Male zusammenbricht. Daher vor einer Demo rechtzeitig „warmlaufen“ lassen

Hinweis: der Simulator muss lokal, auf der zwoogle4 an der Console im O028 gestartet worden sein. Auf dem Desktop befindet sich dazu ein Hinweis.

Ubuntu 18.04

Für VPN: Install openconnect on Ubuntu using

    sudo apt-get install network-manager-openconnect-gnome

Dann:

    sudo openconnect vpn.hs-kl.de

> Certificate from VPN server "vpn.hs-kl.de" failed verification.
Reason: signer not found
To trust this server in future, perhaps add this to your command line:
  --servercert
sha256:bc3d12e80da00ea6537c2edad01efcbd98e277315aa4cbb4ad94b5731e78dcaa
Enter 'yes' to accept, 'no' to abort; anything else to view:

    Yes

-Am Ende: ctr-C zum Beenden

Für VNC:

    sudo apt-get install xtightvncviewer
    xtightvncviewer zwoogle4.ds.fh-kl.de::5900

-Am Ende: Fenster schließen genügt

Dokumentation erzeugen mit Doxygen, Lesetipps in der Doku

Installation:
sudo apt-get install doxygen
// evtl. erforderlich
sudo apt --fix-broken install
// lädt libs nach etc.
sudo apt-get install doxygen
doxygen -g Doxyfile //generiert config Template

- ☒ Editieren der Datei Doxyfile; die meisten Defaults stehen lassen.

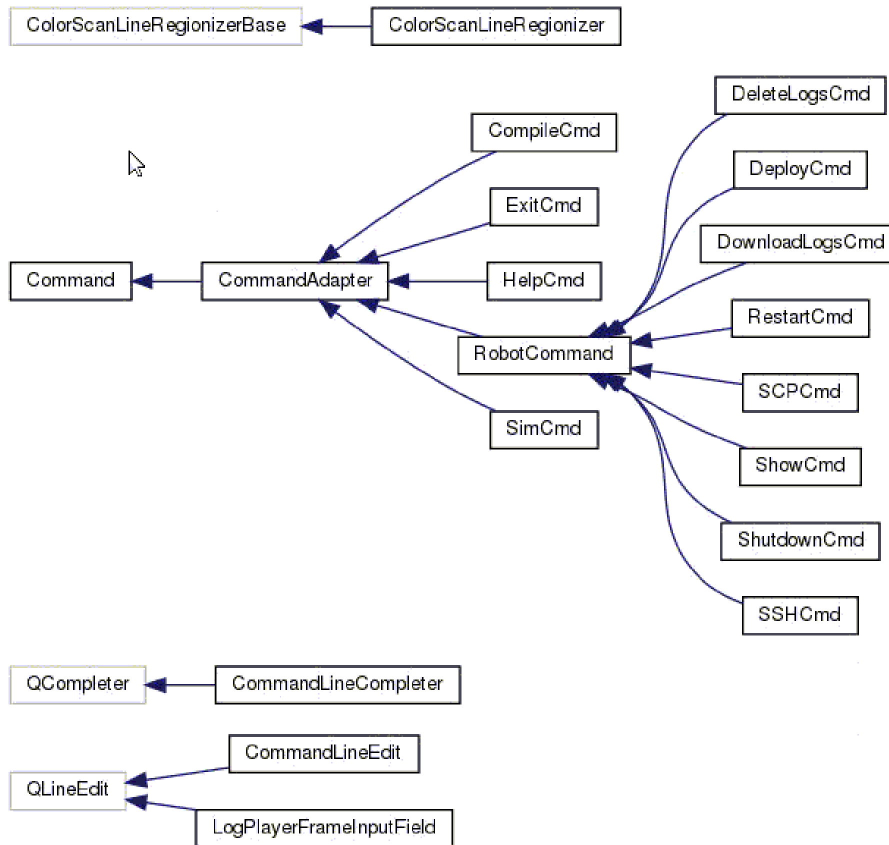
================= Doxyfile ====================================
PROJECT_NAME = "HS-KL RoboCup 2021"
INPUT = /home/maker/workspace/hskl-robocupgermanopen-2021/Src
RECURSIVE = YES
OUTPUT_DIRECTORY = /home/maker/workspace/hskl-robocupgermanopen-2021/
QT_AUTOBRIEF = YES
EXTRACT_PACKAGE = YES
===

Erzeugen bzw. refresh der Doku:
maker@zwoogle4:~/workspace/hskl-robocupgermanopen-2021/Src$ doxygen

Im Firefox öffnen
file:///home/maker/workspace/hskl-robocupgermanopen-2021/html/index.html

Empfehlenswerte Ansichten:

Klassenhierarchie graphisch
file:///home/maker/workspace/hskl-robocupgermanopen-2021/html/inherits.html



Namespaces

HS-KL RoboCup 2021

Main Page | Related Pages | Namespaces | Classes | Files

Projection Namespace Reference

Functions

void	**computeFieldOfViewInFieldCoordinates** (const RobotPose &robotPose, const CameraMatrix &cameraMatrix, const CameraInfo &cameraInfo, const FieldDimensions &fieldDimensions, std::vector< Vector2f > &p)
Geometry::Line	**calculateHorizon** (const CameraMatrix &cameraMatrix, const CameraInfo &cameraInfo)
bool	**calculateBallInImage** (const Vector2f &ballOffset, const CameraMatrix &cameraMatrix, const CameraInfo &cameraInfo, float ballRadius, Geometry::Circle &circle)
float	**getDistanceBySize** (const CameraInfo &cameraInfo, float sizeInReality, float sizeInPixels)
float	**getDistanceBySize** (const CameraInfo &cameraInfo, float sizeInReality, float sizeInPixels, float centerX, float centerY)
float	**getSizeByDistance** (const CameraInfo &cameraInfo, float sizeInReality, float distance)
void	**calculateAnglesForPoint** (const Vector2f &point, const CameraMatrix &cameraMatrix, const CameraInfo &cameraInfo, Vector2f &angles)
bool	**calculatePointByAngles** (const Vector2f &angles, const CameraMatrix &cameraMatrix, const CameraInfo &cameraInfo, Vector2f &point)

Simulator: Neue Befehle *randomize* und *spread* für den gc

15-06-2020, Hash e38e6e51, Author: Jonas Lambing

GIT Message
added basic benchmarking terminal output (first ballcontact, time until goal scored), added *gc randomize* + *gc spread* commands

Inhalt

- Ziel: Positionierung erfolgt im ¼ Feld, Random-Positionierung eines Nao's und eines Balls, Random-Positionierung eines Nao's mit einem Winkel zwischen 0° und 45° zum Ball, das Prefix gc steht für *GameController* – also alles was mit der eigentlichen Spielsimulation zu tun hat.

- Eine Testszene mit einem Ball und einem simulierten Nao
    - *Config/Scenes/BH Testing.ros2*
    - *Config/Scenes/BH Testing.con*

- Ein Include, das von der Testszene benutzt wird
    - Config/Scenes/Includes/Single.rsi2

- Die Konsolenbefehle *gc randomize und gc spread* im Simulator
    - Der Befehl *gc randomize* platziert den simulierten Nao und den Ball in der Testszene jeweils an zufallsgenerierten Positionen im rechten, oberen Viertel des Spielfelds (GameController.cpp, Z. 328):

```
// calculate delta
// scale down by 50% to end up in a quadrant
float dx = (fieldDimensions.xPosOpponentGroundline - fieldDimensions.xPosOwnGroundline) * 0.5f;
float dy = (fieldDimensions.yPosLeftSideline - fieldDimensions.yPosRightSideline) * 0.5f;

// randomize pos
float x = fieldDimensions.xPosOwnGroundline + fmodf(static_cast<float>(rand()), dx);
float y = fieldDimensions.yPosRightSideline + fmodf(static_cast<float>(rand()), dy);

// move robot
robots[i].simulatedRobot->moveRobot(Vector3f(x, y, dropHeight), Vector3f(), false);
```

Resultat:

- Der Befehl *gc spread* platziert den Nao an einem zufallsgenerierten Punkt entlang der Mittellinie am rechten, oberen Viertel. Der Ball wird statisch in der Mitte des unteren Rands des Viertels platziert (GameController.cpp, Z. 360):

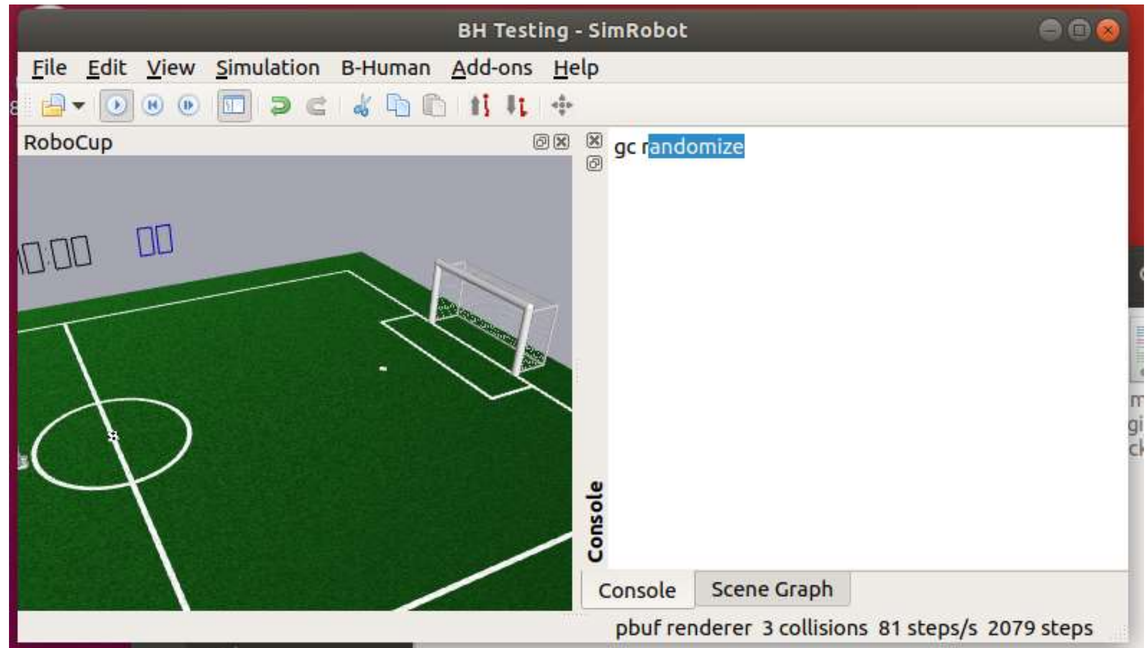
```
// randomize pos
// add arbitrary offset of 100.0
// goal gets selected according to the origin of the nao apparently
float x = fieldDimensions.xPosOwnGroundline + dx + 100.0f;
float y = fieldDimensions.yPosRightSideline + fmodf(static_cast<float>(rand()), dy);

// move robot
robots[i].simulatedRobot->moveRobot(Vector3f(0, y, dropHeight), Vector3f(), false);
```

Resultat:

- Die beiden Befehle wurden im Projekt *Controller* in der Datei *GameController.cpp* in der Funktion *GameController::handleGlobalCommand()* hinzugefügt.
- Des Weiteren wurden die Befehle auch zur Autocompletion in *GameController::addCompletion()* in *GameController.cpp* (Z. 888) und

ConsoleRoboCupCtrl::createCompletion() in *ConsoleRoboCupCtrl.cpp* (Z. 712) hinzugefügt:



- Terminal output für Benchmarking
  - Logging für geschossene Tore und vergangene Zeit seit Spielbeginn in *GameController::handleGoalCommand()* in *GameController.cpp (Z. 166)*
  - Logging für die Zeit vom Beginn des Spiels bis zum ersten Ballkontakt in *GameController::setLastBallContact()* in *GameController.cpp (Z. 813)*

```
default@default: ~/nao/hskl-robocupgermanopen-2021/Build/Linux/SimRobot/Release
File Edit View Search Terminal Help
default@default:~/nao/hskl-robocupgermanopen-2021/Build/Linux/SimRobot/Release$ ./SimRobot


[ BENCHMARK ]
First ballcontact after 7548 ms



[ BENCHMARK ]
Goal scored after 35076 ms
```

Simulator: Logging von Daten für Benchmarks und Tests

18-06-2020, Hash 89153b78, Author: Jonas Lambing

GIT Message
added benchmark logging to file

Inhalt

- Ziel: Zeitmessung bis zum ersten Ballkontakt und bis zum Tor nach Rundenanfang.

