Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Thomas Drauschke

Hardware in the Loop Simulationssoftware eines Festo-Transfersystems

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Thomas Drauschke

Hardware in the Loop Simulationssoftware eines Festo-Transfersystems

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Thomas Lehmann
Zweitgutachter: Prof. Dr. Martin Becke

Eingereicht am: 12. Juni 2018

Thomas Drauschke

Thema der Arbeit

Hardware in the Loop Simulationssoftware eines Festo-Transfersystems

Stichworte

Hardware-in-the-Loop, zeitdiskrete Simulation, BeagleBone Black, Kollisionserkennung zweidimensionaler Körper, Physik-Engine, Schnittpunktberechnung, entfernte Prozeduraufrufe

Kurzzusammenfassung

Diese Bachelorarbeit befasst sich mit der Entwicklung einer Hardware in the Loop Simulationssoftware eines Festo-Transfersystems. Für diesen Zweck wurde das zu simulierende Festo-Transfersystem bezüglich seines Aufbaus und seiner Funktionen analysiert. Mit Box2D als Physik-Engine können Kollisionsmodelle zwischen Körpern realisiert werden. Hierfür werden verschiedene Hüllkörper verwendet, die in einem zweiphasigen Ansatz für die Kollisionsberechnung genutzt werden. Über Netzwerkdienste kann mittels entfernter Prozeduraufrufe auf die Simulation eingewirkt werden. Ein Überblick der Architektur ermöglicht ein besseres Verständnis der entwickelten Software.

Thomas Drauschke

Title of the paper

Hardware in the loop simulation software of a Festo transfer system

Keywords

Hardware-in-the-Loop, discrete-time simulation, Beaglebone Black, collision detection of two-dimensional bodies, remote procedure calls

Abstract

This bachelor thesis deals with the development of a hardware in the loop simulation software of a Festo transfer system. For this purpose, the Festo transfer system was analyzed in terms of its design and its functionality. With Box2D as its physics engine, collision models between bodies can be realized. The collision calculation uses a two-phase approach in which different hull bodies are used. The simulation can be influenced via network services by remote procedure calls. An overview of the architecture allows a better understanding of the developed software.

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Motivation 1
1.2. Problemstellung 2
1.3. Aufbau der Thesis 2

2. Grundlagen **4**
2.1. Festo-Transfersystem 4
2.1.1. Transfergut - Werkstücke 5
2.1.2. Verwendete Sensorik 7
2.1.3. Verwendete Aktorik 7
2.1.4. Interaktionen mit dem Transfersystem - Bedienfeld 8
2.1.5. Evaluierung von Kenngrößen des Transfersystems 8
2.2. Verwendeter Einplatinencomputer 12
2.2.1. Anforderungen 12
2.2.2. Spezifikationen 13
2.2.3. Nutzen für die Simulation 13
2.3. Computersimulation 13
2.3.1. Simulationsarten 14
2.3.2. Verwendete Simulationsform 16
2.4. Hardware-in-the-Loop 16

3. Anforderungsanalyse **18**
3.1. Funktionale Anforderungen vom Realsystem 18
3.2. Nichtfunktionale Anforderungen 19
3.2.1. Simulationskern 19
3.2.2. Kommunikationskern 24

4. Physik-Engine **30**
4.1. Funktionsweise der verwendeten Engine 32
4.2. Nutzen für die Simulation 33

5. Kollisionsbehandlung **35**
5.1. Hüllkörper 35
5.1.1. Schnitt zweier Axis-Aligned-Bounding-Boxen 38
5.1.2. Schnitttest zwischen Kreisen 40
5.1.3. Schnitttest zwischen Halbgerade und Kreis 42
5.1.4. Schnitttest zwischen Kreis und AABB 46

5.2. Kollisionsberechnung 47
5.2.1. Strategien für die Erkennung 48
5.2.2. A posteriori Verfahren 50
5.2.3. A priori Verfahren 51

6. Systemarchitektur 53
6.1. Simulationskern 53
6.1.1. TimerUnit und TimingViolationHandler 54
6.1.2. Simulationscontroller 55
6.1.3. Repräsentation der Register - DataHandler 56
6.1.4. DataHandler-Adapter 56
6.1.5. ObjectFactory 56
6.1.6. Komponenten 57
6.2. Kommunikationskern 60
6.2.1. Ansteuerung der Simulation über das Netzwerk 61
6.2.2. Nachrichtendienste 65
6.2.3. HiL-Schnittstelle 65

7. Implementation 66
7.1. Simulationszyklus 66
7.2. Zeitgeber 67
7.3. Probleme 69
7.3.1. RPC-Bibliotheken 69
7.3.2. Ansteuerung der GPIO-Pins 69
7.3.3. Programmoptionen 70

8. Validierung der Simulation 72

9. Fazit 74

A. Anhang 76
A.1. JSON-Nachricht aller Steuernachrichten - RPC Zugriff 76
A.2. JSON-Nachricht des Gesamtzustandes 77
A.3. JSON-Nachricht aller heruntergefallenenen Werkstücke 78

Tabellenverzeichnis

2.1. Werkstücktypen 6
2.2. Positionierung der Lichtschranken 10

3.1. Port A 20
3.2. Port B 20
3.3. Port C 20

Abbildungsverzeichnis

2.1. Festo-MPS® Transfersystem . . . 5
2.2. Werkstücke - Querschnitt und Draufsicht . . . 6
2.3. Maße des Transfersystems . . . 9
2.4. Positionierung von Sensorik und Aktorik . . . 9
2.5. Seitenansicht der Höhenmessung . . . 11
2.6. Versuchsaufbau der Geschwindigkeitsmessung . . . 11
2.7. BeagleBone Black . . . 12
2.8. Versuchsaufbau - HiL . . . 17

3.1. Initiale Registerbelegung der HiL-Simulation . . . 21
3.2. Proof of Concept - Förderband . . . 22
3.3. Proof of Concept - Höhenmessung . . . 23
3.4. Mapping der GPIO-Pins . . . 26
3.5. Exklusive Pins für den GPIO-Modus . . . 27

4.1. Box2D Testbett . . . 32

5.1. Verschiedene Hüllkörper . . . 36
5.2. Kollisionen verschiedener Hüllkörper . . . 37
5.3. Erweiterung verschiedener Hüllkörper mit mehreren gleichartigen Hüllfunktionen für eine höhere Kollisionspräzision . . . 38
5.4. Schnitt zweier AABBs . . . 39
5.5. Schnitt zweier Kreise . . . 41
5.6. Schnitt Halbgerade-Kreis . . . 44
5.7. Kollisionen zwischen Kreis und AABB . . . 46
5.8. Tunneling Effekt . . . 48
5.9. A posteriori Verfahren . . . 50
5.10. A priori Verfahren . . . 52

6.1. Simulationskern - Übersicht . . . 54
6.2. Kommunikationskern - Übersicht . . . 60
6.3. Aufbau Schnittstellenbeschreibungssprache . . . 64

7.1. Aufrufreihenfolge im Simulationszyklus . . . 67
7.2. Programmcode TimerUnit . . . 68
7.3. Boost ProgramOptions . . . 71

8.1. Testboard für die Überprüfung von Sensorik und Aktorik 72
8.2. Programm für die Überprüfung der HiL-Simulation 73

A.1. RPC Nachrichten 76
A.2. JSON - Gesamtzustand 77
A.3. JSON-Nachricht aller heruntergefallenenen Werkstücke 78

1. Einleitung

1.1. Motivation

Hardware-in-the-Loop Simulationen, auch HiL-Simulationen genannt, ermöglichen es das Verhalten von realen Systemen nachzustellen. Hierfür werden Verhaltensmodelle von dem zu simulierendem Testsystem in Echtzeit möglichst exakt nachempfunden. Vor allem in der Automobil-, Raum-, Automatisierungs- und Luftfahrtindustrie werden HiL-Simulationen verwendet. Systeme die zu gefährlich, zu teuer oder zu komplex zum Testen sind, lassen sich mithilfe von HiL-Simulation dennoch sicher testen [1]. Selbst wenn kein reales System existiert, kann vor Beginn einer Produktionsreihe ein simulativer Prototyp getestet werden, ohne dass derselbe real hergestellt werden muss.

In dieser Bachelorarbeit wird die Realisierung eines Prototypen einer HiL-Simulation vorgestellt, die ein Transfersystem mit der zugehörigen Steuereinheit nachbildet. An der *Hochschule für angewandte Wissenschaften Hamburg*, kurz *HAW Hamburg*, wird im vierten Fachsemester das Modul *Software Engineering 2* angeboten, indem die Studenten die erlernten Inhalte aus den Vorlesungen vorheriger Semester in einem fächerübergreifenden Projekt in die Praxis umsetzen. Die Aufgabe besteht darin, in einer Gruppe von vier Personen eine Ansteuerungssofterware für die Steuereinheit eines Transfersystems zu designen und zu implementieren.

Aus Modernisierungsgründen werden die zuvor genutzten, fest verbauten, Steuereinheiten mit Monitor, nun durch einen Einplatinencomputer ersetzt. Hierdurch kann dem Problem entgegengewirkt werden, dass die angebotenen Arbeitsplätze nicht ausreichen, sodass jede studentische Arbeitsgruppe gleichzeitig ihre Software weiterentwickeln kann. Durch die geringe Größe des Einplatinencomputers können die Transfersysteme neu positioniert werden und es können zudem weitere Arbeitsplätze geschaffen werden.

Eine HiL-Simulation bietet den Vorteil, dass die Studenten zu Hause ihre Arbeit fortsetzen können, ohne abhängig vom Realsystem zu sein. Um die Qualität der Steuerungssoftware zu

sichern, kann diese mittels automatisierter Tests gegen die HiL-Simulation verifiziert werden. Hierfür können Metriken aufgestellt werden, die vorzeitig Fehlschritte oder Erfolge darlegen. Mit einem automatisiertem Deployment der Software auf ein Testsystem, welches mit der Hil-Simulation gekoppelt ist, kann den Studenten ein wertvolles Hilfsmittel bereitgestellt werden, um Software Engineering 2 erfolgreich zu durchschreiten.

1.2. Problemstellung

Die HAW Hamburg verfügt über keine lauffähige Simulation des Festo-Transfersystems. Durch den Umbau der Steuereinheit auf einen Einplatinencomputer, besteht die Möglichkeit eine neue Simulation zu entwickeln. Die neue Simulation soll im Gegensatz zu der bereits existierenden, nicht mehr lauffähigen Simulation flexibler in der Verwendung sein. Beispielsweise konnten bei der vorherigen Simulation nur Werkstücke an bestimmten Positionen hinzugefügt werden. Es bestand auch nicht die Möglichkeit Werkstücke zu löschen oder deren Orientierung zu ändern. Weiterhin waren die Kollisionen von Werkstücken ungenau, wodurch unbestimmte Verhalten der Werkstücke auftraten. Zusätzlich wies die Simulation eine niedrige Aktualisierungsfrequenz auf, die sich durch eine hängende grafische Repräsentation bemerkbar machte. Die neue Simulation soll diese Probleme adressieren und besser als die alte Simulation werden.

Für den Softwareentwicklungsprozess der HiL-Simulation muss herausgefunden werden, wie das reale Transfersystem für den Einplatinencomputer abstrahiert werden kann, ohne dass Abweichungen in der Funktionsweise entstehen. Der Nutzer soll keine signifikanten Verhaltensunterschiede zwischen echtem System und simuliertem System bemerken. Um dies zu erreichen, muss die Simulation das Transfersystem mit Steuereinheit und den Benutzer des Transfersystems simulieren. Es müssen die Bewegungsabläufe und Kollisionen von Werkstücken, die Sensorik und die Aktorik nachgebildet werden. Weiterhin muss die Kommunikation an die Steuereinheit abstrahiert werden und es müssen Lösungen für die Einsicht auf den globalen Gesamtzustand des HiL-Systems gefunden werden.

1.3. Aufbau der Thesis

Um zu verstehen was die HiL-Simulation simuliert, wird in dem Grundlagenkapitel das Festo-Transfersystem vorgestellt. Der darauffolgende Abschnitt gibt Aufschluss über die Zielplattform auf der die HiL-Simulation in Betrieb genommen wird. Weiterhin wird erklärt wofür Hardware-in-the-Loop genutzt wird. Abschließend werden die in Betracht zu ziehenden Si-

mulationsarten vorgestellt. Für die Softwareentwicklung ist die detaillierte Analyse eines Systems unverzichtbar. Im Kapitel Anforderungsanalyse wird das Transfersystem inspiziert. Es werden die Anforderungen an die Simulation, Interprozesskommunikation und Abstraktion der Realität erhoben. Weil Bewegungsabläufe am genauesten durch physikalische Zusammenhänge beschrieben werden können, wird evaluiert, ob etwaige in der HiL-Simulation auf Basis einer Physik-Engine realisiert werden sollen. Für das dynamische Verhalten auf dem Förderband sind Kollisionen zwischen Werkstücken verantwortlich. Im Kapitel Kollisionsbehandlung wird aufgezeigt, welche Probleme es bei Kollisionsbehandlungen geben kann und wie gängige Physik-Engines Kollisionsberechnungen optimieren. Weiterhin werden die verwendeten Schnittpunktverfahren erläutert und grafisch repräsentiert. Für die Funktionsweise der HiL-Simulation sind Schnittpunktberechnungen unverzichtbar. Nach der Analyse der Anforderungen und der Betrachtung von Möglichkeiten der Realisierung, wird die Systemarchitektur näher betrachtet. Das Hauptaugenmerk liegt auf dem Aufbau des Simulationskerns und des Kommunikationskerns. Im Kapitel Implementation wird anhand von Pseudocode der Simulationszyklus erläutert. Näher betrachtet wird die Aufrufreihenfolge der nötigen Komponenten, die den Gesamtzustand der HiL-Simulation beeinflussen. Um die korrekte Funktionsweise der HiL-Simulation zu garantieren, werden im Kapitel Validierung der Simulation die verwendeten Testverfahren vorgestellt mit derer die Funktionsweise überprüft wurde. Abschließend folgen Fazit und ein Ausblick auf wünschenswerte Erweiterungen der HiL-Simulation.

2. Grundlagen

Für ein besseres Verständnis der Kernelemente auf dem die Arbeit basiert, wird in folgendem Kapitel das nötige Basiswissen für die Grundlagen der HiL-Simulation geschaffen. Hierfür wird ein detaillierter Einblick auf das zu simulierende Festo-Transfersystem geboten. Anschließend werden die Eigenschaften des zum Einsatz gebrachten Einplatinencomputers näher betrachtet. Weil die Hauptfunktionalität der Simulation über eine Hardware in the Loop Schnittstelle realisiert wird, muss erörtert werden wofür HiL-Systeme verwendet werden. Um die Entscheidung für den verwendeten Simulationstypen nachzuvollziehen, werden die zur Verfügung stehenden Simulationsarten vorgestellt und miteinander verglichen.

2.1. Festo-Transfersystem

Das in Abbildung 2.1 vorgestellte Transfersystem ist ein von der Firma FESTO hergestelltes Lernsystem der mechatronischen Systemreihe MPS®. Zweck dieser ist es Ausbildungen in verschiedenen Sparten innovativer und spannender zu gestalten [2]. Das Transfersystem soll Werkstücke auf eine Rutsche aussortieren oder bis an das Ende des Förderbandes befördern. Die Werkstücke werden auf dem Transportband von verschiedenen Sensoren erkannt und abhängig von unterschiedlichen Kriterien, mit den am System befindlichen Aktoren, auf eine Rutsche aussortiert. Um noch komplexere Sortiermechanismen zu realisieren, können mehrere Transfersysteme aneinandergereiht werden. Damit Benutzer des Systems mit Selbigem interagieren können, verfügt das Transfersystem über Leuchtdioden für Signalisierungen und Taster für Interaktionen. Die HiL-Simulation des Förderbandes basiert auf einem Referenzsystem eines Festo-Transfersystems aus der HAW Hamburg. Die verfügbaren Festo-Transfersysteme an der HAW Hamburg weichen untereinander in der Positionierung der Sensorik und Aktorik um wenige Millimeter ab. Aus diesem Grund beziehen sich die folgenden Aussagen auf das Festo-Transfersystem mit der Nummer *5*.

Abbildung 2.1.: Festo-MPS® Transfersystem

2.1.1. Transfergut - Werkstücke

Das Transfergut, folgend bezeichnet als Werkstück, bildet die Grundlage für die Sortierung auf dem Förderband. Ein Werkstück ist ein Zylinder aus Kunststoff. Dieser besitzt verschiedene Merkmale. Je nach Ausprägung der Merkmale des Werkstückes, kann ein Werkstückstyp definiert werden. Ein Werkstückstyp bezieht sich im Folgenden immer auf eine bestimmte Kombination aus Merkmalsausprägungen.

Folgende Merkmale können identifiziert werden :

- Radius - Der Radius beträgt 20 mm. Jedes Werkstück weist einen Durchmesser von 40 mm auf.
- Höhe - Die Höhe beträgt 21 mm oder 25 mm.
- Farbe - Für Werkstücke werden die Farben rot, schwarz, grau und weiss verwendet.
- Bohrung - Eine Bohrung ist eine mittig platzierte, kreisförmig gefräste Vertiefung in einem Werkstück.

- Metalleinsatz - Ein Metalleinsatz ist ein metallener Ring mit einem Radius von 11 mm und einer Breite von 3.5 mm. Voraussetzung ist eine Bohrung, damit der Metalleinsatz eingepresst werden kann.
- Kodierung - Eine Kodierung ist eine Reihe von radialen Fräsungen im Werkstück. Jede Fräsung besitzt einen Abstand von 3 mm zur nächsten Fräsung. Die Fräsungen können 3 mm oder 5 mm tief sein.
- Gewicht - Das Gewicht kann durch Bohrungen, Fräsungen, Dimensionen und Verwendung eines Metalleinsatzes variieren.

Auflistung der Werkstücktypen

Im Folgendem werden die zur Verfügung stehenden Werkstückstypen mit ihren Merkmalsausprägungen grafisch und tabellarisch vorgestellt.

Typ	Höhe	Farbe	Bohrung	Metall	Kodierung	Gewicht
Flach	21 mm	Schwarz/weiß	Nein	Nein	Nein	27 g
Normal	25 mm	Schwarz/rot	Ja	Nein	Nein	37 g
Metall	25 mm	Weiß	Ja	Ja	Nein	42 g
Kodiert 1	25 mm	Grau	Nein	Nein	Drei Fräsungen	35 g
Kodiert 2	25 mm	Grau	Nein	Nein	Drei Fräsungen	35 g

Tabelle 2.1.: Merkmalsausprägungen vorhandener Werkstücktypen

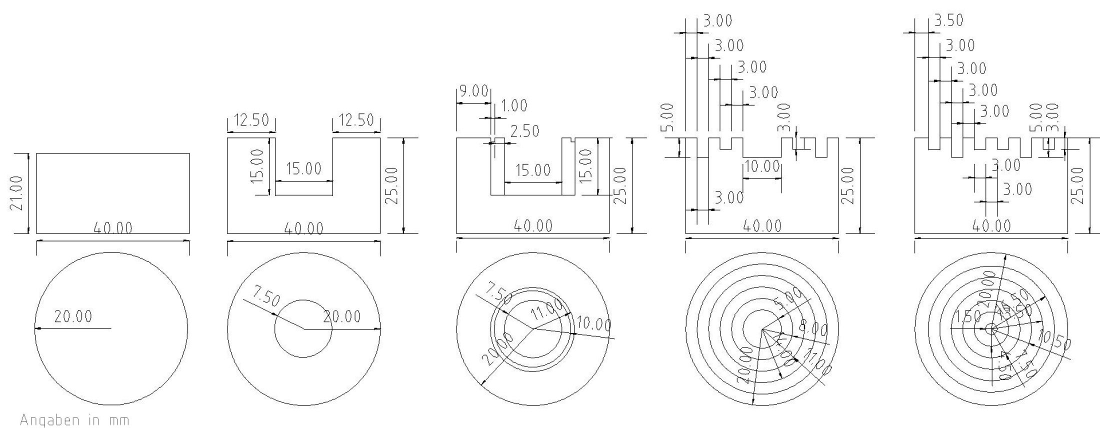

Abbildung 2.2.: Werkstücke - Querschnitt und Draufsicht

2.1.2. Verwendete Sensorik

Die in Abschnitt 2.1.1 genannten Werkstücktypen müssen lokalisiert und anhand ihrer Merkmalsausprägungen erfasst werden. Hierfür verfügt das Transfersystem über unterschiedliche Sensoren. Sensoren sind Funktionseinheiten, die ein elektrisches Signal anhand einer physikalischen oder geometrischen Größe erzeugen. So können beispielsweise elektrische Signale für Abstände oder Feldstärken erzeugt werden.