- Logging wird jetzt auch in einer Datei gespeichert, die in */Config/Scenes/benchmark_log.txt* liegt.

- Zum Loggen habe ich den alten Loggingcode etwas verändert und das Ganze in die Methode GameController::addBenchmarkLog() in GameController.cpp verpackt. Diese loggt nun ins Terminal und in die Logdatei.

```
void GameController::addBenchmarkLog(std::string text)
{
    std::ofstream logFile("benchmark_log.txt", std::ios::out | std::ios::app);
    logFile << text << std::endl;
    logFile.close();

    printf("\n\n[ BENCHMARK ]\n");
    printf("%s\n\n", text.c_str());
}
```

Pipeline: Gitlab-ci Config für Docker

18-06-2020, Hash 9fb8f0a2, Author: Jonas Lambing

GIT Message
Update .gitlab-ci.yml

Inhalt

- Ziel: Automatisiertes Bauen und Testen / Benchmarken des Projekts mit Hilfe eines Docker Containers und der Gitlab CI CD Pipeline. Dafür wird eine Configdatei benötigt die als Bauplan („Blueprint") verwendet wird.
- Dieser Bauplan gibt u.A. an, welches Image für das Betriebssystem des Containers verwendet werden soll. In unserem Fall Ubuntu 18.04.

```
default:
    image: "ubuntu:18.04"
```

- Der Plan ist auch in 2 Stages unterteilt: Build und Test.

```
stages:
    - build
    - test
```

- Die Buildstage besteht darin, in den *./Make/Linux* zu navigieren („cd") und den Buildbefehl auszuführen.

```
compile_simrobot:
    stage: build
    script:
        - cd ./Make/Linux/
        - make SimRobot Release
```

- Wenn die Buildstage vollendet ist, wird der Simulator mit der Testmap ausgeführt.

```
benchmark_simrobot:
    stage: test
    script: ./Build/Linux/SimRobot/Release/SimRobot "./Config/Scenes/BH Testing.ros2"
```

Erstellung einer Objekterkennung

Erkennung eines einzelnen Nao Roboters

Dieser Abschnitt dient als Zusammenfassung des Jutyper Notebooks “single_nao_trainer” (TODO: Pfad in Git angeben und kurz erklären was Jutyper ist).

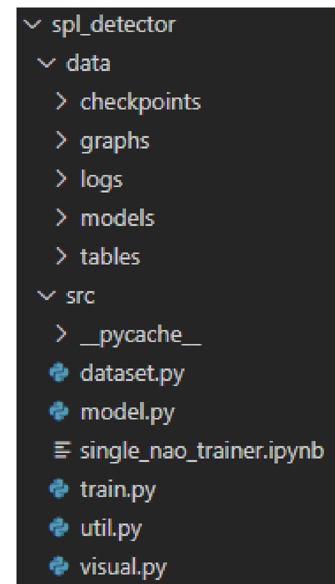
Für eine ausführlichere Dokumentation des Codes sollte das Notebook gelesen werden.

Ziele und Definitionen

Ziel: Erstellung und Training eines Modells, welches einen einzelnen Nao im Spielfeld erkennen und per Bounding Box lokalisieren kann. Dabei liegt der Fokus darauf einen ersten Funktionsprototyp zu erstellen und ihn an einer von uns generierten Spielsequenz zu testen, eine hohe Robustheit und Optimierung sind dabei nicht im Vordergrund.

Das Modell bekommt als Eingabe ein 100x100x1 Grayscale Bild (1 Kanal) und sollte als Ausgabe 4 Werte liefern, die als 4 BBOX Koordinaten (xmin, ymin, xmax, ymax) interpretiert werden.

Code Architektur



Im Bild rechts sieht man die Ordner- und Modulstruktur des Codes.

Der Ordner “data” beinhaltet jegliche Daten wie Modelle, Graphen und CSV Tabellen, die bei der Verarbeitung der Trainingsdaten und beim Training des Modells erzeugt werden.

In “src” befinden sich mehrere Module, die Hilfsfunktionen und Datenstrukturen bereitstellen:

- dataset.py: Vorverarbeitung von Trainingsdaten
- model.py: Definition der Architektur des CNNs
- util.py: Sonstige hilfreiche Algorithmen
- visual.py: Visualisierung von Bildern/Daten (z.B. BBOX)

In “single_nao_trainer.ipynb” werden schließlich die genannten Module verwendet, um die Trainingspipeline zu bilden und somit das eigentliche Training des Modells durchzuführen.

Trainingsdaten

Um das Modell zu trainieren werden von dem Simulator generierte Daten benutzt, keine echten Bilder und Labels. Der Vorteil daran ist das durch automatische Annotation der Bilder im Simulator mehr Labels gesammelt werden können wie z.B. die Distanz und Orientierung der Naos im Bild, die manuell für Menschen sehr mühsam zu erstellen wären.

Der folgende Datensatz wird genutzt: https://sibylle.informatik.uni-bremen.de/public/JET-Net/ (TODO: Kurz erklären, wo der Datensatz herkommt)

Dieser besteht aus 10 Unterordnern mit den Sichtweisen der 10 Naos im Spiel. In jedem Unterordner gibt es ein “Images/” Ordner mit den Bildern und eine Tabelle “labels.csv” für die Labels der Bilder. Die Labels haben die Spalten: (filename, minX, minY, maxX, maxY, type, distance, orientation)

Vorverarbeitung
Da immer nur ein einzelner Nao erkannt werden soll, muss das Dataset zum Training auch so gefiltert werden, dass nur Bilder übrig bleiben bei denen ein einziger Nao im Bild ist, da bei mehreren Naos keine eindeutige BBOX geschätzt werden kann.
Sind die Daten gefiltert, müssen sie in einem mit Keras kompatiblen Format konvertiert werden, um sie der model.fit() Methode zu übergeben.
Daher werden die Daten erst im TFRecord Format auf der Festplatte gespeichert, um später daraus ein tf.data.Dataset Objekt zu erzeugen, dafür gibt es die Funktionen “`create_tfrecord_from_dir()`” und “`load_combined_dataset()`” in “dataset.py”.
Das Dataset Objekt kümmert sich um die Speicherverwaltung der Trainingsdaten und kann direkt der model.fit() Methode übergeben werden. Beim Training werden dann die Daten durch das TF Dataset Objekt automatisch inkrementell ins RAM geladen, wenn sie benötigt werden, so dass wir uns nicht darum kümmern müssen. Mehr Infos dazu im Notebook und in “dataset.py”

Modell Architektur
Das Modell ist ein CNN und wird in dem Modul 'model.py' definiert.
Die Architektur orientiert sich am JET-Net von B-Human, aber leicht modifiziert und ohne 'depthwise separable convolutions'.
Input: Bild mit größe “target_size” aus “dataset.py” (Momentan 100x100x1)
Output: 4 BBOX Koordinaten
Struktur:

- Bestehend aus 4 'Blöcken' in Sequenz
- Jeder Block besteht aus den folgenden Layern:
    1. Batch Normalisierung
    2. n mal: Convolution (24 Kernels, 3x3) + Leaky ReLU
    3. Convolution mit strides=(2,2) für Downsampling
    4. Leaky ReLU
- Letzter Block weicht ab: kein Pooling am Ende wie in den anderen Blöcken
- Output Layer ist ein Dense Layer mit 4 Nodes

```
x = BatchNormalization()(x)
x = Conv2D(24, (3, 3), strides=(1,1), padding='same', use_bias=False, kernel_initializer='he_uniform')(x)
x = LeakyReLU(alpha=0.1)(x)
x = Conv2D(24, (3, 3), strides=(1,1), padding='same', use_bias=False, kernel_initializer='he_uniform')(x)
x = LeakyReLU(alpha=0.1)(x)
x = Conv2D(24, (3, 3), strides=(2,2), padding='same', use_bias=False, kernel_initializer='he_uniform')(x)
x = LeakyReLU(alpha=0.1)(x)
```

Die Funktion “SingleNaoModel()” in “model.py” liefert die Modelldefinition, im Notebook wird es dann Kompiliert und ein Graph der Architektur ausgegeben.

Training

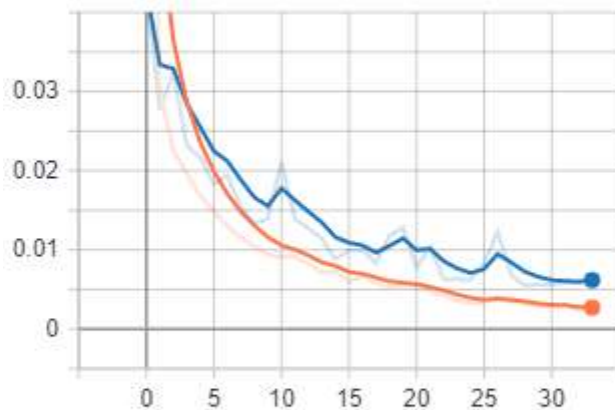
Sind die Daten und das Modell bereit, kann jetzt das eigentliche Training stattfinden. Dafür wird in der model.fit() Methode das Trainings- und Validierungsdataset übergeben.

```
model.fit(
    train_dset_opt,
    validation_data=val_dset_opt,
    epochs=50,
    callbacks=[callbacks]
)
```

Desweiteren werden einige “Callbacks” übergeben. Callbacks in Keras sind Funktionen, die beim Training nach jedem Batch, jeder Epoche oder beim Trainingsende von Keras aufgerufen werden. Hier werden für das Training folgende vordefinierte Standard-Callbacks übergeben:

- **EarlyStopping**: Falls sich der Fehler der Validiationsdaten nach den letzten n Epochen nicht verbessert hat, stoppe das Training.
- **ModelCheckpoint**: Speichere das Modell nach jeder Epoche ab, als Backup falls das Training abbricht. Dabei wird immer nur das bisher beste Modell gespeichert.
- **TensorBoard**: Zur Visualisierung der Logs wie Metriken, Fehlerfunktion, Graphen, usw. (s. Tensorboard Doku). Weiter unten sieht man ein von TensorBoard erzeugten Graph, welcher den Fehler der Trainings und Val. Daten per Epoche ausgibt.

Hier wurden für das Training zwar 50 Epochen angegeben, dank EarlyStopping stoppt es aber schon bei ca. 30 Epochen automatisch, da sich das Modell danach nicht mehr verbessert.



Zum schluss wird das Trainierte Modell in verschiedenen Formaten gespeichert und mit TFLite Kompiliert.

Nao: Flashen der Robocup Firmware

Hinweis: Das Image *nao-2.8.5.11_ROBOCUP_ONLY_with_root.opn* und eine funktionsfähige Kopie des *NaoFlashers* befinden sich im Downloads-Ordner auf der *zwoogle4*.

Vorbereiten des Flashers unter Ubuntu 18.04

Zuerst muss die Flashersoftware heruntergeladen werden, aktuelle Releases gibt es unter: https://www.softbankrobotics.com/emea/en/support/nao-6/downloads-softwares

Der Flasher funktioniert unter Ubuntu 18.04 standardmäßig nicht, da der eine veraltete Version des ZLIB-Moduls verwendet. Dies kann aber relativ einfach behoben werden.

Nach dem Extrahieren des Flashers muss man in den Unterordner */lib/* im Flasherverzeichnis navigieren und diesen im Terminal öffnen.

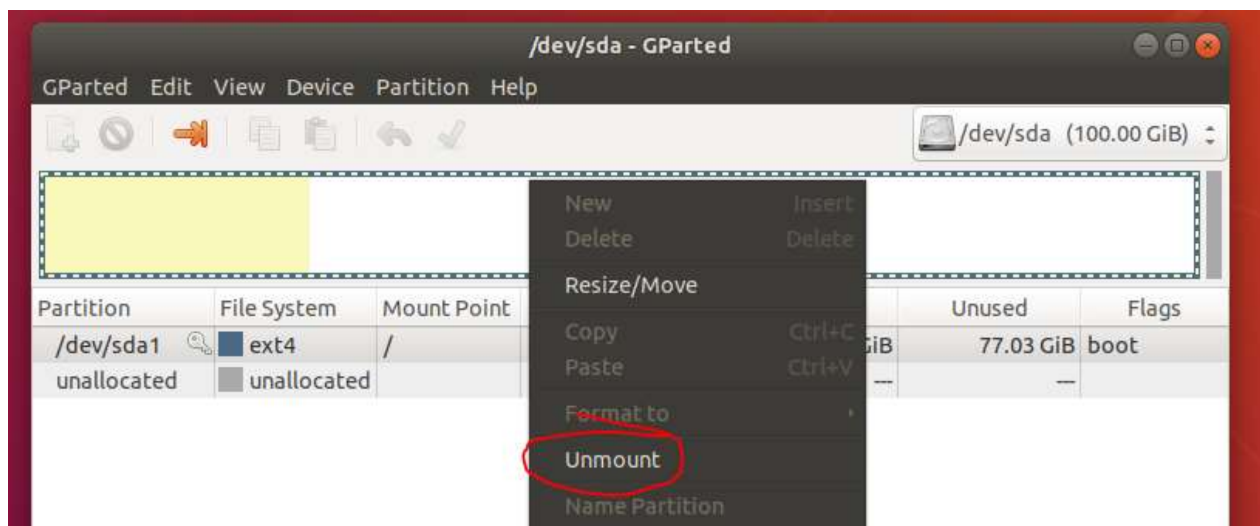
Dort benennt man das alte ZLIB-Modul mit dem Befehl *mv libz.so.1 libz.so.1.old* um und erstellt mit dem Befehl *ln -s /lib/x86_64-linux-gnu/libz.so.1* einen Link zur neueren Version des ZLIB-Moduls im System. Jetzt kann man den Flasher Problemlos ausführen (Immer als Root starten!).