Für die Positionserfassung von Werkstücken werden Einweglichtschranken verwendet. Einweglichtschranken verfügen über einen Sender und einen Empfänger. Das vom Sender kommende Sendelicht wird auf den Empfänger abgestrahlt. Bei Unterbrechung des Sendelichts erfolgt eine Änderung des Ausgangszustands am Empfänger [3]. Das Transfersystem verfügt über vier Einweglichtschranken. Die Lichtschranke an der Rampe, siehe Abbildung 2.4, weist eine Besonderheit auf. Sie ist eine Reflexions-Lichtschranke. Hierbei befinden sich Sender und Empfänger in einem Gehäuse und der ausgehende Lichtstrahl wird an der gegenüberliegenden Seite an einem Reflektor auf den Empfänger zurückgeworfen.

Damit Entscheidungen zur Sortierung anhand der Höhe von Werkstücken getroffen werden können, verfügt das Transfersystem über einen Abstandssensor. Der am Transfersystem angebrachte Abstandssensor *SOEL-RTD-Q50-PP-S-7L* misst nach dem Triangulationsprinzip. Dabei wird der Abstand zwischen Objekt und Sensor anhand der Position des Lichtflecks auf dem Detektor bestimmt [4].

Für die berührungslose Erkennung von Metall in Werkstücken wird ein induktiver Näherungssensor verwendet, folgend Metallsensor genannt. Von der aktiven Oberfläche des Metallsensors wird ein oszillierendes elektromagnetisches Feld abgestrahlt. Zielobjekte aus Metall, die in das Feld gelangen, nehmen aufgrund des Wirbelstromeffekts eine geringe Menge an Energie von dem Oszillator auf. Wenn die Energieübertragung einen Schwellenwert erreicht, wird die Objekterfassung bestätigt und der Sensorausgang ändert seinen Zustand [5].

2.1.3. Verwendete Aktorik

Aktoren wandeln elektrische Signale in mechanische Bewegung oder andere physikalische Größen um. Um das Förderband zu bewegen, wird ein Vierquadranten-Antriebsregler für die Ansteuerung des Motors verwendet. Grundlage ist eine über Pulsweitenmodulation gesteuerte H-Brücke. Hierdurch kann sich das Förderband mit unterschiedlichen Geschwindigkeiten nach

links und rechts bewegen [6].

Für die Sortierung auf die Rampe wird ein Schalt-Drehmagnet verwendet. Dieser garantiert einen maximalen Ausschlagbereich von 65°. An die Aufnahmeachse ist ein metallener Stift befestigt. In eingeschaltetem Zustand versperrt der Stift das Förderband. Bewegt sich das Förderband nach rechts, werden durch die schräge Positionierung Werkstücke auf die Rampe aussortiert.

Damit das Transfersystem dem Nutzer optische Rückmeldung über dessen Zustand geben kann, besitzt es sieben Leuchtdioden. Wie in Abbildung 2.1 zu sehen, sind drei Leuchtdioden als Ampel am System installiert. Durch die farbigen Hüllen können die Farben grün, orange und rot dargestellt werden.

2.1.4. Interaktionen mit dem Transfersystem - Bedienfeld

An dem Transfersystem befindet sich ein Bedienfeld. Auf diesem Bedienfeld sind zwei Aktor-Sensor Felder installiert. Diese weisen eine Leuchtdiode und einen Taster auf. Durch die Leuchtdiode sind es Sensoren, durch den Taster jedoch Aktoren. Die Sensor-Aktor Felder haben die Bezeichnung *Start* und *Reset*. Das Bedienfeld weist zusätzlich noch einen Taster mit der Bezeichnung *Stop* auf. Zwei Leuchtdioden namens *Q1* und *Q2* ergänzen das Bedienfeld. Rechts neben den Sensor-Aktor-Feldern befindet sich der *Not-Aus*. Dies ist ein Schalter und somit ein reiner Sensor.

2.1.5. Evaluierung von Kenngrößen des Transfersystems

Damit das Referenzsystem korrekt abgebildet werden kann, muss es eingemessen werden. Wichtige Kenngrößen bilden die Positionierung der verfügbaren Aktoren und Sensoren. Für die korrekte Arbeitsweise muss zudem die Geschwindigkeit des Förderbandes erfasst werden. Folgender Abschnitt beinhaltet die für die Software nötigen Messergebnisse. Koordinaten sind in der Form $P(x/y)$ angegeben und definieren einen Punkt in einem zweidimensionalen kartesischen Koordinatensystem.

Positionierung der Aktoren und Sensoren

Für eine genaue Erfassung wurde das Transfersystem in Millimetern vermessen. Das Förderband beträgt eine Länge von 700 mm mit einer Tiefe von 70 mm. Der Abstand zu den äußeren Förderbandbegrenzungen beträgt 5 mm. Die effektiv nutzbare Gesamthöhe des Förderbandes

weist somit 80 mm auf. Die am Transfersystem angebrachte Rampe ist 180 mm lang und 70 mm breit. Sie weist eine Verengung von 30 mm an der zulaufenden Seite vom Förderband auf.

Angaben in mm

Abbildung 2.3.: Maße des Transfersystems

Das Transfersystem wird in fünf unterschiedliche Bereiche aufgeteilt. Jeder Bereich weist eine Lichtschranke zur Erfassung von Werkstücken auf. Folgende Bereiche mit korrespondierender Lichtschranke können identifiziert werden. Abbildung 2.4 bietet einen Überblick über die Positionierung.

Abbildung 2.4.: Positionierung von Sensorik und Aktorik

- Einlauf - Im Einlaufbereich werden die Werkstücke auf das Förderband gelegt.

- Höhenmessung - Ist die korrespondierende Lichtschranke des Bereiches unterbrochen, befindet sich ein Werkstück in der Höhenmessung.
- Sortierung - Ist die korrespondierende Lichtschranke des Bereiches unterbrochen, befindet sich ein Werkstück in der Sortierung. Entweder wird das Werkstück von hier aus auf die Rampe aussortiert oder es wird bis zum Ende des Förderbandes transportiert.
- Rampe - Unterbricht ein Werkstück die Lichtschranke auf der Rampe, ist dies ein Indiz dafür, dass die Kapazität der Rampe ausgeschöpft ist.
- Ende - Unterbricht ein Werkstück die Lichtschranke im Endbereich, müssen Maßnahmen der Steuerungssoftware für den weiteren Verlauf des Werkstückes getroffen werden. Das Werkstück befindet sich am Ende des Förderbandes.

Lichtschranke	**Startpunkt**	**Endpunkt**
Start	P(20 / 0)	P(20 / 80)
Height	P(285 / 35)	P(300 / 80)
Switch	P(410 / 35)	P(430 / 80)
Ramp	P(430 / 106)	P(380 / 106)
End	P(655 / 0)	P(655 / 80)

Tabelle 2.2.: Positionierung der Lichtschranken

Die Weiche befindet sich im Bereich der Sortierung. Sie grenzt im geschlossenen Zustand an den Anfang der Verengung der Rampe. Vor der Weiche ist über dem Förderband der Metallsensor in 50 mm Höhe positioniert.

Im Bereich der Höhenmessung befindet sich der Abstandssensor. Dieser liegt bei P(300 / 55) und ist 270 mm über dem Förderband befestigt. Der Messpunkt befindet sich bei P(275 / 55). Der Abstandssensor misst beförderte Werkstücke seitlich mit einem Winkel von ungefähr 5.3°.

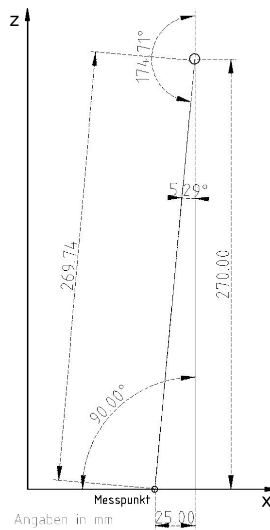

Abbildung 2.5.: Seitenansicht der Höhenmessung

Geschwindigkeit des Förderbandes

Das Förderband verfügt über zwei Geschwindigkeitsstufen. Schnell und langsam. Es weist für den Linkslauf und den Rechtslauf dieselben Geschwindigkeiten in beiden Geschwindigkeitsstufen auf. Um eine Durchschnittsgeschwindigkeit des Förderbandes für beide Geschwindigkeitsstufen zu ermitteln, wurde eine Testreihe mit 10 Testläufen aufgestellt. Hierbei wurde die Länge des Laufbandes um die Breite eines Werkstückes verkleinert. Das Förderband hat demnach eine Referenzlänge von 660 mm.

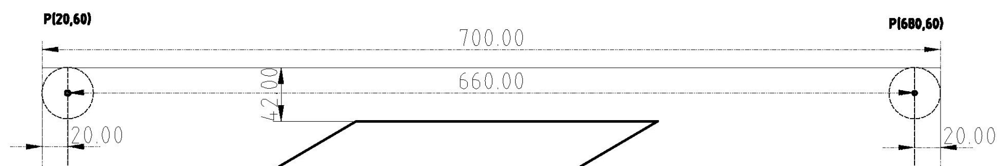

Abbildung 2.6.: Versuchsaufbau der Geschwindigkeitsmessung

Die Werkstücke wurden bei P(20 / 60) positioniert, damit sie gegen keine Verengung oder Förderbandbegrenzung stoßen. Die Referenzpunkte für Start und Stop der Werkstücke liegen somit bei P(20 / 60) und P(680 / 60). Der Mittelwert über die gemessenen Testresultate bildet die Geschwindigkeit V. Folgende Geschwindigkeiten wurden für den schnellen und langsamen Betrieb des Förderbandes gemessen.

$$V_{fast} = \frac{660mm}{6175ms} = 0.106883\frac{m}{s}$$

$$V_{slow} = \frac{660mm}{15500ms} = 0.042581\frac{m}{s}$$

2.2. Verwendeter Einplatinencomputer

In diesem Abschnitt werden die Anforderungen an den verwendeten *BeagleBone Black*, siehe Abbildung 2.7, näher betrachtet und evaluiert, ob er sich als geeignetes Embedded System für die HiL-Simulation erweist.

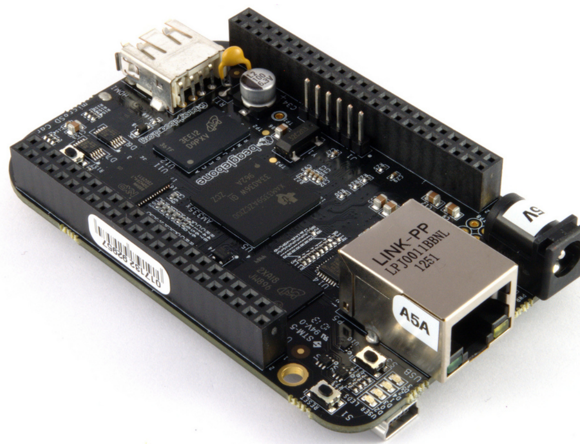

Abbildung 2.7.: BeagleBone Black

2.2.1. Anforderungen

Für die HiL-Simulation werden 18 digitale Eingänge und 8 digitale Ausgänge für die Abfrage der Sensoren und Interaktion der Aktoren benötigt. Weiterhin muss die Möglichkeit auf Kommunikation über die verfügbaren Pins bestehen. Zusätzlich wird ein Digital-Analog beziehungsweise Analog-Digital Wandler benötigt, um analoge Höhenwerte zu lesen/schreiben. Der Prozessor des Einplatinencomputers muss dieselbe oder höhere Leistung erbringen, als der Prozessor, der vorher verwendeten Steuereinheit. Es muss die Möglichkeit auf eine Netzwerkverbindung über Ethernet oder Wireless Lan bestehen.

2.2.2. Spezifikationen

Der BeagleBone Black - Revision C ist ein kostengünstiges Entwicklungsboard für 59.90 Euro. Es basiert auf einem 1 GHz Singlecore Sitara AM3358BZCZ100 ARM® Cortex®-A8 Prozessor von Texas Instruments. 512MB DDR3 RAM und 4GB 8-bit Embedded Multi Media Card Flashspeicher komplementieren das Entwicklungsboard. Es verfügt über Fast-Ethernet (100 Megabit/Sekunde) und einen Mikro SD Kartenslot. Durch zwei 46-polige Stiftleisten bietet das Board die Möglichkeit auf 65 *General Purpose Input/Outputs* (GPIOs), zwei *Inter-Integrated Circuits* (I^2C), ein *Serial Peripheral Interface* (SPI), 5 Konnektoren für *Universal Asynchronous Receiver Transmitter* (UART), 8 Pins für *Pulsweitenmodulation* (PWM), 4 Pins für Timer und 7 Analogeingänge. Das Entwicklungsboard kann über USB mit 3.3 Volt oder mit 5 Volt per Netzteil betrieben werden. Außerdem verfügt es über eine Mini-HDMI Schnittstelle für Multimediasupport [7].

2.2.3. Nutzen für die Simulation

Vergleichbare Einplatinencomputer verfügen meist über einen schneller getakteten Prozessor oder einen Mehrkernprozessor mit größerem Arbeitsspeicher [8]. Trotzdem sind sie aufgrund ihrer begrenzten Pinanzahl nicht für die HiL-Simulation geeignet. Durch den verfügbaren 4 GB großen eMMC Flashspeicher werden SD Karten obsolet. Das Betriebssystem kann direkt geflasht werden. Somit besteht kein Zwang der Verwendung einer SD-Karte und der BeagleBone Black wird zu einer eigenständigen Funktionseinheit. Mithilfe der Mini-HDMI Schnittstelle kann ein Bildschirm verwendet werden. Falls nötig können SSH Sessions somit vermieden werden. Als Betriebssystem stehen verschiedene Linux Distributionen wie Debian oder Angström zur Verfügung. Es besteht zusätzlich noch die Möglichkeit das Echtzeitbetriebssystem QNX zu verwenden.

2.3. Computersimulation

Computersimulationen stellen einen sicheren Weg zur Verfügung, um dynamische Prozesse zu testen. Hierfür werden digitale Abbildungen von Systemen oder Abläufen verwendet [9]. Mithilfe von Simulationen lassen sich viele Szenarien ausprobieren, die in der Realität nur schwer nachzustellen wären. Ein Raketenstart mit zu wenig Treibstoff wäre in der Realität ein ernshaft teures und zugleich riskantes Vorhaben. In Simulationen lässt sich die Realität flexibler, günstiger und sicherer gestalten. Bevor eine Simulation erstellt werden kann, muss ein hinreichend exaktes Systemmodell [10] definiert sein, um eine möglichst realistische di-

gitale Abbildung des zu simulierenden Systems zu erzeugen. Je exakter das Modell, desto detaillierter kann die Simulation gestaltet werden. Dadurch kann die Qualität von erhobenen Daten und Kenntnissen drastisch verbessert werden. Für die Industrie und Forschung sind Simulationen als Instrument anzusehen. Sie können für Durchführbarkeitsbelege, präventive und generelle Fehlervermeidung für Produkte oder nur für reine Testzwecke verwendet werden.

Computersimulationen lassen sich in statische und dynamische Simulationen unterteilen. In statischen Simulationen werden Änderungen durch die Zeit nicht berücksichtigt. Es ist somit eine reine Berechnung, die nicht weiter von der Zeit beeinflusst wird. Eine statische Simulation wäre beispielsweise eine *Monte-Carlo-Simulation* [11]. Dynamische Simulationen hingegen berechnen zeitabhängige Prozesse. Für die Abstraktion des Festo-Transfersystems ist Zeit eine wichtige Kenngröße. Aus diesem Grund muss die verwendete Simulation dynamisch sein. Wie die Zeit in einer dynamischen Simulation vergehen kann, wird in folgendem Abschnitt behandelt.

2.3.1. Simulationsarten

Wenn als Grundlage einer Simulation Zeitpunkte verwendet werden, ist dies eine diskrete Simulation [12]. Die Zustandsänderungen der Simulation werden zu bestimmten Zeitpunkten neu berechnet. Hierfür muss eine geeignete Form der Darstellung der Zeitpunkte gewählt werden. Für die HiL-Simulation bieten sich zwei Möglichkeiten an, die im Folgendem beschrieben werden.

Zeitorientierte Simulationen

Die *zeitorientierte Simulation* verwendet ein Zeitraster, ähnlich des Zeitrasters beim *Sampling* von analogen Signalen. Es werden äquidistante Zeitabschnitte Δt verwendet, anhand derer eine Aktualisierung des Gesamtzustandes der Simulation erfolgt. Die Vorteile zeitorientierter Simulation liegen in der einfachen Realisierung und der Übersichtlichkeit. Meist wird nur eine Schleife benötigt, die in den fest definierten zeitlichen Abständen die Aktionen der Simulation ausführt.

Algorithm 1 Zeitorientierte Simulation

```
while simulation should be running do
    waitUntilTimeSliceIsOver();
    readInputs();
    updateSimulation();
    updateOutputs();
end while
```

Mithilfe des zeitorientierten Ansatzes kann die physikalisch laufende Echtzeit nachgebildet werden. Mit jeder Iteration der Schleife vergeht ein Zeitabschnitt Δt in der Simulation, wie auch in der Echtzeit. Ein großer Nachteil ist die Möglichkeit, dass geänderte Eingänge der Simulation nicht erfasst werden können, falls die Änderungen kürzer als einen Zeitabschnitt anliegen. Um auf kurzzeitige Ereignisse angemessen zu reagieren, wird die folgende Simulationsart verwendet.

Ereignisorientierte Simulationen

Um eine hohe Genauigkeit von dynamischen Systemen realistisch abzubilden, wird die *ereignisorientierte Simulation* verwendet. Grundlage hierfür sind zustandsändernde Aktionen, *Events* genannt. Für jedes Event wird eine vom Ereignistyp abhängige Ereignisroutine ausgeführt [12], wobei das zeitliche Auftreten des Events berücksichtigt wird. Um die passenden Routinen zu finden, besitzen Events einen Typen, anhand dessen die Ereignisroutine ausgewählt wird. Ausgeführt werden die Ereignisroutinen in der Reihenfolge des Auftretens der Events. Ereignisroutinen sind Teil der Simulationssoftware und gliedern sich im Allgemeinen in drei Aufgabenbereiche. Es muss ein neuer Simulationszustand berechnet werden. Es müssen eventuelle Folgeereignisse generiert werden, die durch das Auftreten des vorherigen Events ermöglicht worden sind und es müssen Auswertungen durchgeführt werden [12]. Folgeereignisse werden in Listen, *Future-Event-Lists*, gespeichert, die sequentiell verarbeitet werden. Eine Mögliche Implementierung könnte wie folgt strukturiert sein.

Algorithm 2 Ereignisorientierte Simulation

```
Event currentEvent;
Time simulationTime;
while futureEventList.hasEvents() do
    currentEvent ← futureEventList.getAndRemoveNextEvent();
    simulationTime = currentEvent.getTimeOfArrival();
    updateSimulationWithEventRoutine(currentEvent.type);
end while
```

Ein Vorteil der ereignisorientierten Simulation ist die detaillierte und zeitlich exakte Erfassung von Events. Kurzzeitig anliegende Events werden nicht wie in der zeitorientierten Simulation eventuell übersehen. Hierfür muss jedoch bei der Implementation ein höherer Aufwand in Kauf genommen werden. Die Menge von möglichen Events muss definiert werden und die korrespondierenden Eventroutinen müssen anhand von Typspezifikationen zugänglich sein.

2.3.2. Verwendete Simulationsform

Für die HiL-Simulation wird eine zeitorientierte Simulation benutzt. Auch wenn ereignisorientierte Simulationen jedes auftretende Event erfassen, könnte es Phasen geben, in denen keine Events auftreten. Dies würde für die Simulation bedeuten, dass Phasen vorkommen können, in denen es keine Zustandsänderungen gibt. Von einem mathematischen Standpunkt betrachtet, sind Berechnungen von zeitabhängigen Funktionen mit einem konstanten Zeitschritt Δt zusätzlich einfacher zu realisieren. Ansonsten müsste für jeden Simulationschritt berechnet werden wie lange zwei Events voneinander entfernt waren. Hier könnte es zu ersten numerischen Ungenauigkeiten kommen. Um dem Problem der nicht berücksichtigten Events entgegenzuwirken, müssen Signale für die Simulation, siehe Abschnitt 3.2.2, mindestens einen Zeitabschnitt Δt anliegen. Die Grundarchitektur, beschrieben in Kapitel 6, benötigt zusätzlich für einige Funktionalitäten einen Zeitgeber. Dieser könnte zwar über einen periodischen Eventgenerator realisert werden, hätte jedoch einen größeren Overhead als die eigentliche Kontrollstruktur der zeitorientierten Simulationsform.

2.4. Hardware-in-the-Loop

Hardware-in-the-Loop bezeichnet ein Testverfahren von eingebetteten Systemen. Es findet verstärkt Anwendung in der Automobilindustrie, weil moderne Fahrzeuge und deren Motoren mit zahlreichen Subsystemen ausgestattet sind [1], die elektrisch physikalische Größen verändern müssen. Als Beispiel können Steuergeräte, wie das Motorsteuergerät im Auto genannt werden, welche anhand von Sensorsignalen Einfluss auf das Auto über verschiedene Mechanismen nehmen. Um eingebettete Controller effizient auf ihre Funktionalität zu testen, werden diese nicht in dem Realsystem, sondern in einer virtuellen Echtzeitumgebung getestet, die das zu regelnde physische System abbildet [1], auch HiL-Simulator genannt. Eingebettete Systeme können über Sensoren den Zustand eines Systems erfassen und über Aktoren auf dieses System einwirken. Um eingebettete Systeme mit HiL zu testen, wird die zu testende eingebettete Einheit mit dem HiL-Simulator auf elektronischer Signalebene verbunden. Die Eingänge des eingebetteten Systems werden mit Sensordaten aus der HiL-Simulation stimu-

liert. Die Ausgänge, die eigentlich Aktoren steuern würden, werden in den HiL-Simulator zur Auswertung zurückgeführt. Durch diesen Testaufbau lassen sich Tests an realen Systemen vehement verringern, wodurch das reale Zielsystem und die Umwelt nicht gefährdet werden.

Die HiL-Simulation bildet das physische Festo-Transfersystem nach. Die zu testende eingebettete Einheit bildet die Steuereinheit, welche sich auf seiner realen Hardware, einem BeagleBone Black, befindet. Folgende Abbildung zeigt einen Testaufbau beider Komponenten.

Abbildung 2.8.: Versuchsaufbau - HiL

Der Hil-Simulator befindet sich auf dem linken BeagleBone Black. Die zu testende Steuerungseinheit befindet sich auf dem rechten BeagleBone Black. Sie sind über die verfügbaren GPIOs miteinander verbunden und bilden eine Hardware-in-the-Loop Teststrecke.

3. Anforderungsanalyse

Damit die HiL-Simulation vom Verhalten des Referenzsystems nicht abweicht, müssen im Systementwicklungsprozess Anforderungen aufgestellt werden. Anhand dieser Anforderungen werden Verhalten und Funktionsweite der HiL-Simulation bestimmt. Durch die Auseinandersetzung mit der Wahl des Simulationstypen und der Zielplattform wurden bereits Anforderungen identifiziert. Im folgendem Kapitel werden die nötigen Anforderungen für den weiteren Systementwicklungsprozess definiert.

3.1. Funktionale Anforderungen vom Realsystem

Die funktionale Anforderungen ergeben sich durch die Nutzung des Transfersystems. So muss die HiL-Simulation in der Lage sein alle Aktoren anzusteuern. Dies bedeutet, dass das Förderband Werkstücke mit den in Abschnitt 2.1.5 definierten Geschwindigkeiten befördern muss. Es muss in der Lage sein zu stoppen und zu starten. Weiterhin muss sich die Weiche öffnen und schließen lassen können. Eine geschlossene Weiche versperrt das Förderband und lenkt beförderte Werkstücke auf die Rampe. Für die Interaktionen mit dem Nutzer müssen die angebrachten Leuchtdioden angesteuert werden können. Die Leuchtdioden der Ampel - rot, gelb und grün, *Q1* und *Q2*, sowie die Leuchtdioden der Aktor-Sensorfelder, definiert in 2.1.4, müssen eingeschaltet und ausgeschaltet werden können. Weiterhin müssen die verfügbaren Sensoren ihren Zustand ändern können. Um die vollständige Nutzerbedienung zu gewährleisten, müssen die verfügbaren Taster und Schalter gedrückt und losgelassen werden können. Dies setzt voraus, dass die Aktor-Sensorfelder *Start* und *Reset*, der Taster *Stop* und der *Not-Aus* gedrückt und losgelassen werden können müssen. Um die Lokalisierung der Werkstücke zu gewährleisten, müssen die verfügbaren Lichtschranken unterbrochen werden können. Es müssen die in Abbildung 2.4 eingetragenen Lichtschranken in der HiL-Simulation auf Unterbrechungen reagieren können. Lichtschranken werden nur von beförderten Werkstücken unterbrochen. Der Metallsensor muss das Metall aus Werkstücken des Typen *Metall* erkennen können. Der Höhensensor muss, wie in Abbildung 2.5 dargestellt, einlaufende Werkstücke im Bereich der Höhenmessung ausmessen. Weil der Benutzer des Systems Teil der HiL-Simulation ist, und Werkstücke auf dem Förderband positioniert werden müssen, muss die HiL-Simulation

in der Lage sein Werkstücke eines in 2.1 definierten Werkstücktypen auf das Förderband zu legen. Weiterhin müssen Werkstücke entfernt werden können. Ein Werkstück muss bei Überschreitung des Förderbandes hinunterfallen. Eine Überschreitung liegt vor, wenn sich der Schwerpunkt des Werkstückes über dem Anfang oder dem Ende des Förderbandes befindet.

3.2. Nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen richten sich an die Qualität, in der die oben definierten Anforderungen realisiert werden müssen. Für eine übersichtlichere Bearbeitung wird die HiL-Simulation in zwei Funktionsblöcke unterteilt. Den Simulationskern und den Kommunikationskern.

3.2.1. Simulationskern

Der Simulationskern beinhaltet alle Funktionalitäten, um das Festo-Transfersystem zu simulieren. Hierzu gehört die Datenrepräsentation und Wartung des Gesamtzustandes der Simulation. Um den Datenbestand abhängig von der Zeit zu modellieren, muss ein Zeitgeber verfügbar sein. Dieser steuert die Aktualisierung der zu simulierenden Welt. Aktorik und Sensorik müssen Einfluss auf den Datenbestand nehmen. Desweiteren müssen Werkstücke abstrahiert und definiert werden. Um eine dynamische Arbeitsweise des Festo-Transfersystems zu garantieren, müssen die in der realen Welt auftretenden physikalischen Abläufe modelliert werden.

Datenrepräsentation

Für die HiL-Simulation gilt, dass dieselbe Repräsentationsform der Register von der vorher verwendeten Steuereinheit übernommen werden muss. Zustandsänderungen von Aktoren und Sensoren müssen demnach im korrespondierendem Statusbit angepasst werden. Die Steuereinheit verfügt über drei 8-Bit breite Register, im folgenden Abschnitt *Port* genannt. Für folgende Tabellen gilt, dass die Spalte *Bezeichnung* Auskunft über die auf dem entsprechendem Bit liegende Funktion gibt. *Bit* gibt die Position der Funktion des Ports an. Die Spalte *Input Output, kurz IO*, definiert, ob die Funktion für Ein- oder Ausgaben definiert ist.

Über *Port A* werden Förderband, Weiche und die Leuchtdioden der Ampel, beschrieben in 2.1.3, gesteuert.

Funktion	Bit	IO	0	1
Motor Rechtslauf	0	IN	-	Das Förderband läuft nach rechts
Motor Linkslauf	1	IN	-	Das Förderband läuft nach links
Motor Langsam	2	IN	-	Benutzung der langsamen Geschwindigkeitsstufe
Motor Stopp	3	IN	-	Das Band stoppt. Bit 0 - 2 haben keinen Einfluss
Weiche Auf	4	IN	-	Die Weiche geht auf
Ampel Grün	5	IN	-	Die Leuchtdiode der grünen Ummantelung leuchtet
Ampel Gelb	6	IN	-	Die Leuchtdiode der gelben Ummantelung leuchtet
Ampel Rot	7	IN	-	Die Leuchtdiode der roten Ummantelung leuchtet

Tabelle 3.1.: Port A

Folgende Tabelle zeigt die Belegung des Ausgaberegisters *Port B*. Dieses Register repräsentiert die Zustände der Lichtschranken, folgend *LB* genannt, den Zustand des Metallsensors, der Weiche und des Abstandssensors.

Bezeichnung	Bit	IO	0	1
LB Einlauf	0	OUT	Unterbrochen	Nicht unterbrochen
LB Höhenmessung	1	OUT	Unterbrochen	Nicht unterbrochen
Abstandssensor	2	OUT	Höhe zu klein/zu groß	Toleranzbereich eingehalten
LB Weiche	3	OUT	Unterbrochen	Nicht unterbrochen
Metallsensor	4	OUT	Kein Metall erkannt	Metall erkannt
Weiche offen	5	OUT	Weiche geschlossen	Weiche offen
LB Rampe	6	OUT	Unterbrochen	Nicht unterbrochen
LB Auslauf	7	OUT	Unterbrochen	Nicht unterbrochen

Tabelle 3.2.: Port B - LB=(Lightbarrier/Lichtschranke)

Folgende Tabelle stellt die Belegung des Ein und Ausgaberegisters *Port C* dar. Dieses repräsentiert die Zustände der Leuchtdioden und Taster im Bedienfeld des Festo-Transfersystems.

Bezeichnung	Bit	IO	0	1
LED Starttaste	0	OUT	Start-LED dunkel	Start-LED leuchtet
LED Resettaste	1	OUT	Reset-LED dunkel	Reset-LED leuchtet
LED Q1	2	OUT	Q1-LED dunkel	Q1-LED leuchtet
LED Q2	3	OUT	Q2-LED dunkel	Q2-LED leuchtet
Taste Start	4	IN	Starttaste nicht gedrückt	Starttaste gedrückt
Taste Stop	5	IN	Stoptaste gedrückt	Stoptaste nicht gedrückt
Taste Reset	6	IN	Resettaste nicht gedrückt	Resettaste gedrückt
Taste Not-Aus	7	IN	E-Stoptaste gedrückt	E-Stoptaste nicht gedrückt

Tabelle 3.3.: Port C

Die initiale Registerbelegung der Steuereinheit lässt sich anhand der obenstehenden Tabellen ablesen. Folgende Abbildung stellt die initiale Registerbelegung dar.

	MSB	Binär						LSB	Dezimal	Hexadezimal
Position	7	6	5	4	3	2	1	0		
Port A	0	0	0	0	0	0	0	0	0	0
Port B	1	1	1	0	1	0	1	1	235	EB
Port C	1	0	1	0	0	0	0	0	160	A0

Abbildung 3.1.: Initiale Registerbelegung der HiL-Simulation

Zeitliche Komponente in der Simulation

Durch die Wahl einer diskreten zeitorientierten Simulation muss eine geeignete zeitliche Auflösung gefunden werden. Anforderungen sind schnelle Rechenzeiten für eine hohe Aktualisierungsfrequenz. Die zeitliche Auflösung definiert eine *feste Deadline*. Werden die Berechnungen des Gesamtzustandes nicht innerhalb der zeitlichen Auflösung erfolgreich abgeschlossen, so verzögert dies die Simulation. Daraus resultiert eine zeitliche Verschiebung der Simulation. Ergebnisse können dadurch unbrauchbar werden. Zugleich muss die Aktualisierungsrate hoch genug für angebundene Programme, wie grafische Nutzeroberflächen, sein. Zudem besteht ein direkter Zusammenhang zwischen der Aktualisierungsrate der Simulation und den genutzten *GPIO*-Pins. Diese müssen nach Zustandsänderungen aktualisiert werden. Sie sind somit Teil des Simulationszyklus. Zu lange Aktualisierungsintervalle könnten als falscher Datenbestand an den GPIO-Pins vorliegen. Falls die Simulation die *feste Deadline* überschreiten sollte, muss für diese Eventualität eine Maßnahme zur Bekanntmachung an angebundene Programme gefunden werden. Ein Überwachungsmechanismus muss alle angebundenen Programme über die Zeitverschiebung in Kenntnis setzen. Zusätzlich sollen die Onboard-LEDs des *BeagleBoneBlack* mit 10 Hertz blinken.

Nach Rücksprache mit Prof. Dr. rer. nat. Thomas Lehmann wurden 100 Millisekunden als Aktualisierungsintervall festgelegt. Diese Zeitspanne ist ausreichend kurz für eine Aktualisierung der Simulationsdaten, bietet jedoch noch Puffer für anstehende Berechnungen von eventuell nötigen Services für die Kommunikation.

Abstraktion der Welt

Die Sichtweise auf das Festo-Transfersystem ist dreidimensional. Um eine möglichst realistische Simulation zu erzeugen, liegt es nahe die simulierte Welt dreidimensional zu gestalten.

Für die HiL-Simulation kann eine zweidimensionale Abstraktion einer dreidimensionalen Weltanschauung genügen. Hierdurch können Berechnungen einfacher realisiert werden, wodurch sich die Komplexität und die Laufzeit des Programmes verringern lassen.

In Abbildung 3.2 wird eine zweidimensionale Abstraktion des Festo-Tranfersystems aus der Vogelperspektive dargestellt. Die Lokalisierung der Werkstücke und die Physik begrenzen sich auf die *X* und *Y* Komponenten der dreidimensionalen Welt. Abgebildet ist ein Versuchsaufbau in der Physik-Engine *Box2D*, aufgeführt in Kapitel 4. Der Versuchsaufbau besteht aus sechs Rechtecken, die als Förderbandbegrenzung verwendet werden. Die Verengung des Förderbandes ist eine Polygonform. Die Werkstücke sind Kreise mit einem 40 mm Durchmesser.

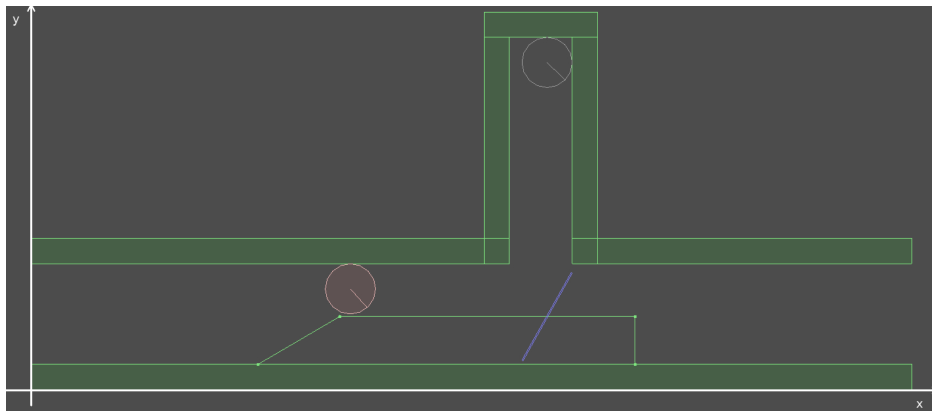

Abbildung 3.2.: Proof of Concept - Förderband

Die Höhenmessung und die Rampe sind die einzigen Komponenten des Transfersystems, die die Z-Achse benötigen. Aus diesem Grund benötigen sie eine seperate Darstellungsform. Auf dem Förderband werden Werkstücke immer in demselben Y-Achsenabschnittbereich durch die Höhenmessung befördert. Der fixe Y-Achsenabschnitt wird durch die Förderbandbegrenzung erzwungen. Einlaufende Werkstücke werden, über die Schräge der Förderbandbegrenzung, auf die richtige Y-Position verschoben. Werden eventuelle Abweichung der Y-Position vernachlässigt, lässt sich die Höhenmessung auf zwei Dimensionen reduzieren. Um realistische Höhenergebnisse zu produzieren, muss der Abstandssensor nur das Profil des Werkstückes messen. Folgende Abbildung zeigt eine mögliche Sichtweise der Höhenmessung auf dem Festo-Transfersystem. Zu erkennen sind die Seitenansichten aller Werkstückstypen. Der Abstandssensor ist für diesen Versuch frei positioniert.

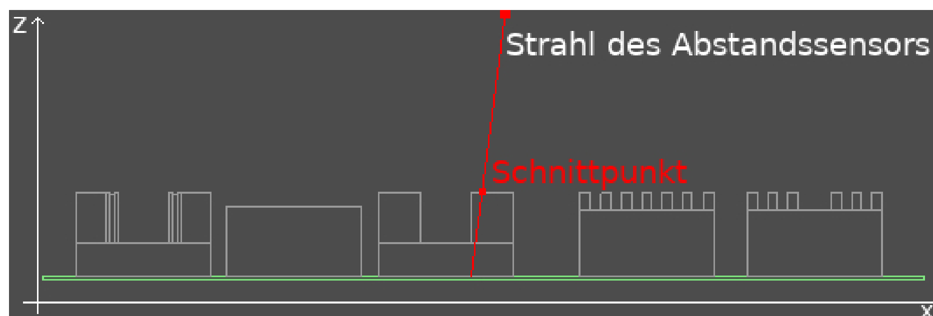

Abbildung 3.3.: Proof of Concept - Höhenmessung

Anhand des Prototypen der Darstellungsform der Höhenmessung kann gezeigt werden, dass die Höhe der Werkstücke sehr simpel berechnet werden kann. Mithilfe des Schnittpunktes, dem Ursprung des Strahls und dem Satz des Pythagoras kann die Distanz zwischen Abstandssensor und Körper berechnet werden. Abschließend kann die Aussage getroffen werden, dass für die HiL-Simulation eine Abbildung der dreidiemensionalen Welt in zwei zweidimensionale Darstellungsformen ausreichend ist. Die nun folgenden Anforderungen stützen sich auf diese Aussage.

Abstraktion des Transfersystems

Damit die zu befördernden Werkstücke bei Kollisionen nicht vom Förderband fallen, müssen die in Abschnitt 2.1 definierten Begrenzungen des Förderbandes realisiert werden. Werkstücke, die auf die Rampe aussortiert werden, müssen bis an das Ende der Rampe befördert werden. Die Rampe ist schräg an dem Festo-Transfersystem befestigt. Die Neigung der Rampe führt mit der Erdanziehungskraft dazu, dass Werkstücke die Rampe herunterrutschen. Es wurde sich für eine Abstraktion der Rampe entschieden. Dadurch genügt eine konstante Geschwindigkeit, die auf das Werkstück einwirkt. Gravitation und Reibungskoeffizienten des Materials aus denen die Förderbandbegrungzen und die Rampe bestehen, können vernachlässigt werden.

Abstraktion der Werkstücke

Werkstücke weisen verschiedene Merkmale auf. Ein Werkstück muss, je nach Typ, die in Abschnitt 2.1.1 aufgelisteten Merkmalsausprägungen aufweisen. Zusätzlich benötigen Werkstücke eine eindeutige Identifikationsnummer und eine Orientierung. Es müssen für die Vogelperspektive und die Seitenansicht der Werkstücke entsprechende Repräsentationsformen vorliegen.

Damit Werkstücke sich gegenseitig beeinflussen, müssen sie bei Überschneidungen ein Kollisionsverhalten aufweisen. Werkstücke können untereinander kollidieren und es können Kollisionen zwischen Werkstücken und den Begrenzungen des Transfersystems existieren. Um Kollisionen zu ermöglichen, müssen Werkstücke über das Förderband befördert werden können. Reibungskoeffizienten des Materials aus denen das Werkstück hergestellt wurde, können vernachlässigt werden.

Abstraktion der Aktorik und Sensorik

Für die Aktorik und Sensorik können Schaltverzögerung von elektronischen Schaltungen vernachlässigt werden. Der Motor des Transfersystems muss demnach keine S-Kurvenapproximation vorweisen. Es genügt eine direkte Umsetzung der korrespondierenden Aktion. Für den Metallsensor genügt es Metall zu erkennen. Es muss kein Schwellwert einer Energieübertragung erreicht werden. Positionsveränderungen der Weiche müssen kein Schwungverhalten aufweisen. Es genügt die Bewegung der Weiche auf zwei Positionen zu begrenzen.

Abstraktion der Physik

Durch die zweidimensionale Modellierung des Transfersystems können Kräfte, welche über die Z-Achse wirken, nicht berücksichtigt werden. In der Simulation genügt es, wenn die Werkstücke auf dem Förderband liegen bleiben. Die Rampe kann statt einer schiefen Ebene auf andere Weise realisiert werden. Es wird demnach keine Gravitation benötigt. Bei Kollisionen soll die gesamte kinetische Energie der zusammenstoßenden Körper erhalten bleiben. Energieverlust durch Reibung, Hitze, Geräusche oder andere Verlustformen können vernachlässigt werden.