USB-Stick zum Flashen vorbereiten

Der USB-Stick darf kein Datei- oder Partitionssystem enthalten! Dieses kann mit einem Tool wie Gparted unter Linux entfernt werden.

Man installiert und startet Gparted und wählt der USB-Stick rechts oben aus (ACHTUNG! Immer darauf achten, dass das richtige Speichermedium ausgewählt wird!)

Diesen muss man jetzt per Rechtsklick auf die Partitionstabelle zuerst unmounten.



Danach kann man entweder *Entf* auf der Tastatur drücken, oder die Partitionen per Rechtsklick und Delete auf die Tabelle entfernen.

Im letzten Schritt muss man noch die Änderungen anwenden. Dazu klickt man auf den Haken oben im Programm.

Flashen des USB-Sticks

Man öffnet den Ordner des Flashers im Terminal und startet ihn mit dem Befehl *sudo ./flasher.*

Über den Button *Browse* neben *Image to flash* kann eine Firmwaredatei ausgewählt werden (in unserem Fall *nao-2.8.5.11_ROBOCUP_ONLY_with_root.opn*).

Unter *Choose your usb stick* wird der zu flashende USB-Stick ausgewählt (ACHTUNG! Immer sicherstellen, dass der richtige Stick ausgewählt ist, der Flasher kann sonst Schäden an den Partitionen des ausgewählten Speichermediums verursachen!).

Jetzt muss man nur noch die Option *Factory Reset* aktivieren und auf den Button *Write* klicken.

Flashen der Software auf dem Nao

Zuerst muss der Nao komplett ausgeschaltet werden (Chestbutton so lange drücken, bis Nao ein akustisches Signal ausgibt).

Dann wird der Stick in die USB-Buchse am Hinterkopf des Naos eingesteckt und man hält den Chestbutton ca. 5s gedrückt, bis er anfängt blau zu leuchten.

Das Update dauert eine Weile und der Roboter sollte nach Abschluss automatisch rebooten.

Näheres zum Flashingprozess findet man hier: http://doc.aldebaran.com/2-1/software/naoflasher/naoflasher.html und hier: *http://doc.aldebaran.com/2-1/nao/boot_process_nao.html#upgrading-process-nao.*

Nao: Deployen der Software auf dem Roboter

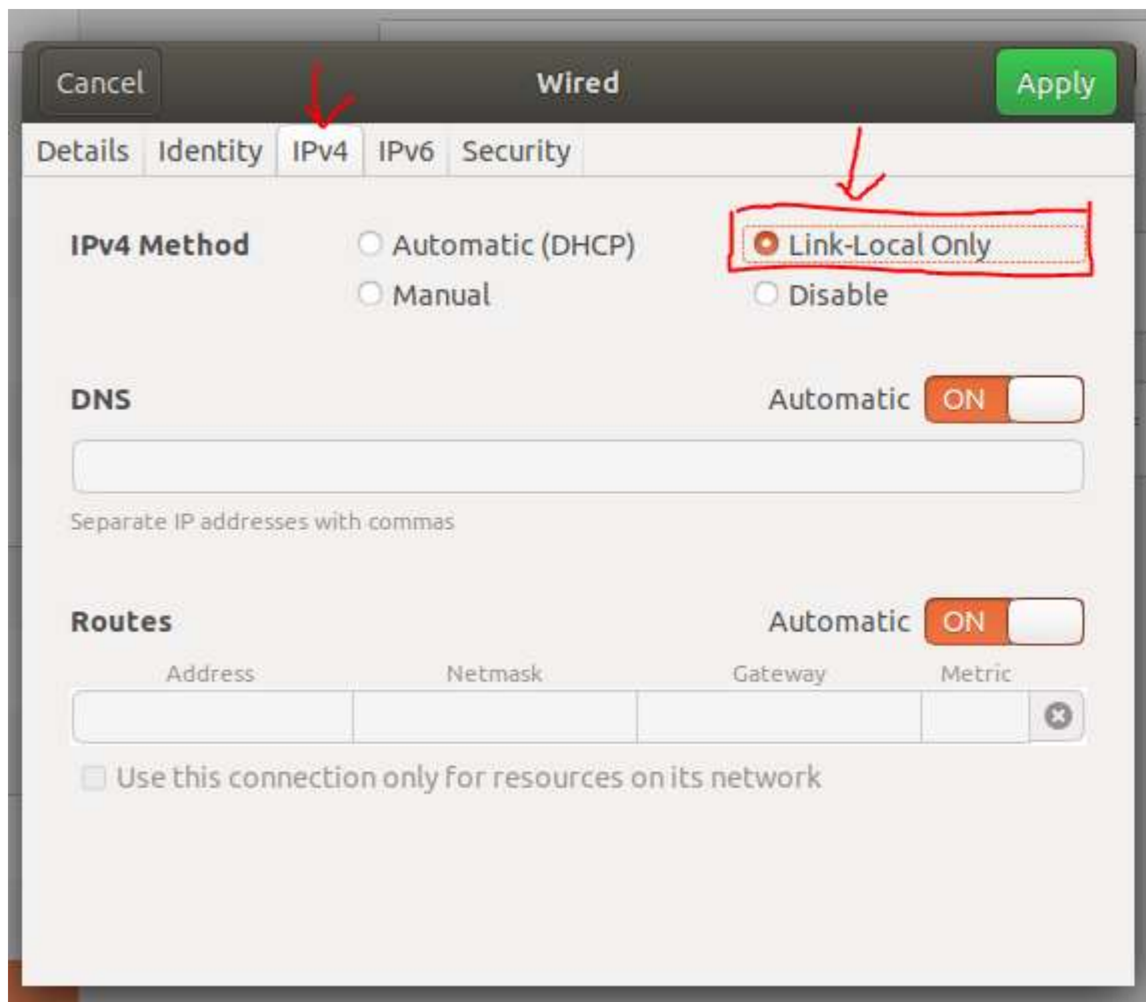
Den Code zum Deployen bauen

Man öffnet den Ordner *<Git-Repo>/Make/Linux* und führt den Befehl „make all“ aus (dieser baut alle Projekte, wenn nichts anderes spezifiziert ist, wird die Develop Buildconfig benutzt. Weitere Optionen findet man in der *CodeRelease2019.pdf* im Punkt 2.2).

Den Nao mit dem PC verbinden

Zuerst muss der Nao mit einem Patchkabel an die LAN-Buchse des PCs angeschlossen werden.

Danach öffnet man die Netzwerkeinstellungen, setzt einen Haken bei *Link- Local Only* im Tab IPv4 und klickt auf *Apply*.



Nun kann man den Chestbutton des Nao drücken, und der Roboter sollte seine IP-Adresse verraten (diese sollte man sich für später notieren!).

Den Nao vorbereiten

Wenn der Nao erfolgreich mit dem PC verbunden wurde und auf *ping* (ping <IP>) reagiert, kann man mit dem Prozess fortfahren.

Jetzt begibt man sich im Terminal in den Ordner *<Git Repo>/Install/* und führt das Skript *createRobot* aus. Dieses Skript hat noch ein paar Parameter, es ist folgendermaßen aufgebaut:

./createRobot -t <Teamnummer> -r <Nummer des Roboters> -i <IP-Adresse> <Name>

Der Parameter *-t <Teamnummer>* ist **optional**, falls nicht angegeben wird die Teamnummer aus *Config/settings.cfg* genommen (standardmäßig 89).

Es wird mit hoher Wahrscheinlichkeit 2x nach einem Passwort gefragt, dies ist normalerweise *nao*.

Weitere Infos gibt es in der Datei *CodeRelease2019.pdf* unter dem Punkt *2.3*.

Wenn dieses Skript erfolgreich ausgeführt wurde kann man mit dem nächsten Schritt beginnen.

Im gleichen Ordner befindet sich ein weiteres Skript mit dem Namen *installRobot*. Dieses wird mit *./installRobot <IP-Adresse>* ausgeführt.

Es wird sehr wahrscheinlich nach dem root-Passwort gefragt, dieses ist im Normalfall auch *root*.

Wenn der Befehl erfolgreich ausgeführt wurde, sollte der Roboter von selbst Neustarten. Nach dem Neustart wird er eine neue IP-Adresse nach dem Muster 192.168.<Teamnummer>.<Nummer des Roboters> haben (ACHTUNG! Die Nummern 0 und 1 sind für die Nummer des Roboters nicht zulässig).

Nun muss man seine Netzwerkeinstellungen verändern, um auf den Nao zugreifen zu können. Man geht wieder in die Einstellungen und setzt im Tab *IPv4* einen Haken bei *Manual*.

Unter *Address* gibt man eine IP im neuen Bereich des Roboters ein, also z.B. *192.168.<Teamnummer>.0* (ACHTUNG! Die letzte Ziffer darf nicht gleich mit der Roboternummer sein!).

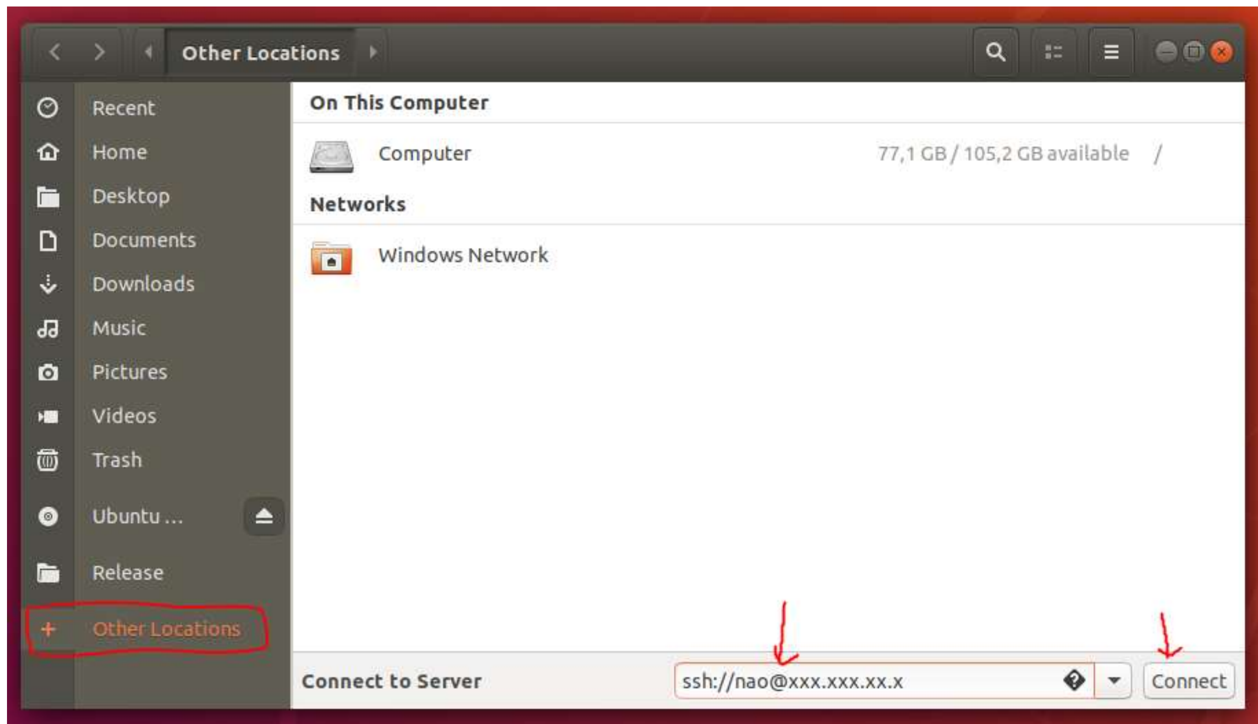
Unter Netmask gibt man *255.255.255.0* ein. Danach klickt man auf den grünen Button mit der Aufschrift *Apply.*

Jetzt muss die Verbindung mit 2 Klicks auf den On / Off Switch neugestartet werden, und man sollte sich wieder mit der neuen IP mit dem Nao verbinden können.

Network
Wired
Connected - 10 Mb/s
ON

Das Deployment vorbereiten

Zuerst muss man sich mit dem Dateimanager auf den Roboter verbinden. Dies funktioniert, indem man auf *Other Locations* auf der linken Seite des Dateimanagers klickt und dann unten bei Connect to Server die Adresse *ssh://nao@<IP-Adresse* eingibt. Danach kann man sich mit dem Connect-button verbinden.



Nach erfolgreicher Verbindung navigiert nach */home/nao/.*

Nun öffnet man sich noch ein zweites Dateibrowserfenster und navigiert zu */<Git-Repo>/Config* und kopiert alle Inhalte (ohne die Ordner /Keys, /Logs und /Scenes) nach */home/nao/Config* auf dem Roboter.

Jetzt kopiert man noch die Datei *bhuman* aus dem Ordner */<Git-Repo>/Install/Files/bin/* nach */home/nao/Config* auf dem Roboter.

Außerdem kopiert man noch die Inhalte aus */<Git-Repo>/Install/Network/Profiles/* nach */home/nao/Profiles* auf dem Roboter.

Den eigentlichen Code deployen

Man muss sich zuerst im Terminal in den Ordner */<Git-Repo>/Make/Common/* begeben, dort befindet sich ein Skript namens *copyfiles*.

Auch dieses Skript hat mehrere Parameter.

Der Aufbau ist *./copyfiles <Build, z.B. Develop> -nc -r <Nummer des Roboters> <IP-Adresse>*.

Das Skript hält den bhuman Dienst auf dem Nao selbständig an, setzt den Code auf dem Roboter auf und startet den Dienst wieder.

Dieser ist der einzige Schritt, der gemacht werden muss, wenn man den Code updaten will und die Skripte *createRobot* und *installRobot* schon auf dem Roboter gelaufen sind.

Nach einem Neustart des Roboters startet sich der Dienst selbstständig und der Roboter kann benutzt werden.

Durch drücken des Chestbuttons kann man durch die verschiedenen Modi der Software gehen.

Näheres gibt es in der Datei *CodeRelease2019.pdf* im Punkt *2.5*.

NAO-PS4-Gamepad Support

Im Zuge der Idee von „Teach-In“ Cards braucht es einen menschlichen Instruktor bzw. „Lehrer“, der gewisse Spielszenarien vorspielt, sodass NAO diese nacheifern kann. Dazu ist es aber erforderlich das ein Mensch den NAO im Simulator direkt steuern kann. Eine Steuerung ausschließlich über Konsolen Kommandos ist aufgrund der Komplexität und Vielfalt der Kommandos, als auch der Anforderung in Echtzeit reagieren zu können nicht empfehlenswert.

Daher wurde eine PS4-Gamepad Steuerung erstellt. Diese baut auf der allgemeinen Gamepad Steuerung von B-Human auf, aber verlagert das Key-Event Handling in einen Thread globalen Speicherraum und ermöglicht es so, ein Event-Handling innerhalb von Cards durchzuführen und dabei ganze highlevel Skills an die Controller Tasten zu binden.

1, 2, 3, 4, 5, 6, share, options, PS-home und das Touchpad sind derzeit ungenutzt.

7: dribbling Skill (work in progress)

8: findBall Skill

9: goAndKickBall Skill

10: Stoppen der momentanen Aktivität

11: umschalten zwischen max. Geschwindigkeit und default Geschwindigkeit

12: Bewegen { vorwärts gehen, links drehen, rechts drehen, rückwärts gehen }

13: umschalten zwischen angehoben und gesenktem Kopf (default: gesenkt)

14: Kopf drehen { links drehen, rechts drehen}

Pipeline: Grundgerüst für Regressionstests

Zum Benchmarken habe ich eine Methode im *GameController* namens *addBenchmarkLog()* hinzugefügt (Weitere Informationen auf Seite 12 dieses Dokuments).

Diese erzeugt eine Datei namens *benchmark_log.txt* in */<Git-Repo>/Config/Scenes/.*

Momentan wird die Zeit vom Starten des Matches bis zum ersten Ballkontakt und bis zum ersten Tor mitgeloggt.

Mein Vorschlag wäre, einen *GameController* Befehl zu Implementieren, der Global eine Variable setzt um das Benchmark ein- / auszuschalten (Mehr Informationen dazu auf den Seiten 10-11 dieses Dokuments).

Papierprototyp

Im Rahmen eines Sprints sollte ein Papierprototyp entworfen werden, um unsere prinzipielle Idee zur hybriden Daten/Regel basierten KI zu testen. Ziel war es Stärken wie auch Schwächen der Idee zu testen. Hierzu wurde eine CSV als Weltmodell angelegt, in der mehrere Datensätze hinterlegt waren. Jede Zeile beinhaltet Information über Sensor Daten und Kamerainformationen pro Frame. Die Daten wurden teils direkt aus der Simulationsumgebung entnommen und teils manuell nach Sinnhaftigkeit ergänzt.

Die CSV „SequenzFileOfWorldModelX“

Jeder NAO verwaltet sein eigens Weltmodell, wodurch die CSV seine subjektive Wahrnehmung repräsentiert. Aufbauend auf der CSV konnte eine erste Sequenzanalyse zum Testen durchgeführt werden.

Zu jedem Zeitpunkt kann das Kommando „gc write“ in der Konsole des Simulators aufgerufen werden, welches die CSV beim ersten Einsatz neu erstellt und direkt um eine neue Zeile mit Informationen erweitert. Bei jedem weiteren Aufruf wird die Datei um eine weitere Zeile ergänzt.

Die CSV besteht aus Folgenden Spalten:X-Pos NAO

- Y-Pos NAO
- Sichtwinkel NAO
- Ballkontakt
- Vorderster Mann
- Gegenspieler voraus (ja/nein)
- Gegenspieler voraus (X-Pos)
- Gegenspieler voraus (Y-Pos)
- Gegenspieler voraus (Sichtwinkel)
- Ball X-Pos
- Ball Y-Pos
- Imagename

Beispiel Datensatz für die Positionsdaten eines NAOs, der sich bewegt:

X-Pos NAO	Y-Pos NAO	Sichtwinkel Nao
-1.002.536.987	0.739979	-179.996.750
-1.199.997.681	-0.012890	-179.996.750
-1.199.997.681	-0.012892	-179.996.750
-1.415.416.382	-0.608894	-174.997.726

-1.415.416.382	-0.608857	-174.997.726
-1.415.416.382	-0.608857	-174.997.726
-1.615.417.725	-200.609.222	-174.997.421
-1.615.306.030	-201.950.638	-169.997.452
-1.615.306.030	-201.950.623	-169.997.452
-1.615.306.030	-201.950.623	-169.997.452
-1.915.306.152	-301.950.653	-169.997.421
-1.915.306.152	-301.950.653	-169.997.421
-2.415.412.354	-399.263.153	-179.997.452

Vollständige Datensätze finden sich unter \HSKL RoboCupGermanOpen 2021\Sequenzen\

Bekannte Probleme:

In manchen Szenarien wird im Controller nach der Ausführung der Kommandos ein „Syntax Fehler“ ausgegeben, jedoch wird die CSV dennoch korrekt beschrieben und es wurden keine Seiteneffekte beobachtet.

Die Szenarios

TODO: ???

Die Simulationsdaten

Die Simulationsdaten werden über dem „GameController“ entnommen, der direkten Zugriff auf die „SimulatedRobot“‘s hat. Dazu wurde der „GameController“ um eine Unterklasse „WorldModel“ erweitert. Diese Unterklasse speichert, verwaltet und schreibt die Daten, die die CSV von den Robotern benötigt. Somit ist jedem „WorldModel“ exakt ein „SimulatedRobot“ zugeteilt. Um Zugriff auf mehr Daten der simulierten Roboter zu besitzen, wurden Teile der Datenkapselung aufgebohrt und damit Teile der „SimulatedRobot“ Klasse, die zuvor *private* waren, nun als *public* deklariert.

Um nun die Daten zu lesen und zu schreiben stehen folgende Befehle an die Objekte der „WorldModel“ Klasse zur Verfügung.

- void initiateCSV();
  Einmaliges Ausführen; Erstellt die CSV und die Spaltenbezeichner. Überschreibt ggf. gleichnamige CSV Files.
- void update();
  Aktualisiert sämtliche Werte, die das „WorldModel“ speichert, und kommuniziert dazu mit dem „SimulatedRobot“.
- void writeNewCSVLine();

Schreibt die Daten als neue Zeile in die CSV.

- void GCcommandWrite();
  Alternativ kann dieses Kommando ausgeführt werden, um direkt (einmalig) zu initialisieren, zu aktualisieren und zu schreiben. Wird vom namensgebenden Controller Kommando „gc writeWorldModels“ genutzt.

Die Daten der Bilderkennung

TODO: ???

Simulator: Erstellen von Scenes

Im Simulator können Testsituationen mit sogenannten Scenes erstellt werden

Diese Scenes bestehen aus 3 verschiedenen Dateien:

- <name>.ros2 – Dieses Script repräsentiert die Visuelle Darstellung und die Anfangsposition des Balls

Training1vs2.ros2 777 Bytes

```
1  <Simulation>
2
3    <Include href="Includes/NaoV6H25.rsi2"/>
4    <Include href="Includes/1vs2.rsi2"/>
5    <Include href="Includes/Ball2016SPL.rsi2"/>
6    <Include href="Includes/Field2017SPL.rsi2"/>
7
8    <Scene name="RoboCup" controller="SimulatedNao" stepLength="0.012"
9      <Light z="9m" ambientColor="rgb(50%, 50%, 50%)"/>
10
11     <Compound name="teamColors">
12       <Appearance name="black"/>
13       <Appearance name="blue"/>
14     </Compound>
15
16      <Compound ref="robots"/>
17      <Compound ref="extras"/>
18
19     <Compound name="balls">
20       <Body ref="ball">
21         <Translation x="-1000mm" y="0mm" z="500mm"/>
22       </Body>
23     </Compound>
24
25     <Compound ref="field"/>
26
27   </Scene>
28 </Simulation>
```

Direkt am Anfang der .ros2 Datei können .rsi2 Scripts inkludiert werden, die Später die verschiedenen Elemente der Szene erstellen.

Weiterhin wird hier die Anfangsposition des Balls festgelegt.

- <name>.con – Dieses Script führt eine Sequenz von Befehlen in der Konsole des Simulators beim Ladevorgang aus.

📄 **Training1vs2.con** 105 Bytes

```
call Includes/Normal

# all views are defined in another script
call Includes/Views


cs HSKL_Team1 team1
```

Dieses Script kann verschiedene andere Scripte nachladen und Befehle in der Konsole ausführen.

- .rsi2 Dateien – Diese Scripte entscheiden wie viele und welche Roboter auf dem Feld stehen.

📄 **1vs2.rsi2** 490 Bytes

```
<Simulation>

  <Compound name="robots">
    <Body ref="Nao" name="robot4">
      <Translation x="-700mm" y="0" z="320mm"/>
      <Rotation z="180degree"/>
      <Set name="NaoColor" value="dblue"/>
    </Body>
    </Compound>
    <Compound name="extras">
    <Body ref="NaoDummy" name="robot1">
      <Translation x="-4300mm" y="0" z="320mm"/>
    </Body>
    <Body ref="NaoDummy" name="robot2">
      <Translation x="-2500mm" y="300mm" z="320mm"/>
    </Body>
  </Compound>

</Simulation>
```

Hier kann angegeben werden wie viele Roboter erstellt werden und ob diese Roboter

oder Dummies sind (Body ref="Nao" oder Body ref="NaoDummy"). Roboter werden im Compound „robots" hinzugefügt, Dummies unter „extras".

Das Tag „Translation“ gibt die Initialposition im Spielfeld an. Mit dem Tag „name“ wird der Name des Roboters mit der Nummer des Roboters angegeben, z.B. „robot4“.

Das Tag „Rotation“ gibt die Anfangsdrehung des Roboters, also die Blickrichtung an.