3.2.2. Kommunikationskern

Der Kommunikationskern beinhaltet die benötigte Interprozesskommunikation über das eigene physikalisch verfügbare Netzwerk. Das vorhandene Netzwerk ist unter Kontrolle des Zuständigen an der HAW Hamburg. Mögliche Probleme wie Bottlenecks, Fairness oder Paketverlust fallen nicht in das Gewicht der Anforderungsanalyse. Der Kommunikationskern muss folgende Anforderungen erfüllen. Der Gesamtzustand der HiL-Simulation muss einsehbar sein. Er muss die Steuerung der HiL-Simulation über das Netzwerk ermöglichen. Zusätzlich muss die Steuerung über die GPIO-Pins des BeagleBone Black ermöglicht werden. Es existiert kein Anwendungsfall für eine gleichzeitige Ansteuerung der Aktorik. Beide Steuerungsvarianten sollen trotzdem parallel zur Verfügung stehen. Es muss möglich sein die Funktionalitäten des Klientenmodus, ausser die an die Aktorik angebundenen Funktionen, über das Netzwerk

anzusprechen. Hierfür soll die HiL-Simulation verschiedene Modi vorweisen. Der Klientenmodus akzeptiert Steuerbefehle über das Netzwerk, ignoriert jedoch Pegeländerungen an den GPIO-Pins. Im GPIO-Modus soll die Simulation Steuerbefehle eines verbundenen Klienten ignorieren. Bei dem Versuch die Simulation über das Netzwerk zu steuern, muss eine Informationsnachricht, über den momentanen GPIO-Modus, der Simulation zurück an den Klienten gesendet werden.

Steuerung über das Netzwerk

Die Steuerung der HiL-Simulation über das Netzwerk muss für noch zu entwickelnde Programme verfügbar sein. Ziel der Netzwerksteuerung ist folgenden Funktionsumfang zu gewährleisten.

- Das Förderband muss angesteuert werden können. Die Geschwindigkeit und die Richtung müssen manipuliert werden können. Es bedarf keiner Rückmeldung der HiL-Simulation.
- Die Leuchtdioden des Festo-Transfersystems müssen ein- und ausgeschaltet werden können. Es bedarf keiner Rückmeldung der HiL-Simulation.
- Die Taster des Festo-Transfersystems müssen gedrückt und losgelassen werden können. Es bedarf keiner Rückmeldung der HiL-Simulation.
- Die Zustände der Sensorik und Aktorik müssen abgefragt werden können. Als Antwort genügt eine Wertinterpretation als boolescher Wert des jeweiligen Zustands.
- Die Simulation muss zurückgesetzt werden können. Dies bedeutet, dass der initiale Zustand der Simulation hergestellt werden muss. Im initialen Zustand befindet sich kein Werkstück auf dem Förderband oder der Rampe. Der Datenbestand weist die initialen Registerwerte, siehe Abbildung 3.1, des Referenzsystems auf.
- Der globale Gesamtzustand der Simulation muss abgefragt werden können. Weitere Erläuterungen in 3.2.2
- Werkstücke müssen an einer variablen Position hinzugefügt werden können. Es werden die Position und der Typ des Werkstücks benötigt. Es ist zu überprüfen, ob sich die angeforderte Position auf dem Förderband befindet. Es ist nicht vorgesehen Werkstücke auf der Rampe zu platzieren. Es wird kein Werkstück hinzugefügt, falls an der gewünschten Position eine Kollision mit einem anderen Werkstück entstehen würde oder die übergebenen Koordinaten außerhalb des Förderbandes liegen. In diesem Fall soll der

Klient über den Fehlversuch informiert werden. Falls dies nicht der Fall sein sollte, wird der Klient über die Identifikationsnummer des neu hinzugefügten Werkstücks informiert.

- Werkstücke müssen anhand ihrer Identifikationsnummer gelöscht werden können. Es ist zu überprüfen, ob sich das Werkstück auf dem Förderband oder der Rampe befindet. Falls dies der Fall sein sollte, wird das Werkstück gelöscht und der Klient wird darüber informiert. Andernfalls wird das Werkstück nicht gelöscht und der Klient wird anhand einer Fehlermeldung über den Fehlversuch aufgeklärt.
- Die vertikale Ausrichtung der Werkstücke muss anhand ihrer Identifikationsnummer verändert werden können. Befindet sich kein Werkstück mit der Identifikationsnummer auf dem Förderband oder der Rampe, wird der Klient anhand einer Fehlermeldung darüber benachrichtigt. Konnte das Werkstück erfolgreich gedreht werden, wird der Klient mit einer Erfolgsnachricht darüber informiert.

Steuerung über GPIO

Die Steuerung über GPIO sieht vor, dass die Steuerungssoftware die Aktorik und Sensorik über die GPIO-Schnittstelle ansprechen kann. Folgende Abbildung zeigt das festgelegte Mapping zwischen Funktionen und verfügbaren GPIO-Pins des BeagleBone Black. Für Testzwecke sind Ausgänge für die Ampel mit *_TESTBOARD* eingetragen.

	P9				P8		
	1	2			1	2	
	3	4			3	4	
	5	6			5	6	
	7	8		BUTTON_START	7	8	BUTTON_STOP
	9	10		BUTTON_EMERGENCY_STOP	9	10	BUTTON_RESET
	11	12	LIGHTBARRIER_RAMP	LIGHTBARRIER_HEIGHT	11	12	LIGHTBARRIER_START
METAL_STATUS	13	14	LIGHTBARRIER_END	LED_RESET	13	14	LED_Q1
ENGINE_STOP	15	16	ENGINE_SLOW	LIGHTBARRIER_SWITCH	15	16	HEIGHT_OK
	17	18		LED_Q2	17	18	
	19	20		LED_START	19	20	
ENGINE_LEFT	21	22	ENGINE_RIGHT		21	22	
SWITCH_STATUS	23	24	GREEN_LIGHT		23	24	
	25	26	YELLOW_LIGHT		25	26	GREEN_LIGHT_TESTBOARD
RED_LIGHT_TESTBOARD	27	28			27	28	
	29	30	YELLOW_LIGHT_TESTBOARD		29	30	
	31	32			31	32	
	33	34			33	34	
	35	36			35	36	
	37	38			37	38	
	39	40			39	40	
SWITCH	41	42	RED_LIGHT		41	42	
	43	44			43	44	
	45	46			45	46	

Abbildung 3.4.: Mapping der GPIO-Pins

Die GPIO-Schnittstelle verwendet alle zur Verfügung stehenden Datenregister der HiL-Simulation. Für die parallele Nutzung der GPIO-Schnittstelle und dem Klientenmodus müssen folgende Registerpositionen exklusiv für den GPIO-Modus zur Verfügung stehen.

	MSB	Binär						LSB
Position	7	6	5	4	3	2	1	0
Port A	x	x	x	x	x	x	x	x
Port B	-	-	-	-	-	-	-	-
Port C	x	x	x	x	x	x	x	x

Abbildung 3.5.: Exklusive Pins für den GPIO-Modus

Durch das exklusive Nutzungsrecht des Ports *A* und *C* im GPIO-Modus, können Schreibzugriffe eines steuernden Klienten unterbunden werden. Es muss keine weitere Zugriffsmechanik zur Realisierung der korrekten Zugriffsreihenfolge realisiert werden. Der Klientemodus verfügt somit nur über lesende Zugriffsrechte, wenn sich die HiL-Simulation im GPIO-Modus befindet.

Veröffentlichung des Gesamtzustandes

Die periodische Veröffentlichung des Gesamtzustandes wird für überwachende Anwendungen wie grafische Benutzeroberflächen oder noch zu entwickelnde Analysetools für die Steuerungssoftware verwendet. Klienten sollen sich mit der HiL-Simulation verbinden können und periodische Zustandsnachrichten der Simulation erhalten. Um Interoperabilität zwischen verschiedenen Systemarchitekturen zu unterstützen, soll als Nachrichtenformat die *JavaScript Object Notation - kurz JSON* verwendet werden. *JSON* ist ein leichtgewichtiges, textbasiertes, sprachunabhängiges Datenaustauschformat [13]. Notwendig für die Zustandsnachricht sind folgende Felder.

- Eine Auflistung aller Werkstücke auf dem Förderband und der Rampe. Es müssen die Werkstückinformationen des Typen, die momentane Position in *X* und *Y*-Koordinaten, die Identifikationsnummer und die Orientierung vorhanden sein.
- Die Ports *A*, *B* und *C* als vorzeichenlose Dezimalzahl.
- Eine Aufschlüsselung der Ports in logische Blöcke. Folgende Zustände sollen in logischen Blöcken realisiert werden.
  - Aktorik
    - Förderbandsteuerung
    - Weiche
    - Schalter
    - Leuchtdioden der Ampel

  - Sensorik
    * Lichtschranken
    * Gemessene Höhe des Abstandssensors
    * Metallsensor
    * Leuchtdioden des Bedienfeldes
  - Sonstiges
    * Ein Zeitstempel
    * Ein Feld indem das Überschreiten der Deadline markiert wird.
    * Aktualisierungsrate der Simulation

Zusätzlich zum globalen Gesamtzustand der HiL-Simulation muss eine Liste über heruntergefallene Werkstücke verfügbar sein. Es sollen maximal 20 heruntergefallene Werkstücke in die Liste aufgenommen werden. Falls die Kapazität der Liste ausgeschöpft ist, sollen neue Einträge die ältesten Einträge verdrängen.

Transportprotokoll

Die Steuerung der Simulation über das Netzwerk beschreibt eine der grundlegenden Funktionalitäten. Aus diesem Grund muss der Nutzer die Gewissheit haben, dass geforderte Steuerbefehle ausgeführt werden und nicht verloren gehen. Hierfür muss die Reihenfolge der abgesetzten Steuerbefehle eingehalten werden. Nachrichten dürfen sich nicht überholen, andernfalls könnten sich Steuerbefehle gegenseitig beeinflussen. Der Netzwerkverkehr in dem Internetprotokoll basierenden, privaten Netzwerk beschränkt sich auf die versendeten Pakete der HiL-Simulation und deren Klienten. Es wird angenommen, dass eine kurze Latenzzeit, sowie eine niedrige Paketverlustrate und somit ein großer Datendurchsatz vorherrscht. Zusätzlich kann davon ausgegangen werden, dass es keine Denial-of-Service Attacken in Form von *Syn-Flooding* [14] gibt. Um Nachrichten eindeutig Nutzern zuzuordnen und den Klientenmodus mit nur einem Nutzer zur Zeit zu ermöglichen, soll zwischen Client und HiL-Simulation eine Ende-zu-Ende Verbindung aufrecht erhalten werden. Für die Ansteuerung der HiL-Simulation soll daher ein verbindungsorientiertes, vollständiges, reihenfolgegesichertes Transportprotokoll benutzt werden.

Für die Nachrichtendienste kann ein unzuverlässiges Transportprotokoll verwendet werden. Eine feste Reihenfolge von Nachrichten ist nicht zwingend notwendig. Zusätzlich kann

die Größe von Zustandsnachrichten variieren, je nachdem wie viele Werkstücke auf dem Förderband befördert werden. Aus diesem Grund sollte der Durchsatz möglichst hoch sein. Das Transportprotokoll sollte deshalb einen niedrigen Overhead an Protokolldaten aufweisen.

Für die Ansteuerung der HiL-Simulation bietet sich das *Transmission Control Protocol - TCP* [15] an. Es weist alle oben genannten Anforderungen auf. Zu bemerken ist, dass die Menge an Nutzdaten, auch Payload genannt, einer Steuernachricht ungefähr 233 Byte beansprucht, wodurch es zu keiner Segmentierung von TCP-Paketen kommt. Für die Nachrichtendienste kann für einen ersten Prototypen der Hil-Simulation auch TCP verwendet werden. Die größtmögliche Nachricht des Gesamtzustandes weist mit 23 Werkstücken einen Payload von 3583 Byte auf. TCP und das IP-Protokoll definieren einen Header von 20 Bytes. Eine maximale Übertragungseinheit, *maximum Transmission Unit - kurz MTU*, beschränkt sich bei Ethernet auf 1500 Bytes [16]. Für die Nutzdaten bleiben in einem TCP/IP-Paket somit 1460 Bytes übrig. Hierdurch würde die größte Statusnachricht in drei Segmente unterteilt werden. Abhilfe würde die Nutzung von Jumboframes schaffen. Mit Jumboframes lassen sich 9000 Byte große Pakete versenden. Falls TCP nicht gewünscht ist, kann für eine schnelle, unzuverlässige Übertragung das *User Datagram Protocol*, kurz *UDP* [17], verwendet werden.

4. Physik-Engine

Verschiedene große Hersteller von *Triple-A-Videospielen*, wie *Santa Monica Studios*, *EA* oder *DICE*, beeindrucken immer wieder mit ihren cinematischen Darstellungen. Seien es einstürzende Häuser, explodierende Autos, realistisch wirkende Fußabdrücke im Schnee oder Reflektionen in Pfützen. Um dies zu realiseren werden von den verschiedenen Herstellern Spiel-Engines verwendet oder programmiert. Alle Spiel-Engines haben eines gemeinsam - die physikalischen Gesetzmäßigkeiten, die in Physik-Engines abgebildet werden.

Eine der größten, für private Zwecke freie, verwendeten Spiele-Engine ist die *UnrealEngine 4*. Sie verwendet als Physik-Engine die *PhysX 3.3-Engine* [18] von *NVIDIA*. Problematisch ist die Portierung dieser Engine auf den BeagleBone Black. Vorzugsweise sollen von der Physik-Engine nur simple Körper verwendet werden. Es besteht kein Grund für eine zu komplexe Darstellungsform. Bei Verwendung bestünde ein enormer Overhead an nicht verwendeten Ressourcen der Physik-Engine. Somit ist es nicht sinnvoll eine derartig große Spiele- oder Physik-Engine zu verwenden.

Andere quelloffene Physik-Engines, wie zum Beispiel *Bullet3* [19], ermöglichen nur unter größten Umständen eine Portierung auf ARM-Plattformen. Sie sind zwar für die Realisierung der HiL-Simulation bestens geeignet, können aber durch Versionierungskonflikte von verwendeten Bibliotheken nicht vollständig für die Zielplattform kompiliert werden.

Ein weiteres Problem besteht in der Integration einer Physik-Engine. Physik-Engines weisen einen monolithischen Aufbau auf und lassen sich meist nur als Ganzes verwenden. Die Verwendung von einzelnen Funktionalitäten, wie zum Beispiel der Kollisionsbehandlung, ist nicht vorgesehen. Die Entscheidung eine Physik-Engine zu benutzen, bedeutet letztendlich die Software auf dieses Kernelement zu stützen. Aufgrund des monolithen Designs und der Inkompatibilität für Arm-Architekturen, musste eine eigene Physik-Engine implementiert werden.

Die erste Iteration der HiL-Simulation benutzte eine dreidimensionalen Weltanschauung. Durch die simple Darstellungsweise der Werkstücke als Zylinder können Überprüfungen auf Kollisionen und Kollisionsauflösungen leicht implementiert werden. Jedoch wurde die selbstentwickelte Physik-Engine bei einer höheren Werkstücksanzahl unzuverlässig. Grund hierfür sind nicht optimierte Kollisionsberechnungen und eine falsche Strategie für Kollisionsbehandlungen.

Im folgendem Abschnitt wird die verwendete Physik-Engine vorgestellt, mit der es möglich war, sowohl die HiL-Simulation zu implementieren, als auch diese auf dem BeagleBone Black lauffähig zu gestalten. Box2D [20] ist eine quelloffene, zweidimensionale Physik-Engine. Sie wurde von Erin Catto in C++ entwickelt und unterliegt der *zlib-Lizens* [21]. Box2D besitzt keine Abhängigkeiten zu anderen Bibliotheken. Es werden nicht einmal *Standard Template Library*-Container verwendet. Somit bestehen keine Schwierigkeiten bei der Portierung auf ein beliebiges Zielsystem. Zusätzlich kann, je nach gewünschter Genauigkeit, bestimmt werden, ob Box2D Gleitkommazahlen oder Festkommazahlen verwenden soll.

Die Physik-Engine verfügt über eine Vielzahl an Features. Die wichtigsten Funktionalitäten sind die Kollisionerkennung, Callbackstrategien bei Kollisionen, Mehrfachhüllen für Körper und eine effiziente Kollisionsauflösung. Physikalisch können starre Körper, sogenannte *rigid bodies*, und *Feder-Modelle* simuliert werden. Starre Körper können über verschiedene Gelenktypen miteinander verbunden werden. Physikalisch werden Gravitation, Reibung, Elastizität des Körpers, Kräfte und Impulse berücksichtigt [22].

Um eine schnelle Entwicklung zu gewährleisten, verfügt Box2D über ein grafisches Testbett. Hierfür muss die Basistestklasse um selbst entwickelte Tests erweitert werden. Es können Interaktionen mit der Maus und vordefinierte Verhaltensmuster getestet werden. Zusätzlich können Hüllfunktionen, Kräfte, Schwerpunkte, Normalvektoren der Kollisionen und die Ausrichtung der Objekte dargestellt werden.

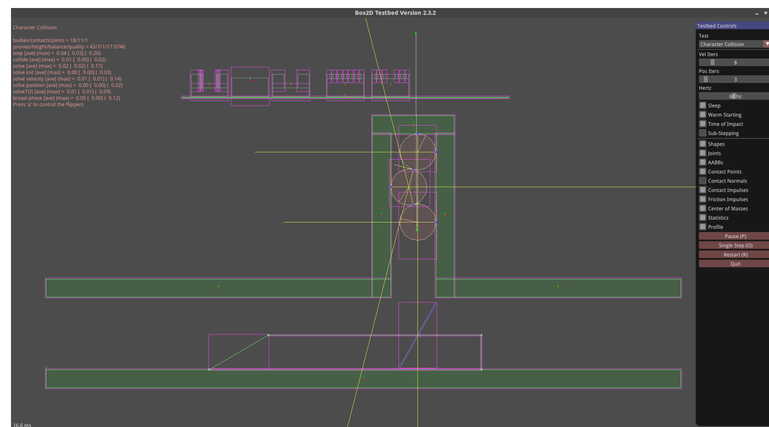

Abbildung 4.1.: Box2D Testbett

Aufgrund der hohen Portabilität, bei einem zugleich hohen Funktionsumfang, finden sich für die verbreitetsten Programmiersprachen und Plattformen Box2D-Portierungen. Verfügbar ist die Physik-Engine für *C++*, *C#*, *Java*, *JavaScript*, *Actionscript* und *Python*. Aufgrund der hohen Nutzerzufriedenheit wurde Box2D in *Unity-2D* integriert [23].

4.1. Funktionsweise der verwendeten Engine

In Box2D werden alle physikalischen Berechnungen über eine Welt gesteuert. In dieser Welt reagieren alle dort registrierten Körper miteinander. Die Welt definiert Gravitation und Simulationsschrittweite. Körper besitzen eine Form. Anhand dieser Form weisen Körper potentielle Berührungspunkte auf. In Abbildung 4.1 haben die Werkstücke auf dem Förderband eine Kreisform. Weitere Formen können Rechtecke oder selbst definierte Polygonhüllen sein. Um Bewegungsabläufe realistischer zu gestalten, benötigen Körper nicht nur eine Form, sondern auch materielle Eigenschaften. Box2D verwendet für diese Zwecke Zubehöreigenschaften. Diese Eigenschaften haben zwei Nutzen. Zum einen binden sie den Körper an eine Form und ermöglichen Kollisionen, zum anderen definieren sie deren Dichte, Reibungskoeffizient und Elastizität. In einer zweidimensionalen Welt haben Körper drei Freiheitsgrade. Die Rotation um sich selbst und zwei Translationen in X und Y-Richtung. Um die Freiheitsgrade zu beschränken, besitzt Box2D Maßnahmen zur Beschränkung. So können Körper auf Translationsebene oder Rotationsebene beschränkt werden.

Box2D verwendet für seine Berechnungen das *Meter-Kilogramm-Sekunde* System und Winkel werden in Radianten angegeben. Die Werkstücke aus der HiL-Simulation weisen in der Box2D-

Engine einen Radius von 20 Metern auf und wiegen 27 Kilogramm. Erin Catto weist darauf hin, dass die Engine zwischen 0.1 und 10 Metern gut funktioniert. Aus diesem Grund muss ein Umrechnungsfaktor zwischen Realität und Physik-Engine genutzt werden.

4.2. Nutzen für die Simulation

Neben den oben erwähnten Gründen für die Physik-Engine, verfügt Box2D Eigenschaften, die Berechnungen effizienter gestalten und die Laufzeit drastisch reduzieren können. Diese werden jedoch erst bei einer größeren Körperanzahl mit vielen Formen bemerkbar. Die HiL-Simulation profitiert trotzdem von folgenden Eigenschaften.