Simulator: Ein Szenario erstellen und einem Team zuweisen

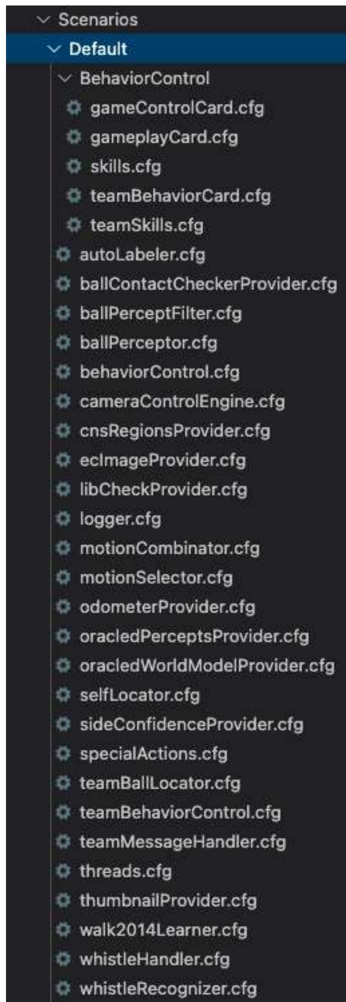
Eine Szene beschreibt die Testsituation im Simulator. Sie legt unter anderem fest, welche Objekte vorhanden sind und wo diese positioniert sind. Wie sich die Roboter nach Start der Simulation verhalten, beschreibt eine Szene nicht. Dazu wird ein Szenario benutzt, welches mit dem Aufruf einer Szene geladen wird.

Ein Szenario bestimmt das Verhalten aller Roboter oder eines einzelnen Team, denn dort kann man für verschiedene Spielzustände (z.B. ownKickoff, opponentKickoff) festlegen, welche Cards gespielt werden können (mehr zu Cards Skills & Cards oder CodeRelease2019.pdf Kapitel 6.1).

Der Aufruf des Szenarios geschieht über das Konfigurationsfile der Szene (<scene>.con). Wenn dort kein explizites Szenario angegeben ist, wird das Default Szenario geladen.

In der Konfigurationsdatei wird mit dem Befehl „cs <scenario> [team1|team2]“ ein Szenario geladen und einem Team zugeteilt. Die Angabe des Teams ist dabei optional. Wenn man darauf verzichtet, gilt das Szenario für beide Teams. (Mehr zu Konsolen Befehle CodeRelease2019.pdf Kapitel 10.1.6 oder in der Konsole des Simulator Befehl „help“ eingeben oder console_commands.txt).

Im Folgenden wird der Aufbau des Default Szenarios erläutert.



Ein Szenario besteht immer aus dem Unterordner Behavior Control und zahlreichen Konfigurationsdateien („autoLabeler.cfg“ …).

Die Konfigurationsdateien außerhalb des Behavior Control Ordners sind in jedem Szenario gleich und können mit Softlinks in ein neues Szenario eingebunden werden.

Die Dateien in Behavior Control werden ebenfalls benötigt. Dort werden die Cards angeben.

In der gameplayCard.cfg gibt es fünf Listen von Cards für die fünf unterschiedlichen Spiel-Zustände (Kick-Off, Strafstoß, … normales Spiel). Die Reihenfolge in der Liste priorisiert die Cards.

Skills & Cards

Zur Vorbereitung unbedingt CodeRelease2019.pdf im Kapitel 6.1 lesen: darin eine Einführung, ein Beispiel und weiterführende Erklärungen.

Zusammenfassung und Erläuterungen:

Für die Auswahl der richtigen *actions* und somit für die Steuerung des Verhaltens verantwortlich ist das Modul *Behavior Control*, welches aus *Skills* und *Cards* besteht.

Card werden in der gamePlayCard.cfg eingetragen. Es gibt dort 5 Listen von Cards für die fünf unterschiedlichen Spiel-Zustände (Kick-Off, Strafstoß, … normales Spiel). Die Reihenfolge in der Liste priorisiert die Cards.

Cards enthalten Pre- und Postcondition; je nach Wahrheitswert werden sie vom Dispatcher aufgerufen bzw. terminiert. Die Tests erfolgen mit der Frequenz des Behaviour Threads. Post-Condition ist normalerweise die Negation der Pre-Condition, sofern dies keine Invariante (wie die Rolle des Spielers) ist. Ein Beispiel

Pre-Condition: Spieler Nr == 4 (Goali) && Gegner ist nah an der Penalty-Zone

Post-Condition: Gegner ist nicht nah an der Penalty-Zone

(in der Karte wird der Torwart sein Verhalten entsprechend ausführen, z.B. den Winkel zum Gegner durch Hinauslaufen verkürzen)

Die genauen Zusammenhänge, sowie die Eigenschaft „sticky" sind in Abschnitt „6.1.2 Card" erläutert. Vereinfacht zusammengefasst: die am höchsten priorisierte Karte, deren Pre-Condition *true* ist wird gestartet. Sie wird verlassen, wenn ihre Post-Condition *true* wird.

Vermutung: wenn der Zustandsautomat keinen aktiven Zustand mehr erreicht, wird die Karte ebenfalls terminiert; dann kann sie ggfs. gleich wieder gestartet werden (mit dem inital_state). Das sollte noch überprüft werden!

Cards enthalten typischerweise einen Zustandsautomaten. Dieser wird durch einen Compiler (CABSL) übersetzt. Hinweis: dies ist eine Option – nicht alle Karten müssen so implementiert werden. Details s. Abschnitt „6.1.3 CABSL". Wichtige Eigenschaften dieses Zustandsautomaten

- der aktive Zustand bleibt zwischen zwei Aufrufen des Threads erhalten
- Zustands-Übergänge steuert die Card über Bedingungen

```
state (someState) {
        transition { if (cond) goto target state }
        acction {do something}
 }
state (targetState) {
…
}
```

- Einen exit-State braucht es nicht; das steuert der Dispatcher über die Post-Condition.

Cards sind Module im Sinne der Code Architektur von B-Human. Deren gegenseitige Abhängigkeiten (bspw. CALLS, PROVIDES, REQUIRES, DEFINES_PARAMETERS etc.) werden durch ein Interface verwaltet. Details s. Abschnitt „3.3.2 Module Definition".

Ein Beispiel;

```
CARD ( KickAtGoalCard ,
{,
CALLS ( GoToBallAndKick ),
REQUIRES ( BallModel ),
REQUIRES ( RobotPose ),
DEFINES_PARAMETERS (
{,
/** Distance of the ball to the goal to enable kicking . */
( float ) (3000. f) maxGoalDistance ,
}) ,
})
```

Das Makro MAKE_CARD(KicktAtGoalCard) wird für Module vom Typ Cards eingesetzt.

In behaviourstatus.h werden alle Behaviors verwaltet:

```
/**
 * @struct BehaviorStatus
 * A struct that contains data about the current behavior state.
 */
STREAMABLE(BehaviorStatus, COMMA public
BHumanMessageParticle<idBehaviorStatus>
{
  ENUM(Activity,
  {,
    unknown,
    fallen,
    finished,
    initial,
    gegnerUeberlaufen,
    set,
    ballsuche,
    codeReleaseKickAtGoal,
    codeReleasePositionForKickOff,
    steilpass,
  }
```

Mit dem Aufruf

theActivitySkill(BehaviorStatus::steilpass);

werden die Abhängigkeiten im laufenden System dynamisch verwaltet.

Skills werden von *Cards* aufgerufen – diese Unterscheidung dient der besseren Strukturierung, und wurden vor einigen Jahren von B-Human eingeführt. Ein *Skill* führt einen Request aus (typischerweise einen MotionRequest), eine Card nimmt keine Parameter entgegen, sondern definiert selbst welche und ruft damit Skills auf. Mehr Info dazu in Abschnitt „6.1.1. Skills“.

Einen Skill anlegen

Bei Skills handelt es sich um von Cards unabhängigen Aktionen. Sie können von jeder Card aufgerufen werden. Ihre Aufgabe liegt darin selbstständig bestimmte Aktionen auszuführen und kapseln auf diese Weise komplexere Abläufe. Da Skills keine Werte zurück liefern, bzw. immer *void* sind, konzentrieren sie sich auf Aktivitäten, deren Ergebnis für die aufrufende Card keine Relevanz in kommenden Entscheidungen hat. Skills sind in der Lage auch andere Skills aufzurufen. Zu beachten ist, dass es sich bei Skills um Datentypen handelt, ein *struct*, welches aus einem Makro generiert wird. Mehr Info dazu in Abschnitt „6.1.1. Skills“ der B-Human Dokumentation,

Erstellen eines Skills:

- Zunächst muss der Skill generiert werden durch das Nennen des Skills in der File „Skills.h“ mit…:
  SKILL_INTERFACE (<Skillname> (, (const value&) <ParamName>) *);
- ein Skill besitzt mindestens eine *execute*(const <SkillName>& p) Methode, die aufgerufen wird, um den Skill auszuführen.
- Der Parameter p beinhaltet alle übergeben Parameter als Attribute.
- In der Implementationsdatei des Skills am Ende ihn mit dem Makro MAKE_SKILL_IMPLEMENTATION(<SkillClassName>) registrieren.
- Multiple Implementierungen eines Skills sind möglich durch Überladung der *execute()* Methode. Rückgabewerte gibt es keine. Paramater sind alle *const ref* mit Ausnahme primitiver Datentypen.
- Die Code Konvention (nur für multiple Implementierungen) ist

```
skillImplementations = [
{ skill = SkillName ; implementation = SkillNameImp ;},
];
```

Ein Beispiel zum besseren Verständnis: theFelixSkill() (also keine Parameter).

Um den Skill anzumelden muss zuerst in der Implementierungsdatei das Makro SKILL_IMPLEMENTATION genutzt werden:

```
SKILL_IMPLEMENTATION(HSKL_OpponentDetectionImpl, // ← SkillNameImp
{,
  IMPLEMENTS(Felix),  //  ← SkillName
```

```
    REQUIRES(WorldModelPrediction), // ein beliebiges benötigtes Modul
});
```

Dann erfolgt die Implementierung des Skills in der Klasse

```
class HSKL_OpponentDetection : public HSKL_OpponentDetectionBase
// class SkillClassName : public SkillClassNameBase ; der Zusatz “Base” ist an dieser Stelle
// unbedingt erforderlich!
{
  void execute(const Felix& p) override {do Something};
// hier könnten weitere Implementierungen folgen; diese müssen dann jeweils
// unterschiedliche Signaturen haben. Der Name des Parameters ist der SkillName!
}
```

Abschließend muss der Skill registriert werden. (Nach dem Ende der Klassen Definition)

```
Make_SKILL_IMPLEMENTATION(HSKL_OpponentDetectionImpl); // ← SkillNameImp
```

Sobald das getan ist, muss nur noch der SkillName in „Skills.h“ hinterlegt werden, damit das zugehörige struct generiert werden kann. Ebenso werden an dieser Stelle die Parameter definiert wie oben angegeben.

Bsp. (Parameterlos) :

```
SKILL_INTERFACE(Felix);
```

Damit ist der Skill fertig implementiert und kann nachfolgend von anderen Skills und Cards genutzt werden, indem diese den Skill mit dem Makro CALLS(SkillName) zugänglich machen und danach mit theSkillName() aufrufen.

Bsp.:

```
CARD(…,
{,
    CALLS(Felix),
});
theFelixSkill();
```

Mit Parametern:

SKILL_INTERFACE(PassTarget, (int) passTarget, (const Vector2f&)(Vector2f::Zero()) ballTarget);

Beim Aufruf können fehlende Parameter – wie in C++ üblich – durch Default Werte ergänzt werden.

Ein Skill wie „PassTarget“ kann Auswirkungen nach außerhalb durch Seiteneffekte (hier: auf das Singleton theBevahiourStatus) erzielen:

```
class PassTargetImpl : public PassTargetImplBase
{
  void execute(const PassTarget& p) override
  {
    theBehaviorStatus.passTarget = p.passTarget;
    theBehaviorStatus.shootingTo = p.ballTarget;
  }
}
```

Das ist jedoch umständlich, weshalb Skills in der Regel doch nur als Prozeduren angewandt werden.

Troubleshooting

Auf Kommata bei den Makros achten und generell die Syntax beachten!

Unbedingt Includes beachten!

REQUIRES bzw. CALLS in der aufrufenden Card nicht vergessen.

Nach der Erstellung eines neuen Skills könnten IDE‘s noch Fehler anzeigen, da der Skill erst beim ersten Kompilieren erzeugt wird.

Eine Card anlegen

Beim Anlegen einer *Card* ist Folgendes zu beachten:

- Der Name einer *Card* muss „Card“ beinhalten. Beispiel: BallsucheCard.
- Die zugehörige Superklasse BallsucheCardBase

class BallsucheCard : public BallsucheCardBase

wird automatisch generiert. Warum das so ist: unklar.

- Man muss die *Card* im System anmelden. Dies geschieht in der Datei: /<GitRepo>/Src/Representations/BehaviorControl/BehaviorStatus.h. Dort im Enum activity muss die *Card* angegeben werden.
- Unsere Karten stehen im virtuellen Ordner Modules/BehaviorControl/BehaviorControl/HSKL/

Nachdem die ersten beiden Schritte erledigt sind kann man nun mit dem eigentlichen Implementieren beginnen.

```
#include "Representations/BehaviorControl/FieldBall.h"
#include "Representations/BehaviorControl/Skills.h"
#include "Representations/Configuration/FieldDimensions.h"
#include "Representations/Modeling/RobotPose.h"
#include "Tools/BehaviorControl/Framework/Card/Card.h"
#include "Tools/BehaviorControl/Framework/Card/CabslCard.h"
#include "Tools/Math/BHMath.h"

CARD(CodeReleaseKickAtGoalCard,
{,
  CALLS(Activity),
  CALLS(InWalkKick),
  CALLS(LookForward),
  CALLS(Stand),
  CALLS(WalkAtRelativeSpeed),
  CALLS(WalkToTarget),
  REQUIRES(FieldBall),
  REQUIRES(FieldDimensions),
  REQUIRES(RobotPose),
  DEFINES_PARAMETERS(
  {,
    (float)(0.8f) walkSpeed,
    (int)(1000) initialWaitTime,
    (int)(7000) ballNotSeenTimeout,
    (Angle)(5_deg) ballAlignThreshold,
    (float)(500.f) ballNearThreshold,
    (Angle)(10_deg) angleToGoalThreshold,
    (float)(400.f) ballAlignOffsetX,
    (float)(100.f) ballYThreshold,
    (Angle)(2_deg) angleToGoalThresholdPrecise,
    (float)(150.f) ballOffsetX,
    (Rangef)({140.f, 170.f}) ballOffsetXRange,
    (float)(40.f) ballOffsetY,
    (Rangef)({20.f, 50.f}) ballOffsetYRange,
    (int)(10) minKickWaitTime,
    (int)(3000) maxKickWaitTime,
  }),
});
```

Nach den Include Angaben folgt ein Enum mit dem Namen der Card. Dort werden mit CALLS*(Skill)* ein Skill angegeben, welcher von der Card aufgerufen wird. Ebenso kann man mit REQUIRES*(Representation)* eine Representation angegeben werden, die dann innerhalb der Card verwendet werden kann (z.b. RobotInfo, um mit der teamNumber arbeiten zu können).

Dann definiert man Parameter, die später beim Aufruf der Skills übergeben werden.

```
class CodeReleaseKickAtGoalCard : public CodeReleaseKickAtGoalCardBase
{
  bool preconditions() const override
  {
    return true;
  }

  bool postconditions() const override
  {
    return true;
  }
```

Nun wird die Klasse definiert, diese erbt von (CardBase?).

```
  void execute() override
  {
    theActivitySkill(BehaviorStatus::codeReleasePositionForKickOff);
    theLookForwardSkill();
    theStandSkill();
    // Not implemented in the Code Release.
    theSaySkill("Please implement a behavior for me!");
  }
};

MAKE_CARD(CodeReleasePositionForKickOffCard);
```

Am Ende wird mit MAKE_CARD(BallsucheCard) das Makro aufgerufen, der die *Card* anmeldet.

Wenn man eine Card mit Zustandsautomat erstellt, folgt noch den pre- und postcondition nicht die execute() Funktion, sondern es werden die states angeben und definiert.

```
bool postconditions() const override
{
  return true;
}

option
{
  theActivitySkill(BehaviorStatus::codeReleaseKickAtGoal);

  initial_state(start)
  {
    transition
    {
      if(state_time > initialWaitTime)
        goto turnToBall;
    }

    action
    {
      theLookForwardSkill();
      theStandSkill();
    }
  }

  state(turnToBall)
  {
    transition
    {
      if(!theFieldBall.ballWasSeen(ballNotSeenTimeout))
        goto searchForBall;
      if(std::abs(theFieldBall.positionRelative.angle()) < ballAlignThreshold)
        goto walkToBall;
    }

    action
    {
      theLookForwardSkill();
      theWalkToTargetSkill(Pose2f(walkSpeed, walkSpeed, walkSpeed), Pose2f(theFieldBall.positionRelative.angle(), 0.f, 0.f));
    }
  }
```

Ein state besteht immer aus *transition* und *action.* Dabei wird *action* vor *transition* ausgeführt.

Am Ende von option{} kann man Funktionen definieren, die in den *states* verwendet werden können.s

```
  Angle calcAngleToGoal() const
  {
    return (theRobotPose.inversePose * Vector2f(theFieldDimensions.xPosOpponentGroundline, 0.f)).angle();
  }
};

MAKE_CARD(CodeReleaseKickAtGoalCard);
```

Troubleshooting

Um Karten zu löschen müssen sie auch im output ordner gelöscht werden.

Teach-In Cards

Die Erweiterung „Teach-In Cards“ des B-Human Konzeptes „Cards“ ist eine konzeptionell eigenständige Entwicklung im Projekt. Technisch weichen sie von den bestehenden Cards nur minimal ab. Bestehende Cards müssen nicht verändert werden. Funktional gibt es große Unterschiede, die im Folgenden dokumentiert werden:

Konzeption

Die Klasse *CardBase* in CardBase.h wurde um die virtuelle Funktion

unsigned int teachin_score() ()

erweitert. Ihre Default-Implementierung liefert den Ergebniswert null. Diese Funktion dient der Priorisierung (s.u.) und als check, ob eine vorliegende Karte konventionell (teachin_score() == 0) oder eine Teach-In Card (teachin_score()>0) ist.

Im Laufe der weiteren Entwicklungen des Projektes wird diese Funktion den aktuellen Spielstand, Spielfeld, Position usw. des Robots auswerten, und die Nähe zu einem vorher definierten Spielszenario berechnen. Der Aufruf erfolgt in der Klasse *PriorityListDealer*, die im Behavior-Modul einmal pro Zyklus mit der Methode *deal()* überprüft, welche Card aktiviert werden soll. Aktuell (12/2020) wird für den proof-of-concept die Berechnung der Spielerrolle und die Nähe zu einem Aktivierungspunkt (hier genannt: triggerPosition) über einen radialen Abstand verwendet.

```
class OffenseStrikesShortPassTICard : public OffenseStrikesShortPassTICardBase

{

  unsigned int teachin_score() const overrideconst float

  dist = Geometry::distance(triggerPosition, theRobotPose.translation);

    if((5 == theRobotInfo.number || 4 == theRobotInfo.number) && dist < triggerThreshold) {

      return 1;

    }

    return 0;

  }}
```

Die Spielszenarien werden beim Teach-In manuell definiert, die dabei geltende Welt-Information – ein Schnappschuss des Spielgeschehens mit den Positionen der Spieler, des Balls, Spielstand usw. - wird im Gamecontroller mit „gc write“ in eine .CSV Datei exportiert. Die Idee dahinter ist es, dass alle verfügbaren Teach-In Cards vom Dispatcher (PriorityListDealer) zur Spielzeit überprüft werden, und die am besten qualifizierte Teach-In Card (größter Wert beim teachin_score()) aktiv gesetzt wird („gedealt wird“). Damit entfällt die Notwendigkeit im Projekt,

auf Basis der konventionellen Karten und ihrer top-down Priorisierung eine komplizierte Steuerung über die precondition und postcondition zu realisieren.

In einer späteren Implementierung können – bei mehr als einer qualifizierten Teach-In Card – diese pseudo-zufällig, verteilt nach ihren relativen Werten, gedealt werden. Dazu würde dann ein Schwellwert erforderlich werden, der experimentell ermittelt werden muss.

Der neue Dispatcher (erweiterter PriorityListDealer)

Der bestehende Dispatcher PriorityListDealer verfolgt einen linearen Ansatz, d. h., die im System angemeldeten Karten werden nach absteigender Priorität auf Qualifikation (precondition ist true) überprüft; die TorabschlussCard hat im Beispiel die höchste Priorität, und wird immer als erstes überprüft, dann die HinterBallPositionierenCard usw.

```
normalPlay = {
 sticky = false;
 cards = [
  TorabschlussCard,
  HinterBallPositionierenCard,
  OffenseDefaultBehaviorCard
 ];
```

Das geschieht in CardBase::deal() so:

```
 for(size_t i = 0; i < deck.cards.size(); ++i) {
      CardBase* card = deck[i];
        if((card == lastCard) ? (!card->postconditions() || card->preconditions()) : card->preconditions()) {
            nextCard = card;
[…]
}};
```

Die im vorherigen Abschnitt umrissene Semantik sieht in der Implementierung konkret so aus (der folgende Abschnitt steht ***vor*** dem obigen Code Abschnitt)

```
// loop over all cards, see above
 if( thisScore > 0) {  // we see the first "fresh" TICard in this round
        if(card != lastCard || // TICards do not kick-out their buddies!
```

```
      card->postconditions()) {  // may be we can re-deal the last card


      // now we scan for all of them, if there are any. First regular card stops scanning
      for(size_t j = i; j < deck.cards.size() && (thisScore = deck[j]->teachin_score()) > 0 ; ++j) {
        i = j;
        if (thisScore >= maxScore) { // we found a new top scorer
          card = deck[j];
          maxScore = thisScore;
        }
      }
    }
    nextCard = card;
// cont. with code from above
```

Im Unterschied zu den konditionellen Cards ist die aktuelle Einstellung, dass eine Teach-in Card eine andere Teach-in Karte nicht verdrängt (siehe Codezeile …). Diese Einstellung ist gegebenenfalls zu überprüfen.

Damit lässt sich ein Mischbetrieb realisieren. Im Folgenden sieht man eine gameplay.cfg Datei, die folgende Spielesteuerung realisiert:

1. TorabschlussCard predondition: ist der Bot in Nähe des Torraums, und im Ballbesitz werden alle anderen Karten verworfen, und der Torschuss angestrebt
2. Auf der zweiten Prioritätsstufe werden im Beispiel alle (hier: zwei) Teach-in Karten überprüft. Im Branch „teachIn_POC" sind dafür zwei Trigger-Punkte in der Nähe der Aufstellpositionen für die angreifende Mannschaft prototypisch implementiert (Passgeber, Passempfänger, in etwa ab Mittelkreis).
3. Auf der dritten und folgenden Prioritätsstufe werden weitere konventionelle Karten überprüft.
4. Gemäß unserer Konvention ist die Default-Card auf dem untersten Prioritätsniveau, mit der precondition „true"

Aus der gameplay.cfg (Syntax ist von B-Human so vorgegeben)

```
  TorabschlussCard,
```

OffenseStrikesShortPassTICard,

OffenseRunsDownPassTICard,

HinterBallPositionierenCard,

OffenseDefaultBehaviorCard

wird damit semantisch

TorabschlussCard,

best-of(OffenseStrikesShortPassTICard, OffenseRunsDownPassTICard,)

HinterBallPositionierenCard,

OffenseDefaultBehaviorCard

, also eine lokale Breitensuche innerhalb einer linearen top-down Suchschleife.

Die Laufzeit der Cards folgt dem Muster für die konventionellen Cards, d. h., eine Karte terminiert, wenn ihre postcondition zu true evaluiert. Bei den Teach-In Karten empfiehlt es sich, diese postcondition an die Spielzeit der eintrainierten Teach-In Sequenz zu koppeln; siehe dazu den Beispielcode der beiden Karten ---TICard in diesem Branch.

```
bool exit_card = false;

bool postconditions() const override {

   return exit_card;  // is set as exit_card = (state_time > 3000);, i.e., card runs for 3sec

 }
```

Und später im Code des Zustandsautomaten (dieser setzt die state_time je Zustand hoch):

```
exit_card = (state_time > 3000);
```

Aufzeichnen:

Beim Aufzeichnen einer Teach-In Sequenz durch den Instruktor folgt er dieser Reihenfolge:

(init) (Wiederholbarer) Aufbau Feld mittels .con und .rsi2, ggfs. mv Befehle, und/oder manuelles Positionieren; bei Bedarf Simulator einige Sekunden laufen lassen, damit die Bots ihre Position ziemlich genau kennen

(pre-start) über einige Sekunden Dauer hinweg: mehre „gc write“ auslösen (diese dienen als Negativ-Beispiele; hier soll die TI-Card _noch_ nicht auslösen; mit time stamp

(start) nach Erreichen der Start Position: gc write (dies ist in diesem Durchgang das einzige Positiv Beispiel; mit time stamp

(teach-in) Kommandos - manuelle Steuerung durch Joystick (Controller) und/oder key-press Events (zum Abfahren von Skills) – werden aufgezeichnet, mit time stamp

(end) gc write

Konventionen:

- das letzte gc write _vor_ Beginn der teach-in Kommandos definiert die „trigger Position“, d.h. die x,y Koordiniation des aktiven Bots _inklusive_ aller vorhandenen worldmodell Informationen

- alle Feld-Positionen werden beim Data Mining relativ zur Position des aktiven Bots gemessen (Bsp.: Abstand des Balls zum Spieler)

- keine Aktion – keine Aufzeichnung – variable zeitliche Abstände zwischen den „Zeilen“ der Aufzeichnung

(Schleife); multiples Wiederholen der obigen Sequenz, wobei in (pre-start) lokale Variationen erfolgen müssen (Data Mining Lerner soll nicht over-fitten!)