- Wirken auf einen Körper keine weiteren Kräfte, verfällt dieser in einen Schlafmodus. Schlafende Körper werden in anstehenden Kollisionsberechnungen nicht berücksichtigt. Kollidiert ein Körper mit einem schlafenden Körper, so wird dieser geweckt und in Folgeberechnung wieder berücksichtigt.
- Box2D verfolgt das Prinzip von *resource acquisition is initialization* - kurz *RAII*. Die Instanziierung von Box2D-Komponenten sollte auf dem Stack erfolgen. Dies schützt vor hinderlichen Memoryleaks.
- Weil in Physik-Engines viele kleine Objekte instanziiert werden, wären zusammenhängede Speicherbereiche ein Performancegewinn für Berechnungen. Durch häufige Speicherreservierung und Speicherbefreiung kann der *Heap* fragmentieren. Dadurch werden Berechnungen zeitintensiver. Um diesem Verhalten entgegenzuwirken, verwendet Box2D einen *small-object-allocator* - *SOA*. Zweck ist eine Verwaltung von zusammenhängendem, vorreserviertem Speicher auf dem Heap. Es werden dynamische Speicherpools mit verschiedener Größe auf dem Heap reserviert, die die Physik-Engine ausschliesslich für die Berechnungen verwendet. Werden neue Objekte wie Körper oder Formen instanziiert, fordert die Physik-Engine Speicher vom *SOA* an und nicht direkt vom Betriebssystem. Dieser gibt einen Speicherabschnitt zurück, der für die angeforderte Speichergröße am besten passt. Wenn Speicher befreit wird, wird die entsprechende Speicherstelle im Pool wieder für neue Anforderungen verfügbar.
- Durch eine kontinuierliche Kollisionsbehandlung kann dem *Tunnelingproblem* entgegengewirkt werden. Siehe Abschnitt 5.8 und 5.2.3 .
- Anhand von *Axis-Aligned-Bounding-Box Bäumen* kann die Selektierung von potentiellen Kollisionspartnern innerhalb der Broadphase beschleunigt werden. Siehe Abschnitt 5.2.1.

Durch die Optimierungen kann Rechenzeit in der HiL-Simulation gespart werden. Durch das Einhalten der Programmiertechnik RAII können Speicherlöcher vermieden werden, wodurch der verfügbare Arbeitsspeicher nicht unnötigt belegt wird. Dieser könnte sonst bei längeren Laufzeiten komplett belegt werden, wodurch das Betriebssystem gezwungen wäre Auslagerungsdateien zu verwenden. Dies würde die HiL-Simulation unnötig verlangsamen und eventuell eine Überschreitung des Aktualisierungsintervalls hervorrufen.

5. Kollisionsbehandlung

Die Kollisionsbehandlung ist das Kernelement für die Berechnung von realistischem Verhalten nach einem Zusammenstoß von Objekten. Sie unterteilt sich in zwei Probleme. Wann kollidieren Objekte miteinander und wie kann die Kollision aufgelöst werden. Im folgendem Kapitel werden Strategien zur Erkennung und Behebung von Kollisionen vorgestellt. Hierzu werden verschiedene Ideen vorgestellt und erklärt welche Ansätze der Kollisionsbehandlung Box2D verwendet.

5.1. Hüllkörper

Eine Kollision liegt vor, wenn sich zwei oder mehrere Objekte schneiden. Um herauszufinden, ob sich Objekte schneiden, muss es Regeln für die Berechnung dieser Überschneidungen geben. Eine Möglichkeit wäre die genaue Form der Objekte als mathematische Funktion zu definieren. Bei der Schnittüberprüfungen muss die Funktion der eventuell kollidierenden Objekte gleichgesetzt werden und die Gleichung gelöst werden. Es gibt jedoch unendlich viele Formen und Darstellungen von Objekten, sodass einige Körper sich nur schwer oder gar nichts als Funktion beschreiben lassen können. Zusätzlich kann die Berechnung ein sehr rechenintensives Vorhaben sein, welches sich nicht als praktikabel erweisen würde, wenn die Aktualisierungsrate der Simulation darunter leiden müsste. Um die Vielzahl von möglichen Objekten trotzdem effizient auf Überschneidungen zu prüfen, werden Hüllkörper verwendet. Hüllkörper schliessen Objekte ein. Je nach Hüllkörper variiert die Hülleffizienz, die angibt wie gut ein Hüllkörper einen Körper umschliesst. In folgender Abbildung werden vier Hüllkörper vorgestellt. Zu sehen ist ein Hahn mit einem komplexen geometrischen Körper. Er verfügt über Halbkreise, Linien und Dreiecke.

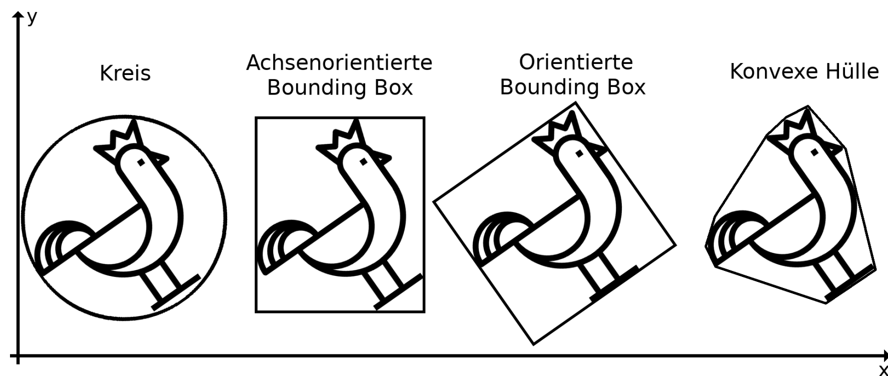

Abbildung 5.1.: Verschiedene Hüllkörper

Der Kreis ist die einfachste Form eines Hüllkörpers. Eine Kollisionserkennung zwischen zwei Kreisen kann sehr effizient realisiert werden. Ein Nachteil ist die schlechte Hülleffizienz. Der Radius des Kreises entspricht dem Abstand vom Mittelpunkt zu dem entferntesten Punkt des Körpers. Die Hülleffizienz hängt wiederum von dem zu umhüllenden Körper ab. In Abbildung 5.1 ist zu erkennen, dass der Hahn den Kreis nicht gesamt ausfüllt. Kollidieren zwei Kreise, wie in Abbildung 5.2 miteinander, kann nicht davon ausgegangen werden, dass die Körper wirklich kollidiert sind. Rotationen des Körpers sind ohne Neuberechnung des Kreises möglich.

Der achsenorientierte Hüllkörper, auch *axis-aligned bounding box, kurz AABB,* genannt,ist die wichtigste Hüllfunktion. Sie umhüllt einen Körper meist besser, als der Kreis. Um den Körper wird ein zu den Achsen des Koordinatensystems ausgerichtetes Rechteck aufgespannt. Hierfür werden der oberste linke und der unterste rechte Punkt des Körpers verwendet. Die Hülleffizienz ist schlecht und mögliche Kollisionen bedeuten nicht, dass die innenliegenden Körper sich wirklich berühren. Die Komplexität hingegen ist sehr gering. Zwei *AABB* schneiden sich, wenn sich die Hüllfunktionen auf allen Achsen überlagern. Eine Rotation des Körpers erfordert eine Neuberechnung des Rechtecks, weil dieses seine Höhe und Breite verändern muss. Hierbei kann die Hülleffizienz leiden.

Die orientierte Hüllfunktion, auch *oriented-bounding-box, kurz OBB*, genannt, ähnelt sehr stark der *AABB*. Anstatt der Orientierung an den Achsen des Koordinatensystems, wird der zu umhüllende Körper zur Orientierung des Rechtecks verwendet. Hierdurch ist es möglich bei Rotationen nur die Eckpunkte des Rechtecks neu zu berechnen, nicht aber die Maße des Rechtecks. Die Erstellung ist schwieriger als die der *AABB* oder des Kreises. Der Körper muss

zum Mittelpunkt des Koordinatensystems verschoben werden, rotiert werden, es müssen die Maximalwerte der Punkte des Körpers gefunden werden und der Körper samt Rechteck muss an die ursprüngliche Position zurücktransformiert werden [24]. Durch den Mehraufwand ist die maximale Hülleffizienz des Rechtecks immer garantiert. Durch die Verbesserung der Hülleffizienz steigt die Komplexität der Kollisionsüberprüfungen. Für die Kollisionsberechnung wird das *Seperating-Axis-Theorem* verwendet [25].

Die konvexe Hülle [26], siehe Abbildung 5.1, weist von allen dargestellten Hüllfunktionen die höheste Hülleffizienz auf. Dies liegt an der Verwendung der kleinsten konvexen Menge für die Hülle. Weil in konvexen Mengen zwei beliebige Punkte immer verbunden werden können müssen, sodass die Verbindungsstrecke in der konvexen Menge liegt, können keine Einbuchtungen in der Hülle entstehen. Dies verschlechtert die Hülleffizienz. Nichtsdestotrotz lässt sich eine genauere Erfassung der Silhouette realisieren. Die Komplexität der Erstellung und der Schnittberechnung sind hingegen deutlich höher als bei vorherigen Hüllkörpern.

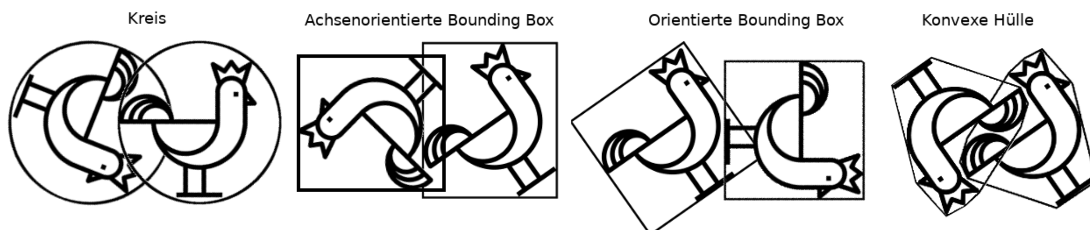

Abbildung 5.2.: Kollisionen verschiedener Hüllkörper

Verschiedene Hüllfunktionen besitzen verschiedene Qualitäten. Je genauer eine Hüllfunktion, desto komplexer ist die Erstellung und die Berechnung von Kollisionen. Wie in Abbildung 5.2 zu sehen ist, bedeutet das nicht, dass Körper sich wirklich schneiden müssen. Um komplexere Körper wie den Hahn aus Abbildung 5.1 präzise kollidieren zu lassen, wird der Körper in weitere Hullkörper aufgeteilt. In Abbildung 5.3 ist der Hahn jeweils von einer AABB und einem Kreis umhüllt. Um nach einer erfolgreichen Kollision mit dem äußeren Hüllkörper die Präzision der Kollision zu erhöhen, wird der Körper des Hahns wie folgt unterteilt. Der Hals bis zum Hahnenkamm, der Körper, die Beine plus Füße und der Schwanz bilden eigene Kollisionszonen. Tritt eine Kollision mit dem äußeren Hüllkörper auf, werden die präziseren Kollisionszonen für weitere Kollisionsüberprüfungen verwendet. In Abbildung 5.3 wird deutlich, dass Kreise im Vergleich zu AABBs für die Unterteilung des Hahns nicht geeignet sind. Für die gleiche Unterteilung wird mehr Platz verwendet, dadurch weisen die Kreise eine niedrigere Hülleffizienz auf.

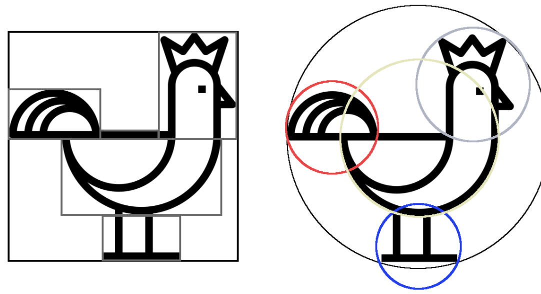

Abbildung 5.3.: Erweiterung verschiedener Hüllkörper mit mehreren gleichartigen Hüllfunktionen für eine höhere Kollisionspräzision

Für die HiL-Simulation muss berücksichtigt werden, dass simple Körper verwendet werden. Somit können konvexe Hüllen ausser Acht gelassen werden. Weiterhin bedarf es keiner Rotation der Objekte, weshalb die orientierte Bounding Box keine Verwendung findet. Für die Simulation werden die ersten beiden Hüllkörper verwendet, der Kreis und die achsenorientierte Bounding Box. Werkstücke benötigen in der zweidimensionalen Vogelperspektive keinen extra Hüllkörper, weil deren Form ohnehin durch einen Kreis beschrieben wird. Die Förderbandbegrenzungen, die Begrenzungen der Rampe, die Förderbandverengung und die Weiche können anhand ihrer Form als *AABB* realisert werden. In Abbildung 4.1 sind die Werkstücke von Rechtecken umgeben. Box2D verwendet für eine schnelle Selektion von möglichen kollidierenden Objekten *AABBs*.

Um Kollisionen zwischen Werkstücken zu erkennen, müssen sich zwei Kreise schneiden. Vorbedingung ist, dass sich die umgebenden *AABBs* überlappt haben. In den folgenden Abschnitten wird gezeigt wie Kollisionsen zwischen *AABBs* und Kreisen berechnet und realisiert werden können.

5.1.1. Schnitt zweier Axis-Aligned-Bounding-Boxen

Ein achsenorientiertes Rechteck kann auf viele Weisen definiert werden. Im untenstehenden Beispiel werden die Rechtecke durch ihre obere linke Ecke - *topLeft* - *tl*, ihre Höhe und Breite definiert. Hiermit lassen sich die übrigen drei Eckpunkte berechnen. Im folgendem Abschnitt *topRight* - *tr*, *bottomLeft* - *bl*, *bottomRight* - *br* genannt. Sei P die obere linke Ecke, h die Höhe und w die Breite des Rechtecks R, dann gilt

$$tl_R = (x_P, y_P),$$
$$tr_R = (x_P + w, y_P),$$
$$br_R = (x_P + w, y_P - h),$$
$$bl_R = (x_P, y_M - h)$$

Damit zwei *AABBs* sich schneiden, muss mindestens ein Punkt der kollidierenden Rechtecke in dem anderen Rechteck positioniert sein. Dadurch überlagern sich X und Y-Achse beider Rechtecke. In folgender Abbildung schneiden sich zwei *AABBs*. Das Rechteck *R1* wird mit $P_{R1} = (1/3)$ mit einer eine Breite von 5 und einer Höhe von 2 Maßeinheiten positioniert. *R2* ist bei $P_{R2} = (4/6)$ mit einer Höhe und Breite von 4 Maßeinheiten positioniert.

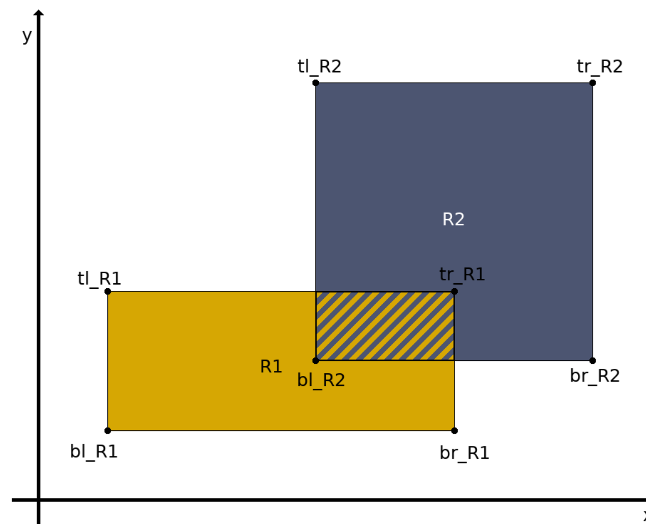

Abbildung 5.4.: Schnitt zweier AABBs

Um herauszufinden, ob sich *R1* und *R2* schneiden, müssen folgende Schnittszenarien überprüft werden. Sind die untenstehenden Bedingungen erfüllt, schneiden sich die Rechtecke.

- Liegt die rechte Seite $\overrightarrow{tr_{R1}br_{R1}}$ von *R1* rechts zu der linken Seite $\overrightarrow{tl_{R2}bl_{R2}}$ von *R2*.
- Liegt die linke Seite $\overrightarrow{tl_{R1}bl_{R1}}$ von *R1* links zu der rechten Seite $\overrightarrow{tr_{R2}br_{R2}}$ von *R2*.
- Liegt die untere Seite $\overrightarrow{bl_{R1}br_{R1}}$ von *R1* unter der oberen Seite $\overrightarrow{tl_{R2}tr_{R2}}$ von *R2*.
- Liegt die obere Seite $\overrightarrow{tl_{R1}tr_{R1}}$ von *R1* über der unteren Seite $\overrightarrow{bl_{R2}br_{R2}}$ von *R2*.

Eine mögliche Implementation könnte wie folgt realisiert werden.

Algorithm 3 Kollisionsüberprüfung zweier *Axis-Aligned-Bounding Boxen*

if ($R1.tr.x$ >= $R2.tl.x$) and ($R1.tl.x$ <= $R2.tr.x$) and ($R1.bl.y$ >= $R2.tl.y$) and ($R1.tl.y$ <= $R2.bl.y$) **then**
    return true
end if
return false

5.1.2. Schnitttest zwischen Kreisen

Um die Kollision von Werkstücken zu berechnen, müssen Kollisionen zwischen Kreisen unterstützt werden. Ein Kreis wird durch seinen Mittelpunkt und seinen Radius definiert. Sei *k* ein Kreis im zweidimensionalen kartesischem Koordinatensystem mit dem Mittelpunkt *M* und dem Radius *r*, dann lautet die allgemeine Kreisgleichung für einen nicht im Ursprung positionierten Kreis mit x_M und y_M als Koordinaten des Mittelpunktes wie folgt.

$$k : (x - x_M)^2 + (y - y_M)^2 = r^2$$

Um herauszufinden, ob zwei Werkstücke *A* und *B* miteinander kollidieren, müssen die Radii der Werkstücke größer als der Abstand der Mittelpunkte sein. Zusätzlich muss der Abstand der Mittelpunkte größer als die Differenz der Radii sein. Somit gilt folgende Bedingung :

$$|r_A - r_B| < |\overrightarrow{M_A M_B}| < r_A + r_B$$

Werkstück *A* sei bei P(20 / 40) positioniert und Werkstück *B* bei P(23 / 42). Die allgemeine Kreisgleichung für die Werkstücke lautet nach 5.1.2 demnach

$$k_A : (x - 20)^2 + (y - 40)^2 = 2^2 \text{ bzw. in Vektorform } k_A : \left[\vec{x} - \begin{pmatrix} 20 \\ 40 \end{pmatrix}\right]^2 = 2^2$$
$$k_B : (x - 23)^2 + (y - 42)^2 = 2^2 \text{ bzw. in Vektorform } k_B : \left[\vec{x} - \begin{pmatrix} 23 \\ 42 \end{pmatrix}\right]^2 = 2^2$$

Der Abstand zwischen den Mittelpunkten kann über die Länge des Verbundvektors beider Mittelpunkte bestimmt werden.

$$|\overrightarrow{M_A M_B}| = |\begin{pmatrix} 23 - 20 \\ 42 - 40 \end{pmatrix}| = \sqrt{3^2 + 2^2} = \sqrt{13} \approx 3.60555$$

Die Summe der Radii ist $r_{A+B} = 4$ und die Differenz $r_{A-B} = 0$. Die Schnittbedingung ist demnach $0 < 3.60555 < 4$ und wird erfüllt. Für die Implementation bietet es sich an die

Quadratwurzel zu vernachlässigen und den quadrierten Abstand mit den quadrierten Radii zu vergleichen.

Algorithm 4 Kollisionsüberprüfung zweier Kreise

circle1, circle2, combined;
combined $\leftarrow circle2 - circle1$;
float distance $\leftarrow combined.x \cdot combined.x + combined.y \cdot combined.y$;
float radii $\leftarrow (circle1.radius + circle2.radius) \cdot (circle1.radius + circle2.radius)$;

if distance < radii **then**
  return true
end if
return false

Um die entstandenen Schnittpunkte *S1* und *S2* in Abbildung 5.5 zu berechnen, müssen die gemeinsamen Punkte der Kreise gefunden werden.

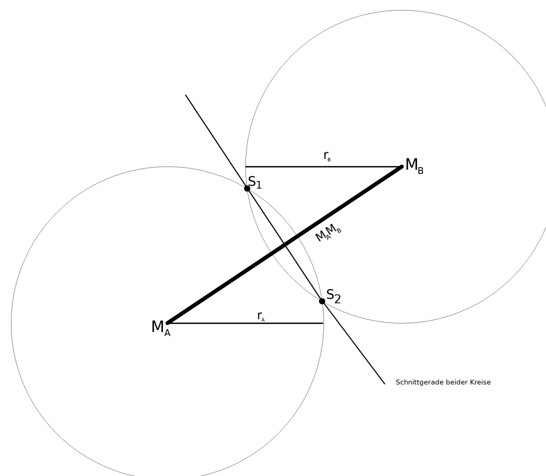

Abbildung 5.5.: Schnitt zweier Kreise

Anhand der resultierenden Geraden, die durch beide Schnittpunkte verläuft, können die gesuchten Koordinaten berechnet werden. Um die Gerade zu berechnen, müssen die Kreisgleichungen gleichgesetzt und umgeformt werden.

$$k_A : (x-20)^2 + (y-40)^2 - 4 = 0 \rightarrow x^2 - 40x + y^2 - 80y + 1996$$
$$k_B : (x-23)^2 + (y-42)^2 - 4 = 0 \rightarrow x^2 - 46x + y^2 - 84y + 2289$$

$$x^2 - 40x + y^2 - 80y + 1996 = x^2 - 46x + y^2 - 84y + 2289 \qquad | \text{ umformen} \tag{5.1}$$
$$-1.5x + 73,25 = y \tag{5.2}$$

Um die Schnittpunkte zu finden, muss das Schnittproblem *Gerade-Kreis* gelöst werden. Die Geradengleichung wird entweder in k_A oder k_B eingesetzt und umgeformt. Anschliessend muss die resultierende quadratische Gleichung, hier mithilfe der PQ-Formel, gelöst werden.

$$(x-20)^2 + (-1.5x + 33,25)^2 - 4 = 0 \qquad | \text{ auflösen} \tag{5.3}$$
$$x^2 - 43x + 462,019230769 = 0 \tag{5.4}$$

$$x_{1,2} = \frac{p}{2} \pm \sqrt{-\frac{p}{2}^2 - q}$$
$$x_1 = \frac{43}{2} + \sqrt{-\frac{43}{2}^2 - 462,019230769} \approx 21.02$$
$$x_2 = \frac{43}{2} - \sqrt{-\frac{43}{2}^2 - 462,019230769} \approx 21.98$$

Um die Schnittpunkte zu berechnen müssen die X-Werte in die Geradengleichung eingesetzt werden.

$$y_1 = -1.5 \cdot 21.02 + 73,25 \approx 41.72$$
$$y_2 = -1.5 \cdot 21.98 + 73,25 \approx 40.28$$
$$S_1(21.02|41.72)$$
$$S_2(21.98|40.28)$$

5.1.3. Schnitttest zwischen Halbgerade und Kreis

Das Transfersystem muss die verfügbaren Lichtschranken simulieren. Für die Lokalisierung von Werkstücken mittels Lichtschranken, kann das Schnittproblem *Halbgerade-Kreis* genutzt werden. Eine Halbgerade, auch Strahl genannt, ist eine auf einer Seite begrenzte Linie durch zwei Punkte. Der erste Punkt definiert den Anfang des Strahls und der zweite Punkt gibt die Richtung des Strahls vor. Die Linie des Strahls erstreckt sich in Richtung des zweiten Punktes in das Unendliche. Sei der Strahl *s* definiert durch zwei Punkte *A* und *B*, dann kann für ein beliebiges *t* jeder Punkt auf dem Strahl definiert werden durch:

$$s(t) = \overrightarrow{OA} + t \cdot \overrightarrow{AB} = \vec{a} + t \cdot \vec{b}$$

Ist t größer als 0, so liegt der Punkt des Strahls vor dem Ursprung in Richtung B. Ist t genau 0, ist dies der Punkt A und somit der Ursprung des Strahls. Ist t kleiner als 0, liegt der Punkt hinter dem Strahl. Für den Schnitttest kann die Kreisgleichung aus 5.1.2 verwendet werden. In Vektorform lautet diese wie folgt :

$$k : \left[\begin{pmatrix} x \\ y \end{pmatrix} - \begin{pmatrix} x_M \\ y_M \end{pmatrix} \right]^2 = r^2 \Rightarrow \left[\vec{x} - \vec{m} \right]^2 = r^2$$

Um auf mögliche Schnittstellen zu prüfen, muss die Strahlgleichung in die Kreisgleichung substituiert werden. Weil die vektorielle Darstellungsweise der Kreisgleichung einem Skalarprodukt entspricht, kann die substituierte Form wie folgt aufgelöst werden.

$$\left[\left(\vec{a} + t\vec{b} \right) - \vec{m} \right]^2 - r^2 = 0 \qquad | \text{ ausmultiplizieren} \tag{5.5}$$

$$\vec{a}^2 + 2\vec{a}\vec{b}t + t^2\vec{b}^2 - 2\vec{a}\vec{m} - 2t\vec{b}\vec{m} + \vec{m}^2 - r^2 = 0 \qquad | \text{ umformen} \tag{5.6}$$

$$\vec{b}^2 t^2 + 2\vec{b}\left(\vec{a} - \vec{m}\right)t + \left(\vec{a} - \vec{m}\right)^2 - r^2 = 0 \tag{5.7}$$

Die obenstehende Gleichung hat die Form einer quadratischen Gleichung $f(x) = at^2 + bt + c$. Die Koeffizienten sind somit $a = \vec{b}^2$, $b = 2\vec{b}\left(\vec{a} - \vec{m}\right)$, $c = \left(\vec{a} - \vec{m}\right)^2 - r^2$. Mithilfe der Mitternachtsformel [27] können die Schnittstellen des Strahls und des Kreises berechnet werden. Wenn der Richtungsvektor des Strahls normiert ist, hat dieser eine Länge von 1. Der Koeffizient a wird somit zu 1 und die Mitternachtsformel hat demnach folgende Form.

$$t_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \Rightarrow \frac{-b \pm \sqrt{b^2 - 4c}}{2} \tag{5.8}$$

Anhand der Diskriminante kann bestimmt werden, ob der Strahl den Kreis schneidet. Die Diskriminante ist der unter der Wurzel stehende Term. Ist die Diskriminante $\Delta = b^2 - 4c$ größer als Null, schneidet der Strahl den Kreis an zwei Punkten. Ist sie genau Null existiert nur eine Schnittstelle. Wenn die Diskriminante kleiner als Null ist, existieren keine Schnittpunkte. Der Strahl verfehlt den Kreis. Im folgendem Beispiel schneidet ein Strahl einen Kreis. Die Strahlgleichung s besitzt einen normierten Richtungsvektor. Der Mittelpunkt M des Kreises und der Radius lauten wie folgt.

$$s(t) = \begin{pmatrix} 5 \\ 1 \end{pmatrix} + t \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix}, M = \begin{pmatrix} 4 \\ 4 \end{pmatrix}, r = 2.$$

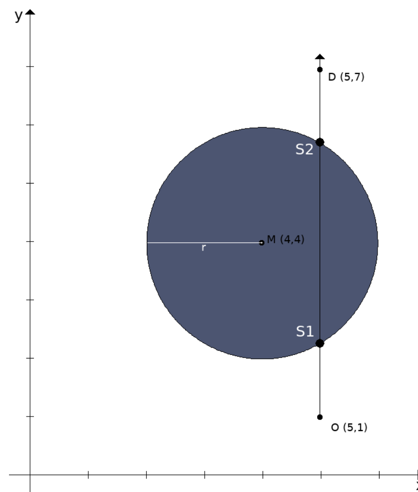

Abbildung 5.6.: Schnitt Halbgerade-Kreis

Die Koeffizienten der quadratischen Gleichung lauten somit :

$$a = \vec{b}^2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}^2 = 1 \tag{5.9}$$

$$b = 2\vec{b}(\vec{a} - \vec{m}) = 2 \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} (\begin{pmatrix} 5 \\ 1 \end{pmatrix} - \begin{pmatrix} 4 \\ 4 \end{pmatrix}) = -6 \tag{5.10}$$

$$c = (\vec{a} - \vec{m})^2 - r^2 = (\begin{pmatrix} 5 \\ 1 \end{pmatrix} - \begin{pmatrix} 4 \\ 4 \end{pmatrix})^2 - 4 = 6 \tag{5.11}$$

Die Diskriminante Δ lautet für oben stehende Funktion $(-6)^2 - 4 \cdot 6 = 12$. Es existieren somit zwei Schnittpunkte S_1, S_2.

$$t_1 = \frac{-b-\sqrt{b^2-4c}}{2} = \frac{6-\sqrt{12}}{2} = 3-\sqrt{3} \approx 1.268 \quad (5.12)$$

$$t_2 = \frac{-b+\sqrt{b^2-4c}}{2} = \frac{6+\sqrt{12}}{2} = 3+\sqrt{3} \approx 4.732 \quad (5.13)$$

$$S_1 = \begin{pmatrix} 5 \\ 1 \end{pmatrix} + 3-\sqrt{3} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 5 \\ 4-\sqrt{3} \end{pmatrix} \approx \begin{pmatrix} 5 \\ 2.268 \end{pmatrix} \quad (5.14)$$

$$S_2 = \begin{pmatrix} 5 \\ 1 \end{pmatrix} + 3+\sqrt{3} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 5 \\ 4+\sqrt{3} \end{pmatrix} \approx \begin{pmatrix} 5 \\ 5.732 \end{pmatrix} \quad (5.15)$$

Für eine Implementation sollte statt der *PQ-Formel* aus Abschnitt 5.1.1 die *Mitternachtsformel* verwendet werden. Mithilfe der Diskriminante können Wurzelberechnungen umgangen werden, weil sich herausfinden lässt, ob der Strahl den Kreis überhaupt schneidet. Dies ermöglicht eine vorzeitige Abbruchbedingung. Eine Implementation für das Schnittproblem könnte wie folgt aussehen.

Algorithm 5 Schnitttest - Strahl Kreis

```
Vector origin, direction, circleCenter
float radius;
Vector t ⇐ origin − circleCenter;
float a ⇐ 2 · DotProduct(direction, direction);
float b ⇐ 2 · DotProduct(t, direction);
float c ⇐ DotProduct(t, t) − (radius · radius);
flaot discriminant ⇐ (b · b) − (4 · a · c);
if discriminant < 0 then
    return false;
end if
float t1, t2, sqrtDisc;
sqrtDisc ← sqrt(discriminant)
t1 ← (−b − sqrtDisc)/2;
t2 ← (−b + sqrtDisc)/2;
if t1 < 0 then
    if t2 < 0 then
        return false;
    end if
end if
return true;
```

5.1.4. Schnitttest zwischen Kreis und AABB

Damit Werkstücke mit den Förderbandbegrenzungen des Transfersystems kollidieren, muss eine Kollisionserkennung zwischen *AABB* und Kreis verfügbar sein. Der Kreis *K1* wird beschrieben durch den Radius $r = 4$ und seinen Mittelpunkt *m* mit der Dartellung P(x / y). Die *AABB* wird durch seine Länge, Breite und durch den oberen linken Eckpunkt *tl* definiert.

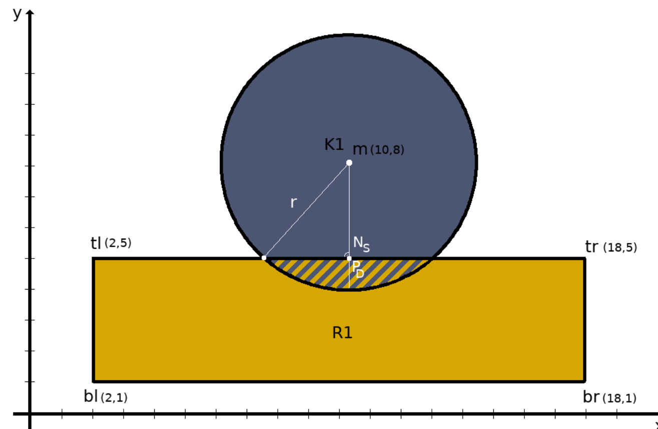

Abbildung 5.7.: Kollisionen zwischen Kreis und AABB

Für die Kollisionsüberprüfung muss herausgefunden werden, ob der naheliegendste Punkt vom Rechteck zum Kreis kleiner oder größer als der Radius des Kreises ist. Dieser Punkt kann über das sogenannte *Clamping* berechnet werden.

$$x_{nearest} = Max(tl_x, Min(m_x, tr_x))$$
$$y_{nearest} = Max(bl_y, Min(m_y, tl_y))$$

$$x_{nearest} = Max(2, Min(10, 18)) = Max(2, 10) = 10$$
$$y_{nearest} = Max(1, Min(8, 5)) = Max(1, 5) = 5$$
$$S_{Nearest} = (10/5)$$

Mithilfe des gefundenen Punktes kann die Distanz zum Mittelpunkt *m* über die Länge des Vektors $\overrightarrow{N_s m}$ bestimmt werden.

$$Distance = |\overrightarrow{N_s m}| = |\begin{pmatrix} 0 \\ -3 \end{pmatrix}| = \sqrt{0^2 + (-3)^2} = 3$$

Anhand der *Penetrationstiefe* kann bestimmt werden, wie weit ein Körper einen anderen Körper überschneidet. Für die Überschneidungen muss in diesem Beispiel lediglich die Distanz vom

Radius subtrahiert werden. $P_{Depth} = r - Distance = 4 - 3 = 1$. Falls die Schnittpunkte des Kreises mit dem Rechteck benötigt werden, können diese wie in 5.5 berechnet werden. Grundlage der Gerade bilden die zum nahesten Punkt passenden Eckpunkte oder der Normalvektor des Punktes. In diesem Beispiel *tl* und *tr*.

Eine Implementation sollte auf die Wurzelberechnung verzichten und stattdessen die quadrierten Werte vergleichen.

Algorithm 6 Kollisionsüberprüfung zwischen einem Kreis und einer *AABB*

$nearestX \Leftarrow Max(AABB.x, Min(Circle.x, AABB.x + AABB.width));$
$nearestY \Leftarrow Max(AABB.x - AABB.height, Min(Circle.y, AABB.y));$
$deltaX \Leftarrow Circle.x - nearestX$
$deltaY \Leftarrow Circle.y - nearestX$
return $(deltaX \times deltaX + deltaY \times deltaY) < (Circle.radius \times Circle.radius)$

5.2. Kollisionsberechnung

Zeitdiskrete Simulationen verwenden eine festgelegte Aktualisierungsrate. Mithilfe der Aktualisierungsrate kann eine neue Position eines Objektes für den nächsten Simulationsschritt *t+1* berechnet werden. Je länger ein Zeitintervall, desto größer die zurückgelegte Distanz von Objekten. In die Berechnung der Neupositionierung müssen die Kräfte, die auf den Körper wirken, miteinbezogen werden. Als Beispiel können Geschwindigkeit, Reibung als entgegenwirkende Kraft, eventuelle gleichzeitige Kollisionen, die das Objekt von seiner eigentlichen Position verschieben, genannt werden. Zeitorientierte Kollisionsberechnung birgen das Problem des Tunneleffektes. Als Beispiel soll die in Abbildung 5.8 aufgezeigte Situation dienen. Ein Ball fliegt mit einer hohen Geschwindigkeit auf eine Wand zu. Der Ball ist sehr schwer, sodass die Gravitation den Ball in Richtung Boden zieht. Der Ball befindet sich zum Zeitpunkt *t* kurz vor der Kollision mit der Wand. Zum Zeitpunkt *t+1* sollte der Ball wie in Skizze *2* von der Wand abgeprallt sein und sich nah an der Wand befinden. Durch die hohe Geschwindigkeit des Balls ensteht eine große Schrittweite für das Objekt. Der Ball überspringt die Wand und mögliche Kollisionen werden nicht berücksichtigt.

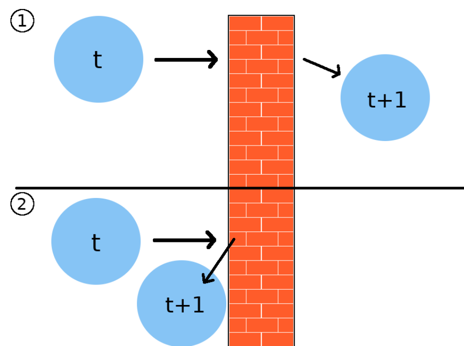

Abbildung 5.8.: Tunnelingproblem

Im folgendem Abschnitt werden die grundlegenden Ideen der Kollisionserkennung vorgestellt und es wird gezeigt, wie das *Tunnelingproblem* behoben werden kann.

5.2.1. Strategien für die Erkennung

Um die Komplexität der Kollisionberechnung zu reduzieren, wird für die Erkennung von Kollisionen ein zweiphasiger Ansatz verwendet. Dieser teilt sich in *Broadphase* und *Narrowphase* auf. In der *Broadphase* werden grob alle mögliche Kollisionen zwischen Objekten erkannt und die kollidierenden Paare gespeichert. Hierfür werden ressourcensparende Hüllfunktionen wie die *AABB* aus Abschnitt 5.1.1 verwendet. Diese haben zwar eine schlechte Hülleffizienz, sind für die Selektion jedoch schnell zu erstellen und zu berechnen. Durch die Selektion werden die nötigen Kollisionen für den Simulationsschritt *t+1* drastisch reduziert. Hierdurch lassen sich weitere unnötige Schnitttests in der nächsten Phase vermeiden. Die *Narrowphase* verwendet die detaillierteren Hüllkörper und evaluiert welche Objekte an welchen Stellen miteinander kollidieren. Diese Phase ist rechenintensiver als die *Broadphase*, weil Kollisionen detaillierter berechnet werden müssen. Aus diesem Grund müssen alle unnötigen Kollisionen in der *Broadphase* vorab entfernt werden, andernfalls könnten die Berechnungen länger als die eigentliche Zeitspanne des Aktualisierungsintervalls dauern.

Die nun folgenden Ansätze sollen nur grob umrissen werden, weil detaillierte Darstellung von Methoden und Problemen Themen vieler Forschungsarbeiten sind.

Broadphase

Der einfachste Weg Kollisionen zu erkennen, wäre mit einer verschachtelten Schleife über alle verfügbaren Objekte. Dies könnte wie folgt realisiert werden.

Algorithm 7 Kollisionsüberprüfung - Bruteforce

```
pairsOfCollidingObjects, objects, currentObject, otherObject
for i = 0; i < objects.length; i++ do
    currentObject ⇐ objects[i]
    for j = i+1; j < objects.length; j++ do
        otherObject ⇐ objects[j]
        if currentObject.intersects(otherObject) then
            pairsOfCollidingObjects.add(currentObject, otherObject)
        end if
    end for
end for
```

Die Summe der aufgerufenen Kollisionsüberprüfungsroutinen beläuft sich mit oben stehendem Pseudocode mit $k = 16$ zu prüfenden Objekten auf $\sum_{n=1}^{k-1} n = 120$. Eine Überprüfung von $k = 8192$ Objekten würde $\sum_{n=1}^{k-1} n = 33550336$ Kollisionsüberprüfungsroutinen aufrufen. Angenommen eine Kollisionsüberprüfungsroutine dauere eine Nanosekunde, so würde die gesamte Berechnung ohne Kollisionsauflösung 33,5 Millisekunden dauern. Für den restlichen Simulationszyklus blieben somit noch 66,5 Millisekunden zur Verfügung. Mit einem Bruteforceansatz könnte die HiL-Simulation unbrauchbar werden.

Weil die Erstellung von möglichen Kollisionspaaren ein Bottleneck für die Kollisionsbehandlung werden könnte, müssen optimierte Algorithmen verwendet werden. Box2D verwendet für diesen Zweck einen *dynamischen AABB-Baum*. Ziel ist die Reduzierung der Laufzeit des Bruteforceansatzes, um in der rechenintensiveren *Narrowphase* mehr Rechenzeit zur Verfügung zu haben. Der *AABB-Baum* verwaltet alle in der Welt existierenden *AABBs*. Aufgrund der zugrundeliegenden Datenstruktur eines Binärbaums, kann eine Liste von potentiellen Kollisionen mithilfe einer In-Order Traversierung mit einer Komplexität von $\mathcal{O}(n)$ erstellt werden [28].

In dem *AABB-Baum* werden die Objekte aus der Welt mit ihren *AABBs* in den Blättern gespeichert. Die darüberliegenden Elternknoten umhüllen mit einer weiteren *AABB* die Hüllkörper der Kindsknoten. Die Wurzel umhüllt somit die gesamte Szene mit allen *AABBs*. Für eine Kollisionsüberprüfung genügt eine Berechnung der Kollisionen zwischen den Blättern eines Elternknoten. Falls ein Elternknoten ein Blatt und einen weiteren Elternknoten aufweist, müssen für die Berechnungen die Blätter des Kindsknoten verwendet werden. Eine detaillierte Implementierung bietet Randy Gaul [29].