Welche Daten sind relevant: wissen wir noch nicht; Aufzeichnen von

- Bezeichnung der TI-Cards, # der Wiederholung

- welcher Bot ist aktiv (ist am Joystick),

- ground-truth (Simulator Daten) Positionen aller Feldspieler

- Ballposition, Bewegungsvektor falls möglich

- (optional) Spielstand, Zeit und Spielhälfte, Zustand Ball (ruht/in Besitz

- (optional) Bewertung des (end) Spielfeldes

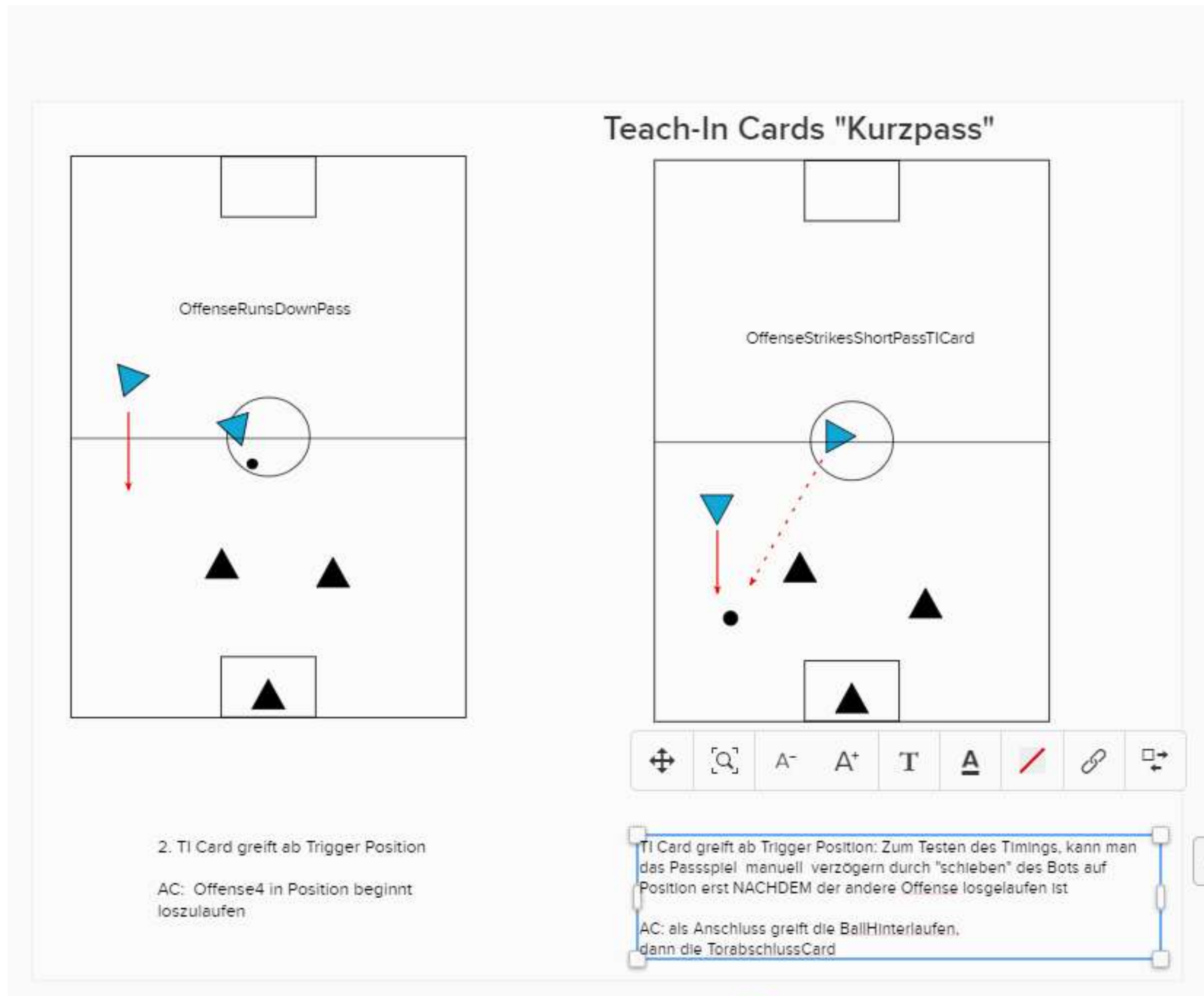
- (optional) Bewertung durch den Automatic Referee

Teach-In Cards proof-of-concept

Im Branch **teachInPOC**

5a1066e8 · final code for phase a (hard wired trigger) and b (playback sequence sample)

Sind folgende zwei taktische Szenarien implementiert. Beide Cards verwenden die B-Human state-machine (CABSL). Durch Verwendung von state_time lassen sich bspw. Pausen realisieren (Warten, damit Mitspieler in Position laufen kann).

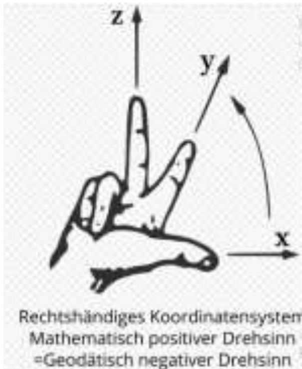


Best-practices für den B-Human Code

Die folgenden Abschnitte entstanden im November 2020, während Sprint 2, aus eigenen Erfahrungen im Team und Q&A mit Arne Hasselbring, langjähriger Entwickler und Team Leiter im B-Human Team

Das Feld: Koordinaten, Aufsetzpunkte, Roboternummer

Für Team Black, spielt von links nach rechts, gilt:



Rechtshändiges Koordinatensystem
Mathematisch positiver Drehsinn
=Geodätisch negativer Drehsinn

- Aufsetzen des Bots in der eigenen Hälfte an der Außenlinie
- Empfohlene Translation: s. 5vs5.rsi2
- Die x,y,z Koordinaten entsprechend der drei-Finger Regel eines rechts-händischem Koordinatensystem, d.h. mit zunehmendem x kommt der Bot in Richtung gegnerisches Tor, zunehmendes y geht zum linken Spielfeldbereich
- 1 Goali
- 2,3 left/right Defense
- 4,5 left/right Offense
- Teamerkennung: Team 1 = bots #1 – 5, Team 2 = bots 6..11 (in der .rsi2 Datei) Referenz: 5vs2.rsi2

x ...-3..-.2..-.1...0..+1..+2...3

--------1--2 --3 ---|---4--5 ------ **y = +3**

|

|

Goal

Goal o

Goal

|

|----------5--4 -----|---3--2--1 --- **y= -3**

Für Team Blue werden die x und y Koordinaten gespiegelt.

ACHTUNG: im Labor muss – wg. des nur partiellen Spielfelds – mit dem Flag „alwaysAssumeOpponentHalf“ gearbeitet werden. Dieses stört im Simulator das Weltmodell massiv, die bots invertieren ihre Koordinaten beim Überqueren der Mittellinie.

Tipps für den Simulator

- Die drei Views Fast, Normal, und PerceptOracle ergeben sehr unterschiedlich konfigurierte Laufzeitsysteme der Bots.
- Wenn man im Simulator nur Verhalten und weder Bildverarbeitung noch Modellierung testen will, bietet es sich auch an, eine der Fast-Szenen zu benutzen, oder wenn man eine eigene Szene erstellt hat, ist in die entsprechende „.con“-Datei statt "call Includes/Normal" "call Includes/Fast" zu schreiben. Dann sind auch Lokalisierungsfehler ausgeschlossen, weil die RobotPose direkt vom Simulator geliefert wird.
- PerceptOracle View ist sehr hilfreich, um Lokalisierungsfehler direkt sehen zu können
- Falls nicht debuggt werden kann, kann man mit dem Makro OUTPUT(idText, text, "Nachricht"); eine Nachricht in die Simulator-Konsole abgesetzt werden
- Falls man aus dem Code heraus Befehle im Simulator ausführen möchte, geht das mit OUTPUT(idConsole, text, "Befehl");

Konzeption Behaviour, Card, Skill, Sensoren

- Das Team tauscht Informationen aus. TeamData ist dezentral, TeamPlayersLocator und TeamBallLocator werden abgestimmt. Kritisch ist die geringe update Frequenz (1/sec).
- Im Nachrichtenverkehr zwischen den Bots sind 474bytes freies Datensegment übrig. B-Human setzt auf ein Kompressionsverfahren mit NTP ähnlicher Synchronisation
- thePathToTargetSkill verwendet absolute Koordinaten und ruft den PathPlanner. theWalkToTargetSkill verwendet robot-relative Koordination (beachtet aber nicht den Drehwinkel des Bots)
- Wie kann man in bestimmten Schusswinkeln schießen ohne um den Ball rotieren zu müssen? Das muss man selbst programmieren (B-Human hat es entfernt).
- Die Methode "getAbsoluteBallPosition" des Interface "SimulatedRobot" liefert die Position des Balls im Szenario
- Das Interface "SimulatedRobot" bietet Pointer, die im Zusammenhang mit Methoden genutzt werden, um verschiedene Daten zu lesen. Es gibt zwar Pointer für die US Sensoren, aber keine Methoden um diese Werte zu erhalten. Der Pointer selbst liefert nur QT Informationen (widget, icon,...)
- Freund/Fein Erkennung ist schwierig. In welchem Team bin ich (beide Varianten): In PlayersPerceptors gibt es bereits die Funktion "detectJersey(...)" als Nachverarbeitung. Diese setzt, je nach erkannter Farbe des Jerseys, den "Type" eines Obstacles in ObstaclesFieldPercept auf unknown, ownPlayer, oder opponentPlayer.
- Representation: Perception/ObstaclesPercepts/ObstaclesFieldPercept "This file implements a representation that lists the obstacles that were detected in the current image in robot-relative field coordinates."

Worldmodel, RobotPose, Odometrie, Vision, Representations

- Der WorldModelPredictor ist ein Hilfsmodul für RoboterPose, um aktuelle Schätzungen (Bsp.: Ball bewegt sich) zu antizipieren. In der WorldModelPrediction steht das Ergebnis aus der Lokalisierung aus dem letzten Zeitschritt plus die Odometriedifferenz seitdem, während in der RobotPose auch schon die neuesten Daten aus der Bildverarbeitung und Spielzustandsübergänge (penalized), berücksichtigt sind. Wenn man also nicht gerade an einem Modellierungsmodul arbeitet, welches laufen muss, bevor die SelfLocator läuft (z.B. weil seine Ausgabe als Eingabe des SelfLocators gebraucht wird), sollte man die RobotPose nutzen
- Was muss man ggfs. selbst programmieren, wenn wir die Bildanalyse austauschen? Klassifizierung des Balls, und Detektion Roboter. Linie, Kreis, Penalty sind algorithmisch (ScanLine), und bleiben verfügbar
- CompiledNN ist schneller als TensorFlowLite bemi ball classifier; bei geringerem Anzahl an Netzformen und Topologien (#convolutions). Keras Model als Input geht
- Die Odometrie ist die von der Bewegungssteuerung zurückgelieferte Distanz, die der Roboter zurückgelegt hätte, wenn es keine Effekte wie ungenaue Gelenkansteuerung und Durchrutschen auf dem Boden gäbe
- Bei der Bearbeitung von Representations mit dem Makro STREAMABLE keine template Typen verwenden, da diese erst nach dem Makro ermittelt werden. => führt zu komplexen Compile Errors

Nützliche Codesnippets

Im Header „Tools/Math/Geometry.h" befinden sich verschiedene Mathematikfunktionen, z.B. für Vektormathematik:

- Die Funktion angleTo() um einen Winkel zwischen 2 Vektoren zu berechnen
- Die Funktion distance() um den Abstand von 2 Vektoren zu berechnen
- Verschiedene Funktionen um Punkte zu prüfen („isPointInsideRectangle()", „isPointInsidePolygon()", etc.)