Narrowphase

Wurden in der *Broadphase* mögliche Kollisionen erkannt, müssen sie auf tatsächlich zutreffende Kollisionen untersucht werden. Diese Kollisionen sind detaillierter und beziehen sich auf die in Abbildung 5.3 gezeigten Hüllkörper. Um Kollisionen zu generieren, müssen sich Objekte bewegen. Anhand der zeitlich festgesetzten Aktualisierungsrate der HiL-Simulation, können die neuen Positionen der Objekte, mithilfe der auf sie wirkenden Kräften bestimmt werden. Es haben sich zwei Verfahren zur Berechnung von Kollisionen etabliert. *A priori* und *a posteriori*.

5.2.2. A posteriori Verfahren

Beim *a posteriori* Verfahren, werden Objekte zunächst neu positioniert bevor auf Kollisionen getestet wird. Dies bedeutet, dass ein Objekt zuerst mit seiner berechneten Schrittweite versetzt wird und danach auf mögliche Kollisionen geprüft wird. Ist keine Kollision entstanden, behält das Objekt seine neue Position. Folgende Abbildung zeigt die Vorgehensweise, falls eine Kollision aufgetreten ist.

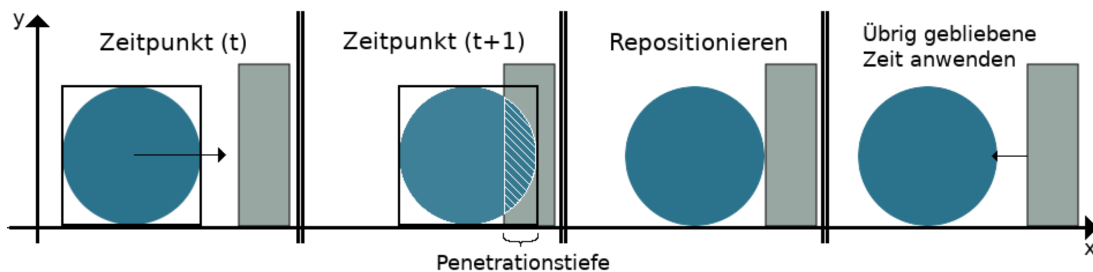

Abbildung 5.9.: A posteriori Verfahren

Von der Position des Objektes zum Zeitpunkt *t* wird die mögliche Schrittweite für die nächste Iteration berechnet. Diese wird auf die Position addiert. Hieraus resultiert eine Überschneidung mit der in grau gekennzeichneten Wand zum Zeitpunkt *t+1*. Um die Kollision aufzulösen, muss die Penetrationstiefe des Kreises mit der Wand berechnet werden. Mithilfe des Aktualisierungsintervalls und der Penetrationstiefe kann die übrig gebliebene Zeit Δt für weitere Positionsaktualisierungen berechnet werden. Das Objekt muss repositioniert werden, sodass es die Wand minimal penetriert. Nun muss der Kräftevektor des Objekts nach der Kollision berechnet werden. Hier finden die physikalischen Berechnungen der eigentlichen Kollision statt. Anschließend wird das Objekt mit der übrig gebliebenen Zeit Δt des Aktualisierungsintervalls und dem neu berechnetem Kräftevektor repositioniert.

Ein Vorteil ist die einfache Realisierung des Verfahrens. Falls Kollisionen auftreten, ist das einzige Problem die Repositionierung der kollidierenden Objekte. Um diesem Problem entgegenzuwirken, kann entweder der exakte Kollisionpunkt berechnet werden oder es wird eine *Backoffstrategie* verwendet. Hierbei muss beachtet werden, dass Ungenauigkeiten durch die verwendeten Zahlentypen und deren Präzision entstehen können. Aus diesem Grund sollte die minimale Penetrationstiefe mit einem Schwellwert arbeiten. Bei der *Backoffstrategie* wird das Objekt um einen variablen Wert solange zurückgesetzt, bis keine Kollision mehr auftritt. Die Schrittweite des *Backoffs* verkleinert sich bei zunehmender Anzahl der Versetzungen. Die Anzahl der Versetzungen des Objektes sollte begrenzt werden, weil das Verfahren durch die immer kleiner werdenden Intervalle nicht terminieren könnte. Problematisch ist die mögliche Ungenauigkeit beim Umsetzen des Objektes. Es kann passieren, dass nach dem ersten Versetzen keine Kollision mehr vorhanden ist, die Distanz zwischen den kollidierten Objekten jedoch größer als gewünscht ist.

Das größte Problem ist das Auftreten vom *Tunneling*. Wie in Abbildung 5.8 dargestellt, können bei zu schnellen Objekten Hüllkörper übersehen werden. Um dieses Problem zu beheben, wird die kontinuierliche Kollisionserkennung, auch *a priori* oder *Continous Collision Detection - CCD* genannt, verwendet.

5.2.3. A priori Verfahren

Ein schwieriges Vorhaben ist die Erkennung von dynamischen Objekten mit hohen Geschwindigkeiten. Wenn ein Objekt mit einer sehr hohen Geschwindigkeit, wie zum Beispiel ein Projektil einer Pistole, auf ein anderes Objekt mit einer sehr hohen Geschwindigkeit treffen soll, muss höheste Präzision für die Erkennung vorliegen. Ein weiteres Beispiel wäre ein Projektil, welches auf einen statischen Körper, wie eine Wand, gefeuert wird. Die Erwartungshaltung, dass das Projektil die Wand trifft kann mit dem *a posteriori* Verfahren nicht garantiert werden. Um die Kollision mit solch schnellen Objekten zu ermöglichen, wird das *a priori* Verfahren verwendet. Die kontinuierliche Kollisionserkennung ist ein weites Forschungsgebiet bei dem die Präzision unter Berücksichtigung der Laufzeit im Vordergrund steht. Aus diesem Grund wird im folgendem Abschnitt die von Box2D verwendete Kollisionsstrategie umrissen.

Box2D verwendet eine Unterteilung des Aktualisierungsintervalls. Das sogenannte *Substepping* [22]. Potentielle Kollisionen werden in kürzeren Zeitabschnitten überprüft, bevor das Objekt seine eigentliche Position verändert. Folgende Abbildung veranschaulicht das Verfahren.

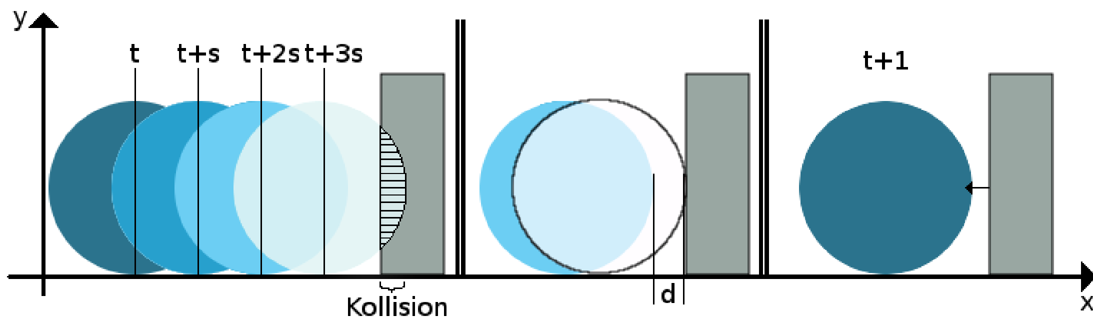

Abbildung 5.10.: A priori Verfahren

Das Aktualisierungsintervall wird in kleinere Zeitscheiben *s* unterteilt. Von der aktuellen Position *t* wird für den Folgeschritt überprüft, ob Kollisionen auftreten. Entsteht keine Kollision, wird für die nächste Überprüfung die Position *t+s* verwendet. Dies funktioniert bis zum Zeitpunkt *t+2s*. Der nächste Schritt *t+3s* generiert eine Kollision. Nun muss die Distanz *d* mit der minimalen Penetrationstiefe zum Kollisionsobjekt berechnet werden. Mithilfe der Ausgangsposition zum Zeitpunkt *t* und der Position mit der minimalen Penetrationstiefe lässt sich die *Time of Impact (TOI)* berechnen. Dies ist die Zeit zu der die Objekte kollidieren würden. Das Objekt wird anschließend mit der übrig gebliebenen Zeit $\Delta t = Aktualisierungsintervall - TOI$ von der Position $t + 3s - d$ aktualisiert. Die Position für den nächsten Iterationsschritt *t+1* wurde gefunden. Das *Substepping* kann noch weiter unterteilt werden, sodass nach der Berechnung ab Zeitschritt *t+3s* weitere Kollisionen gesucht werden. Dies ist nur nötig, wenn in der *Broadphase* weitere potentielle Kollisionen erkannt worden sind.

Dieser Ansatz ist sehr rechenintensiv durch die wiederholenden Überprüfungen und sollte nur für schnelle Objekte verwendet werden. Die HiL-Simulation besitzt keine sich derartig schnell bewegenden Körper. Um die kontinuierliche Kollisionerkennung mit *Substepping* performanter zu gestalten, können die verwendeten Schritte, wie bei einer binären Suche, unterteilt werden. Wenn die Zeitpunkte *t* und *t+1* das Aktualisierungsintervall aufspannen, wird zum Zeitpunkt *t+0.5* überprüft, ob eine Kollision auftrat. Trat keine Kollision auf, wird der Zeitschritt verkleinert und die Position zum Zeitpunkt *t+0.75* wird überprüft. Trat nun eine Kollision auf, wird der Zeitschritt um die Hälfte des zuvorgenutzten Schrittes verkürzt. Somit würde eine erneute Kollisionsüberprüfung zum Zeitpunkt *t+0.625* ausgeführt werden. Dieser Vorgang wiederholt sich solange bis sich die Objekte mit einem definierten Schwellwert überlagern [30].

6. Systemarchitektur

Im folegendem Kapitel werden Architektur und Designentscheidungen der HiL-Simulation vorgestellt. Um einen detaillierteren Überblick über die Architektur zu erhalten, wird diese in zwei Funktionsblöcke unterteilt. Den Simulationskern und den Kommunikationskern. Für beide Funktionsblöcke werden zuerst die nötigen Funktionseinheiten vorgestellt. Diese bilden im Zusammenspiel die Gesamtfunktion des jeweiligen Funktionsblockes.

6.1. Simulationskern

Der Simulationskern ist verantwortlich für die Simulation des Festo-Transfersystems. Er abstrahiert zusätzlich einen Benutzer und ermöglicht es somit dessen Interaktion mit dem Transfersystem nachzuahmen. Der Simulationskern ist für die Instanziierung der Simulationselemente zuständig. Hierzu zählen die Werkstücke, die Sensorik als auch die Aktorik. Die Simulationselemente generieren Daten oder ändern ihren Zustand anhand des Datenbestandes des DataHandlers, siehe Abschnitt 6.1.3. Damit der Gesamtzustand der HiL-Simulation erfasst werden kann, verfügt der Simulationskern über eine zentrale Datenerfassungskomponente. Um die Simulation zu aktualisieren, wird ein vorgegebenes Aktualisierungsintervall verwendet. In jedem Aktualisierungsintervall wird der Gesamtzustand der HiL-Simulation neu berechnet. Um diese Funktionalität zu realiseren, muss der Simulationskern in eigenständige Funktionseinheiten mit definierten Aufgaben unterteilt werden. Folgende Abschnitte beschreiben die oben genannten Funktionseinheiten. Diese werden anhand der Abhängikeiten zueinander vorgestellt.

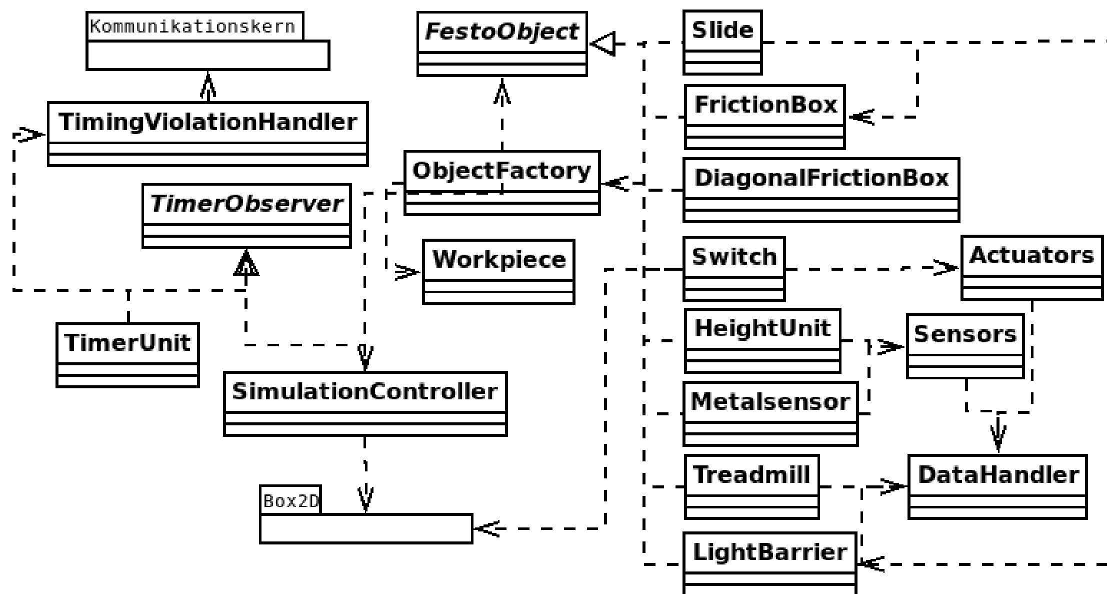

Abbildung 6.1.: Simulationskern - Übersicht

6.1.1. TimerUnit und TimingViolationHandler

Damit die Simulation periodisch mit einer festgesetzten Aktualisierungsfrequenz gesteuert werden kann, wird ein Zeitgeber verwendet. Dieser verwendet eine Aktualisierungfrequenz von 100 Millisekunden. Jede Funktionseinheit, die zu den Aktualisierungsintervallen benachrichtigt werden möchte, muss bei dem Zeitgeber registriert werden. Voraussetzung hierfür ist, dass diese das *TimerObserver-Interface* aufweisen. *TimerObserver* besitzen nur die Funktionalität einer Aktualisierungsroutine, die in dem jeweiligen *TimerObserver* definiert werden muss.

Weil der Zeitgeber die Informationen über die Laufzeit der Aktualisierung der registrierten *TimerObserver* besitzt, hat dieser die zusätzliche Funktion der Fehlerberichterstattung. Die Fehlerberichterstattung wird von dem *TimingViolationHandler* übernommen. Der Zeitgeber muss bei einer Überschreitung der Laufzeit dem *TimingViolationHandler* davon Bericht erstatten. Dieser meldet dem Kommunikationskern, siehe Abschnitt 6.2, dass die Laufzeit länger als das Aktualisierungsintervall war und löst einen Fehlerfall aus. Die Nachrichtensysteme benachrichtigen anschliessend die angemeldeten Klienten.

6.1.2. Simulationscontroller

Weil das Simulieren des Festo-Transfersystems in der Welt von Box2D realisiert wird, muss es eine Möglichkeit der Manipulation für diese geben. Der Simulationscontroller übernimmt diese Aufgabe und verwaltet die registrierten Objekte in der Welt von Box2D, siehe Abschnitt 6.1.6. Seine Hauptaufgabe besteht in der Erstellung und Aktualisierung des digitalen Abbildes des Transfersystems. Hierfür implementiert der Simulationscontroller das *TimerObserver-Interface* und registriert sich bei Instanziierung bei der *TimerUnit*. Hierdurch werden periodische Aktualisierungen der Welt von Box2D möglich

Für den Aufbau der Welt verwendet der Simulationscontroller die *ObjectFactory*, siehe 6.1.5. Die instanziierten Objekte werden sowohl in der Welt von Box2D, als auch im Simulationscontroller gewartet. Dies bedeutet nicht, dass Komponenten doppelt gespeichert werden. Objekte sind Wrapper mit Metadaten für die von Box2D benötigten Enginestrukturen. Für den Aufbau des Transfersystems werden Förderbandbegrenzungen, Förderband, Lichtschranken, Metallsensor, Höhensensor, Weiche und Rampe benötigt. Um diese Komponenten zu aktualiseren, weisen sie die Schnittstelle eines *FestoObjects* auf. Über die Aktualisierungsroutine eines *FestoObjects* kann dessen Zustand neu berechnet werden.

Damit Werkstücke ihren Weg auf das Förderband finden, muss der Simulationscontroller das Hinzufügen von Werkstücken unterstützen. Hierfür wird der Werkstückstyp, die Position und die Orientierung des hinzuzufügenden Werkstückes benötigt. Bevor ein Werkstück dem Laufband hinzugefügt wird, muss überprüft werden, ob die gewünschte Position gültig ist und sich kein anderes Werkstück dort befindet und die Position auf dem Förderband liegt. Um Werkstücke zu entfernen, wird die Identifikationsnummer des jeweiligen Werkstückes benötigt. Die letzte mögliche unterstützte Interaktion mit Werkstücken ist das Verändern der Orientierung. Dieser Vorgang ahmt das Umdrehen eines Werkstückes nach und erzwingt eine Änderung der Orientierung. Um die Orientierung eines Werkstückes zu ändern, wird die entsprechende Identifikationsnummer des Werkstückes benötigt. Der Simulationscontroller muss zusätzlich alle Werkstücke auf Abfrage bereitstellen können. Dies wird durch die Rückgabe einer Liste aller Werkstücke realisiert.

Weil der Simulationscontroller die registrierten Objekte der Welt verwaltet, muss dieser ein Zurücksetzen der Welt unterstützen. Hierfür bietet Box2D eine Resetmethode an, in der alle momentanen wirkenden Kräfte zurückgesetzt werden. Zusätzlich werden alle erzeugten Werk-

stücke gelöscht, sodass nur das aufgebaute Festo-Transfersystem in der Welt bestehen bleibt. Ein Reset erzwingt die Herstellung des Initialzustands der Register aus Abbildung 3.1.

6.1.3. Repräsentation der Register - DataHandler

Der DataHandler ist die zentrale Instanz für die Speicherung der Zustände der Aktorik und Sensorik der HiL-Simulation. Aus diesem Grund gibt es nur eine Instanz. Die Aktorik und Sensorik setzen sich aus den Registern A, B und C, siehe 3.1, sowie dem Höhenmessergebnis des Höhensensors zusammen. Um die Register zu verändern, verfügt der DataHandler für jedes Register eine eigene Funktion mit der eine logische Konjunktion oder Disjunktion des jeweiligen Registers mit einer Bitmaske ermöglicht wird. Um die Register auszulesen, können diese seperat angefordert werden. Weil der Kommunikationskern zeitgleich während eines Aktualisierungsintervalls die Werte der Register auslesen muss, werden Schattenregister verwendet. Alle schreibenden Operationen werden während des Simulationszyklus auf den Schattenregistern ausgeführt. Alle lesenden Zugriffe auf den eigentlichen Registern. Dies bedeutet, dass die Darstellung der Zustände der Register immer um einen Aktualisierungsintervall verzögert ist.

6.1.4. DataHandler-Adapter

Um die semantische Aussage der Binärwerte der Register des DataHandlers zu erhöhen [31], verwendet die HiL-Simulation den strukturellen Ansatz eines Adapters [32]. Zweck ist es die Verwendung des DataHandlers zu vereinfachen. Dieser stellt eine Sammlung von Funktionen dar, mit denen die jeweiligen Bits der Register des DataHandlers gelesen oder beschrieben werden können. Es werden zwei Adapter verwendet. *Sensors* umhüllt Aufrufe für Port B und C und kann die Höhenwerte manipulieren, *Actuators* umhüllt Aufrufe für Port A und C.

6.1.5. ObjectFactory

Der Simulationscontroller soll nicht zusätzlich die Funktion der Erstellung verschiedener Objekte aufweisen. Damit die Zuständigkeitsbereiche sich nicht überschneiden, übernimmt die ObjectFactory die Funktion der Generierung von Simulationsobjekten. Hierzu zählen sowohl die Werkstücke, als auch die für die Erstellung der Welt nötigen *FestoObjects*. Die ObjectFactory muss das Erstellen aller Werkstückstypen unterstützen. Hierzu müssen die Körper in Box2D definiert werden. Weil die dreidimensionale Welt auf eine zweidimensionale Weltanschauung abstrahiert wurde, müssen zwei Repräsentationsformen aller Werkstücke definiert sein. Für die *FestoObjects*, also Förderbandbegrenzungen, Förderband, Lichtschranken, Metallsensor,

Höhensensor, Weiche und Rampe, werden die in Abschnitt 2.1.5 definierten Positionen und Längen verwendet.