Über den Pfadplaner gibt es im Team Report etwas (S. 109-110). B-Human verwendet einen High-Level-Skill, der WalkToPoint heißt und abhängig von der Entfernung zum Ziel PathToTarget oder WalkToTarget nutzt, wobei für WalkToTarget vorher noch ein modifiziertes Ziel berechnet wird, welches verschoben wird, wenn Hindernisse im Weg sind.

Der „thePathToTargetSkill":

- Dieser Skill verwendet den Path-Planner, um den Roboter zu einer Absoluten Position auf dem Feld zu bewegen, dabei werden eventuelle Hindernisse umlaufen.
- Der Skill lässt den Roboter auch auf die Endposition schauen

Der „theWalkToTargetSkill":

- Dieser Skill lässt den Roboter zu einer relativen Position zu sich selbst laufen
- Der Nao kann dabei auch gedreht werde, wenn im 2. Parameter ein Winkel in der Pose2f angegeben wird
- Beispiel

```
theLookForwardSkill();
theWalkToTargetSkill(poseWalkSpeed, Pose2f(90.0f, 100.f, 500.f));
```

Dieses Beispiel bewegt den Nao um 100mm nach vorne, 500mm nach rechts und dreht ihn um 90° nach rechts (alles robot-relativ)

Der „theLookAtPointSkill":

- Dieser Skill lässt den Roboter per Kopfdrehung auf einen Punkt schauen
- Der Punkt wird als 3D-Vektor angegeben, wobei z die Höhe ist
- Die Koordinaten des Punkts werden in mm angegeben
- Beispiel:

```
const Vector3f desiredPoint(theFieldDimensions.xPosOpponentGoal, 0.0f,200);
// z= 20cm über Boden
theLookAtPointSkill(desiredPoint);
theStandSkill();
```

Das Interface „theFieldBall“:

- Dieses Interface hat viele verschiedene Informationen über den Ball; die meisten der Variablen des Interfaces sind im Code dokumentiert.
- theFieldBall.intersectionPositionWithOwnYAxis gibt die Position relativ zum Nao an, an der der Ball am Nächsten am Körper vorbeirollt (min. Radius für Tangente).
- theFieldBall.timeUntilIntersectsOwnYAxis gibt die Zeit an, die der Ball braucht, bis er am Nao diesen Punkt passiert. Passiert dies nicht wird diese auf float::max() gesetzt.

Das Interface „theRobotPose“

- Dieses Interface beinhaltet Informationen über den Zustand des Roboters
- theRobotPose.translation gibt die absolute Position des Nao im Spielfeld.
- theRobotPose.inversePose gibt die invertierte Pose des Nao an (Rotation + Position). Anwendung: Winkel-berechnung (Berechnung eines gewünschten Drehwinkels x/y´, bspw. wie weit drehen in Richtung Ball.

Deprecated

Dieser Abschnitt dient als Archiv und enthält Einträge im Laborbuch, die nicht mehr von Nutzen oder veraltet sind und daher entfernt wurden.

“Erstellung einer Objekterkennung” (Version von Salome Schlemer)

Der NAO Roboter soll mithilfe von künstlicher Intelligenz lernen, einen anderen NAO, ein Tor oder einen Ball zu erkennen. Die Erkennung erfolgt über die Aufnahmen der drei eingebauten Kameras links und rechts, sowie die nach unten gerichtete Kamera. Mithilfe dieser Aufnahmen kann der NAO seine Umgebung wahrnehmen, allerdings fehlt hier noch die Interpretation des Gesehenen.
Dazu soll die Erstellung eines Modells zur Objekterkennung beitragen. Dieses wurde im Folgenden mithilfe von Tensorflow[1] und Keras[2] konzipiert.
Die Abbildung 1 zeigt den Ausgangspunkt des Modells, von dem aus es weiter verbessert werden soll.

```
model = Sequential()

model.add(Conv2D(16, (3, 3), padding='same', input_shape=(416, 416, 3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, (3, 3), padding='same', activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), padding='same', activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(128, (5, 5), padding='same', activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(128, activation="relu"))
model.add(Dropout(0.1))
model.add(Dense(7, activation="relu"))

# Show info and compile
model.summary()
model.compile(loss='mse', optimizer="adam")
```

Abbildung 1 Ausgangsmodell zur Objekterkennung

[1] https://www.tensorflow.org/

[2] https://www.tensorflow.org/api_docs/python/tf/keras

Mit diesem Aufbau wurden die ersten Versuche zur Verbesserung durchgeführt. Diese sind in **Fehler! Verweisquelle konnte nicht gefunden werden. (TODO: Fix Verweis)** dokumentiert und anschließend mit Begründungen zu den jeweiligen Änderungen versehen.

Zuvor erfolgt aber, zum besseren Verständnis, eine kurze Übersicht über die verwendeten Begriffe. Aufgrund der besseren Wiedererkennbarkeit im Code werden für die Dokumentation der Werte die englischen Begriffe verwendet.

Erläuterungen zum Modell

Tensor

Ein Tensor ist eine mathematische Funktion, die eine bestimmte Anzahl von Vektoren auf einen Zahlenwert abbildet.[3]

Ableitung des Begriffs Tensorflow

Class Sequential[4]

Ein sequenzielles Modell eignet sich für einen einfachen Stapel von Schichten, bei dem jede Schicht genau einen Eingangstensor und einen Ausgangstensor hat.

Die Klasse Sequentiell gruppiert daher einen linearen Stapel von Schichten zu einem tf.keras.Modell.

Class Model

Model gruppiert Schichten zu einem Objekt mit Trainings- und Inferenzmerkmalen.

Inferenz ist der Prozess, durch den die Modelle mit den Daten verglichen werden. Dazu gehört normalerweise, das Modell mathematisch zu gießen und die Wahrscheinlichkeitsprinzipien anzuwenden, um die Qualität der Übereinstimmung zu quantifizieren.

Conv2D

Übersetzt: 2D Convolutional Layer = Faltungsschicht

Hier geschieht eine räumliche Faltung über die Bilder nach den angegebenen Werten.

MaxPooling2D

Verringert die Eingabedarstellung, indem der Maximalwert über das durch pool_size definierte Fenster für jede Dimension entlang der Feature-Achse genommen wird. Das Fenster wird in jeder Dimension um eine Schrittweite verschoben.

Flatten

Verflacht die Eingabe. Hat keinen Einfluss auf die batch_size.

Dense

Dient dazu die Matrizen auf eine einzelne Zahl als Ausgabe zu reduzieren.

---

[3] https://de.wikipedia.org/wiki/Tensor

[4] https://www.tensorflow.org/api_docs/python/tf/keras/Sequential

Dropout
Die Dropout-Schicht setzt die Eingabeeinheiten nach dem Zufallsprinzip auf 0 bei jedem Schritt während der Trainingszeit, was dazu beiträgt, eine Überanpassung zu verhindern. Eingaben, die nicht auf 0 gesetzt werden, werden um 1/(1 - Rate) hochskaliert, so dass die Summe über alle Eingaben unverändert bleibt.

Erläuterungen zu den Trainingsparametern

Epochs[5]
In eine Epoche läuft der gesamte Datensatz einmal vorwärts und rückwärts durch das neuronale Netzwerk.

Batch_Size
Gesamtzahl der Trainingsdaten die in einem Stapel vorhanden sind. Das bedeutet die Aufteilung der Daten, welche in einer Epoche durch das Netz laufen. Damit werden in jeder Epoche nur dieser Teil der Trainingsdaten gelernt.
Batch_Size = 1 entspricht daher allen Trainingsdaten.

Validation Split
Aufteilung der Daten in Trainingsdaten und Testdaten. Mithilfe der Trainingsdaten wird anhand der Labels gelernt was z.B. ein NAO ist. Anhand der Überprüfung mithilfe der Testdaten passt das Netz die jeweiligen Verknüpfungen an.
Validation Split = 0,3 entspricht bei 100 Daten = 70 Trainingsdaten und 30 Testdaten

Loss
Je geringer der Verlust (Loss), desto besser ein Modell (es sei denn, das Modell hat sich zu sehr an die Trainingsdaten angepasst). Der Verlust wird auf der Grundlage von Training und Validierung berechnet, und seine Interpretation gibt an, wie gut das Modell für diese beiden Sätze abschneidet. Im Gegensatz zur Genauigkeit (Accuracy) ist der Verlust kein Prozentsatz. Er ist eine Summierung der Fehler, die für jedes Beispiel in Trainings- oder Validierungssätzen gemacht wurden.

Accuracy
Die Genauigkeit für z.B. eines Klassifikationsalgorithmus für maschinelles Lernen, ist eine Möglichkeit zu messen, wie oft der Algorithmus einen Datenpunkt korrekt klassifiziert. Die Genauigkeit ist die Anzahl der korrekt vorhergesagten Datenpunkte von allen Datenpunkten.

Val_Loss
Validation Loss = Validierte Verlustfunktion

---
[5] https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9

Val_loss gibt die Gesamtzahl der Fehler der Testdaten aus. Das bedeutet der validierten Daten des Netzes. Das Ziel ist es den loss und val_loss zu minimieren auch um Overfitting zu vermeiden.

Val_Accuracy
Validation Accuracy = Validierte Genauigkeit
Die Testdaten werden auf die Genauigkeit hin überprüft. Stimmt beispielsweise die Klasse Nao mit dem Bild Nao überein, geht die Genauigkeit hoch.

	Änderung	**Parameter Training**			**Ergebnis (Angabe letzte Epoche)**			
		Epochs	Batch Size	Val_ Split	Loss (Fehler)	Accuracy (Prozent)	Val_ Loss	Val_ Accuracy
1	Keine Änderung, Test Ausgangsmodell	50	1	0,3	522,2696	0,8654	1376,1442	0,7510
2	Ergänzung eines weiteren Layers	10	1	0,3	1474,1371	0,6704	1533,0802	0,6570
3	Erhöhung der Epochen	50	1	0,3	1459,6091	0,6704	1503,2593	0,6570
4	Loss = categorical crossentropy	20	1	0,3	486,4404	0,8388	530,9670	0,7704
5	Loss bleibt, letzter Layer wird wieder entfernt (Ausgangsmodell)	20	1	0,3	485,5236	0,8483	545,8169	0,7372

Tabelle 1 Durchführung der ersten Versuche zur Verbesserung

Versuch 1

Allgemeines Testen des Ausgangsmodells, daher keine Veränderungen.

Versuch 2

Ergänzung um einen Layer wie folgt:

```
model.add(Conv2D(256, (5, 5), padding='same', activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
```

Änderung der Epoche auf 10 für eine schnelle Überprüfung, ob eine Verbesserung eingetreten ist.

Versuch 3

Erhöhung der Anzahl der Epochen aufgrund, der Verschlechterung des vorherigen Versuchs.

Versuch 4

Änderung der Loss-funktion von „mse“ auf „categorical crossentropy“

Mse: „Mean Square Error“ ist die Summe der quadrierten Abstände zwischen der Zielvariablen und den vorhergesagten Werten.

Categorical Crossentropy[6]: Berechnet den Kreuzentropieverlust zwischen den Labels und Vorhersagen. Verwendung dieser Funktion des Kreuzentropieverlusts, wenn es zwei oder mehr Label-Klassen gibt.

Idee ist es, durch diese Anpassung andere Gewichte zu trainieren, da das Netz anhand des loss angepasst wird.

Änderung der Epochen auf 20, da 10 für eine Aussage zu wenig ist und 50 erheblich länger dauert.

Versuch 5

Änderung auf das Originalmodell, Loss bleibt bei Categorical crossentropy.

Fazit der ersten Durchläufe

Durch die Einstellung des Models auf 7 Ausgabeparameter wurde das Netz vermutlich falsch trainiert.

```
model.add(Dense(7, activation="relu"))
```

Die 7 steht an dieser Stelle für die 4 Koordinaten, welche die Bounding Box aufspannen, sowie die 3 Klassen „NAO, Ball und Tor“. Aufgrund der erwarteten Ausgabe von 7 möglichen Klassifizierungen, kann das Netz keine Unterscheidung zwischen den Klassen und den Bounding Boxen treffen. Daher müssen an dieser Stelle Anpassungen vorgenommen werden.

[6] https://keras.io/api/losses/probabilistic_losses/#categoricalcrossentropy-class