6.1.6. Komponenten

Komponenten sind die eigentlichen Funktionseinheiten der HiL-Simulation. Sie erzeugen entweder neue Daten oder sie ändern ihren Zustand anhand des bestehenden Datenbestandes des *DataHandlers*. Folgender Abschnitt stellt die Funktionalitäten und Eigenschaften aller Komponenten vor.

Werkstück

Werkstücke sind die zu befördernden Objekte in dem Festo-Transfersystem. Sie weisen je nach Typ die in Abschnitt 2.1.1 aufgezeigten Merkmale auf. Damit Werkstücke mit ihrer Umgebung interagieren können, benötigen sie einen von Box2D vorgegebenen Körper. Mithilfe des Körpers können Sensoren die Werkstücke erfassen. Zusätzlich wird die Hauptfunktionalität von Kollisionen zwischen den Werkstücken und den Förderbandbegrenzungen ermöglicht. Für eine eindeutige Identifikation besitzt jedes Werkstück eine Identifikationsnummer. Um die Informationen des Werkstückes einzusehen, können diese von dem Werkstück abgefragt werden. Das Werkstück agiert als Hülle für die von Box2D benötigten Daten und ermöglicht somit eine einheitliche Datenhaltung.

Weiche

Die Funktionalität der Weiche beschränkt sich auf eine Veränderung seines in Box2D registrierten Körpers. Entweder versperrt die Weiche das Förderband oder sie tut es nicht. Um in Erfahrung zu bringen in welcher Weichenstellung die Weiche für den aktuellen Simulationszyklus sein soll, wird das zugehörige Bit des entsprechenden Ports vom *DataHandler* mithilfe eines *DataHandler-Adapters* bei jeder Aktualisierung abgefragt. Je nach vorliegendem Zustand wird die Weichenstellung angepasst. Um eine Aktualisierung zu ermöglichen weist die Weiche die Schnittstelle des *FestoObjects* auf.

Metallsensor

Die Aufgabe des Metallsensors ist die Erkennung von Metall in Werkstücken. Der Metallsensor wird als *FestoObject* behandelt und aktualisert seinen Zustand erst nach Aufforderung vom Simulationscontroller. Hierfür muss der Metallsensor die Positionen, der auf dem Förderband platzierten Werkstücke, vom Simulationscontroller abfragen. Befindet sich ein Werkstück mit

dem Werkstückstypen Metall in einem vorgegebenen Bereich, wird das entsprechende Bit im *DataHandler* mittels *DataHandler-Adapter* verändert.

Lichtschranken

Lichtschranken müssen Unterbrechungen des von ihnen ausgesendeten Lichtstrahls erkennen. Nur Werkstücke sind in der Lage Lichtschranken zu unterbrechen. Die Lichtschranken werden anhand ihrer Positionierung in Typen unterteilt. Jeder Typ repräsentiert eine Lichtschranke der in Abschnitt 2.1.5 vorgestellten Bereiche des Transfersystems. Um den Gesamtzustand der HiL-Simulation zu manipulieren, werden bei Zustandsänderungen der Lichtschranke die korrespondierenden Bits im DataHandler angepasst. Eine Aktualisierung erfolgt über den Simulationscontroller mithilfe der FestoObject Schnittstelle. Damit die Lichtschranke unterbrochen werden kann, besitzt sie eine von Box2D vorgegebene Strahlenkomponente mit der Schnitte zwischen Strahl und Körper berechnet werden können.

Kollisionsboxen

Damit Werkstücke nicht ungewollt vom Förderband oder der Rampe fallen, werden Kollisionsboxen verwendet. Die HiL-Simulation teilt die Kollisionsboxen anhand ihrer Form auf. Für das Festo-Transfersystem werden zwei Formen verwendet. Das Rechteck und das Dreieck. Rechtecke begrenzen das Förderband und die Rampe wie in Abbildung 4.1. Eine Kombination aus beiden Formen repräsentiert die Förderbandverengung. Um die Datenhaltung zu vereinfachen, halten Kollisionsboxen die Speicheradressen auf den verwendeten Körper von Box2D.

Förderband

Ohne die Beförderung von Werkstücken könnten keine Änderungen der Sensorik erfolgen. Das Förderband berechnet in jedem Aktualisierungsintervall mithilfe des Ports A seinen zu verwendenen Zustand. Weil in der Welt des Festo-Transfersystems keine Gravitation verwendet wird, muss ein Werkstück beschleunigt werden, um dessen Position für den nächsten Simulationsschritt zu verändern. Die Beschleunigung wird über die Masse, die zu verwendende Geschwindigkeit und die Länge des Aktualisierungsintervalls bestimmt. Falls das Förderband stoppt, müssen die momentanen Beschleunigungen der Werkstücke entfernt werden. Wenn ein Werkstück mit seinem Schwerpunkt, definiert durch die Mitte seines Körpers, über die Förderbandbegrenzungen gerät, wird es vom Förderband und somit aus der Welt von Box2D entfernt. Damit das Werkstück weiterhin auffindbar ist, wird es in eine Datenstruktur mit heruntergefallenen Werkstücken integriert. Diese Datenstruktur ist ein Ringbuffer [33], um

den Speicherbedarf der Simulation zu minimieren. Es ist nicht notwendig alle heruntergefallenen Werkstücke zu archivieren. Aus diesem Grund weist der Ringbuffer eine Kapazität von maximal 20 möglichen Einträgen auf, siehe Abschnitt 3.2.2. Auf diese Weise können neue heruntergefallene Werkstücke die Ältesten überschreiben.

Rampe

Um die zu selektierenden Werkstücke vom Förderband zu entfernen, ohne manuell eingreifen zu müssen, können mithilfe der Weiche Werkstücke auf die Rampe befördert werden. Die Rampe ist eine von drei rechteckigen Kollisionsboxen umschlossene Plattform und weist eine Lichtschranke an der Oberseite der Überführung von Förderband zu Rampe auf. Sie verfügt über eine eigene Datenstruktur für Werkstücke, die sich auf ihr befinden. Eine geteilte Datenverwaltung ist nötig, weil das Förderband sonst Werkstücke befördern würde, die sich auf der Rampe befinden. Die Rampe implementiert das *FestoObject Interface* und wird vom Simulationscontroller aktualisiert. Die Aktualisierungsroutine repositioniert alle Werkstücke und ruft die Aktualisierungroutine der Lichtschranke auf.

Abstandssensor

Jeder Werkstückstyp verfügt im Querschnitt über ein unterschiedliches Profil. Mithilfe der Höheninformationen der Werkstücke, kann die Steuerungssoftware diese klassifizieren. Damit die in Abbildung 2.2 vorgestellten Querschnitte vermessen werden können, verfügt der Abstandssensor über eine Strahlenkomponente von Box2D. Diese berechnet einen Schnittpunkt zwischen Körper und Strahl. Aus diesem Grund muss ein Werkstück eine Repräsentationsform für die Höhenmessung aufweisen. Wird ein Werkstück durch die Höhenmessung befördert, wird zu jedem Simulationsschritt ein neuer Schnittpunkt berechnet. Mithilfe des Schnittpunktes kann, wie in Abbildung 2.5 präsentiert, die Distanz zwischen Strahlursprung und Schnittpunkt berechnet werden. Wird die berechnete Distanz von der Länge zwischen Strahlursprung und Förderband subtrahiert, ergibt sich die tatsächliche Höhe des Werkstückes.

Damit Werkstücke ausgemessen werden können, wird der Bereich der Höhenmessung über zwei Lichtschranken begrenzt. Einlaufende Werkstücke werden über die Lichtschranke vor dem Messpunkt erkannt. Herauslaufende Werkstücke werden anhand der Lichtschranke hinter dem Messpunkt erkannt. Wird ein Werkstück durch die erste Lichtschranke befördert, wird ein zusätzlicher, zum Werkstückstypen passender, Körper der Höhenemssung hinzugefügt. Verlässt dieses Werkstück die Höhenmessung, wird der Körper aus der Welt von Box2D entfernt.

Aktualisierungen werden über den Simulationscontroller gesteuert. Der Abstandssensor ist ein *FestoObject* und verfügt demnach über eine eigene Aktualisierungroutine. Die Aktualisierungsroutine überprüft zuerst, ob ein vorhandenes Werkstück die Höhenmessung verlassen hat. Ist dies der Fall, wird der instanziierte Körper der Seitenansicht gelöscht. Wurde hingegen die vordere Lichtschranke unterbrochen, wird ein neuer Körper hinzugefügt. Nach der Aktualisierung der Werkstücksliste in der Höhenmessung, wird der gemessene Abstand des Abstandssensors berechnet. Dieser wird über den *DataHandler-Adapter* im *DataHandler* festgeschrieben.

6.2. Kommunikationskern

Um mit der HiL-Simulation zu interagieren, werden GPIO-Schnittstelle und Netzwerkdienst für die Ansteuerung verschiedener Funktionen verwendet. Für eine periodische Statusaktualisierung werden zwei Nachrichtensysteme verwendet. Die Hauptfunktionalität des Kommunikationskerns besteht in der Manipulation von Werkstücken, der Ansteuerung der Aktorik, dem Auslesen der Sensoren und dem Versenden periodischer Statusnachrichten. Um Zugriff auf die Daten des DataHandlers zu erhalten, werden die Datahandler-Adapter aus dem Simulationskern verwendet. Für Werkstücksinformationen wird der Simulationscontroller angefragt. Um periodische Nachrichten zu generieren, müssen einige Komponenten die Eigenschaften eines *TimerObservers* aufweisen. Untenstehende Abbildung 6.2 zeigt eine Gesamtübersicht des Kommunikationskerns, die im weiteren Verlauf des Kapitels erläutert wird.

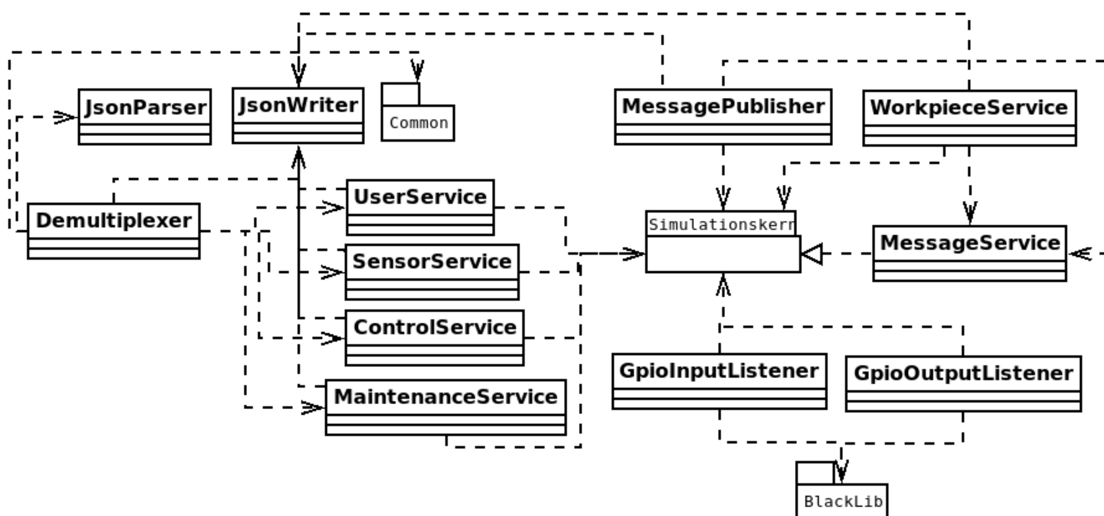

Abbildung 6.2.: Kommunikationskern - Übersicht

6.2.1. Ansteuerung der Simulation über das Netzwerk

Um mit der Simulation in Interaktion zu treten, kann entweder die GPIO-Schnittstelle, beschrieben in Abschnitt 6.2.3, verwendet werden oder es wird der Netzwerkdienst zur Ansteuerung benutzt. Der Netzwerkdienst bietet im Gegenteil zur GPIO-Schnittstelle den Vorteil, dass nicht nur Aktorik und Sensorik angesprochen werden kann, sondern dass zusätzlich die Werkstücke und die Simulation manipuliert werden können.

Für den Netzwerkdienst muss eine logische Teilung in Form von Services gefunden werden. Zur Verfügung stehen vier Funktionseinheiten. Mithile des SensorService können die Sensoren ausgelesen werden. Der ControlService wird für die Ansteuerung der Aktoren verwendet. Der MaintenanceService ermöglicht ein Zurücksetzen und Herunterfahren der HiL-Simulation. Weiterhin kann der Gesamtzustand der Simulation abgefragt werden. Mithilfe des UserService können Interaktionen eines virtuellen Nutzers realisiert werden. Es können Werkstücke hinzugefügt und entfernt, sowie die Orientierung geändert werden.

Die Grundlage für die Netzwerkansteuerung bildet eine Client-Server-Architektur. Diese besteht aus einem Client, der die Dienste des HiL-Servers in Anspruch nimmt. Der Client sendet eine Anforderung an den Server, auch *Request* genannt. Dieser erfüllt den Request und schickt dem Client eine Rückantwort, *Response* genannt. Die Interaktionen zwischen Client und Server finden synchron statt. Dies bedeutet, dass der Client nach einem abgesetzten Request auf den Response des Servers wartet. Um die oben definierten Funktionseinheiten zu addressieren und die HiL-Simulation so wirken zu lassen, als wäre sie lokal verfügbar, werden *Remote Procedure Calls*, kurz *RPCs*, verwendet. RPCs sind entfernte Prozeduraufrufe, die das Verhalten von lokalen Prozeduraufrufen aufweisen [34]. Die Funktionseinheiten weisen eigene Schnittstellenbeschreibungen auf, die die nötigen Parameterlisten und Rückgabewerte der Prozedur beinhalten. Mithilfe der Schnittstellenbeschreibungen können auf Seite des Clients die entfernten Aufrufe in lokale Systemprozeduren umgeformt werden [34]. Hierfür werden Client-Stubs verwendet. Client-Stubs verwenden die Schnittstellen der angebotenen entfernten Prozdeuren, implementieren diese aus und lassen sich danach wie gewöhnliche Objekte verwenden. Die Implementierung der einzelnen angebotenen Prozeduren basiert jedoch auf Netzwerkkommunikation. Hierfür muss anhand eines vereinbarten Übertragungsformates die entfernt aufzurufende Prozedur in eine Nachricht verwandelt werden. Es werden die Parameter der aufzurufenden Prozedur mit weiteren Metadaten für die Adressierung in eine Nachricht umgeformt. Diese Nachricht wird an den Server mit der entfernten Prozedur versendet. Das Versenden der Nachricht und das Empfangen einer Antwort vom Server übernimmt eine

Kommunikationskomponente. Nach dem Absetzen des Requests muss der Client-Stub blockieren und auf ein Response des Servers warten. Empfängt der Server den Request wird dieser von seiner Kommunikationskomponente an den Server-Stub weitergereicht. Dieser entpackt und decodiert die enthaltene Nachricht, auch *Unmarshalling* genannt. Danach wird die in der Nachricht definierte Prozedur mit den übergebenen Parametern aufgerufen. Nach der Ausführung wird der Rückgabewert, falls die Prozedur einen Rückgabewert aufwies, an den Server-Stub weitergeleitet. Dieser verpackt denselben und erstellt eine Nachricht, die über die Kommunikationskomponente des Servers zurück an den Client geschickt wird. Die Kommunikationskomponente des Clients leitet die Nachricht bei Empfang an den Client-Stub weiter, dieser unmarshalled die Nachricht und reicht das Ergebnis der entfernten Prozedur an das Anwendungsprogramm weiter. Die entfernte Prozedur ist abgeschlossen. Um RPCs zu realiseren müssen Client und Server dieselbe Kodierung der Nachrichten nutzen. Diese wird in folgendem Abschnitt vorgestellt.

Nachrichtenaufbau

Um die RPCs korrekt abzubilden, müssen im Request alle nötigen Information für die Ausführung und Adressierung auf der Serverseite vorhanden sein. Prozeduren weisen Signaturen auf. Prozedur-Signaturen bestehen aus dem Prozedurnamen und einer optionalen Parameterliste, die durch die Anzahl, die verwendeten Datentypen und der Reihenfolge der Parameter definiert wird. Zusätzlich weisen Prozeduren Rückgabewerte auf, die wie die Übergabeparamter einen Datentypen und einen Wert besitzen. Um all diese Felder in eine Nachricht zu wandeln, soll, wie in Abschnitt 3.2.2 gefordert, JSON verwendet werden. JSON weist feste Strukturen mit Schlüssel-Werte Paaren auf [13]. Hierdurch wird die Realisierung von verschiedenen Feldern für die Nachrichten leicht lesbar und gut zu parsen. In JSON können geordnete Listen von Werten mithilfe von JSON-Arrays realisiert werden. Arrays können über Indizes angesprochen werden, wodurch die Reihenfolge der Parameterliste einer Prozuder sichergestellt werden kann.

Requests definieren Prozeduraufrufe. Weil JSON über Schlüssel-Werte Beziehungen arbeitet, muss ein Schlüssel für den Nachrichteninhalt des Requests definiert werden. Dieser lautet *Payload*. In dem Nachrichteninhalt muss die anzusprechende Funktionseinheit definiert werden. Diese heißt *Service* und besitzt einen eindeutigen Wert namens *Identifier*. Zusätzlich muss die Signatur der Prozedur aufgeführt werden. Der Schlüssel trägt den Namen *Action*. Falls die Prozedur über Parameter verfügt, werden diese über den Namen und den Wert definiert. Ein gültiger Request hat folgende Form :

$$\{"Payload" : \{"Service" : "Identifier", "Action" : "Procedurename", "Parameter" : \{"Name_1" : "Value_1", \ldots, "Name_n" : "Value_n"\}\}\}$$

Nach jedem gültigen Request muss die HiL-Simulation dem Client ein Response übermitteln. Der Schlüssel für den Nachrichteninhalt lautet *Answer*. Ein Response wird durch den Rückgabewert und dem dazugehörigen Datentypen definiert. Eine gültige Antwort vom Server auf einen Request hat somit folgende Form :

$$\{"answer" : \{"type" : "Datatype", "value" : "value"\}\}$$

Falls es in einer Prozedur zu einer Ausnahmebehandlung kommt oder ein Request fehlerhaft kodiert ist, besteht die Möglichkeit einer Fehlerbenachrichtigung über die Antwort des Servers. Die Antwort definiert für den Rückgabetypen den Nachrichtentypen *ERROR*. Zusätzlich werden der Antwort weitere Felder namens *Category* und *Message* zur Klassifizierung des Fehlers hinzugefügt. Anhand des Typen des Fehlers kann der Client angemessen auf diesen Fehler reagieren. Eine Fehlernachricht hat folgende Form :

$$\{"answer" : \{"type" : "ERROR", "category" : "ERROR_CATEGORY", "message" : "ERROR_MESSAGE"\}\}$$

Eine Liste aller JSON-konformen Nachrichten kann dem Anhang A.1 entnommen werden.

Funktionseinheiten

In der HiL-Simulation wird für die Realiserung des RPC-Mechanismus die Funktionseinheit namens *Demultiplexer* verwendet. Dieser Funktionsblock ist zuständig für die Aufrechterhaltung einer Ende-zu-Ende Verbindung zwischen Client und HiL-Simulation. Der Demultiplexer verarbeitet eingehende Requests und zu versendende Responses. Gemäß der RPC-Mechanik wird im Demultiplexer das Unmarshalling und das Marshalling realisiert. Anhand des Serviceidentifiers in einem Request, ruft der Demultiplexer den passenden Server-Stub mit der dazu angeforderten Prozdeur auf und generiert eine Antwort für den Client. Die Server-Stubs sind der UserService, der SensorService, der ControlService und der MaintenanceService. Diese verwenden einen DataHandler-Adapter oder den Simulationscontroller, um Daten in der HiL-Simulation zu manipulieren oder auszulesen. Damit der Demultiplexer die eingehenden JSON-Nachrichten decodieren kann, verwendet er den JsonParser. Um Rückgabewerte oder Fehler in Nachrichten zu wandeln, wird der JsonWriter verwendet. Beide benutzen die JSON-Bibliothek von Niels Lohmann [35].

Schnittstellenbeschreibungssprache

Um eine plattformübergreifende und programmiersprachenunabhängige Ansteuerung der HiL-Simulation über das Netzwerk zu realiseren, wird eine Schnittstellenbeschreibungssprache, auch *Interface-Definition-Language, kurz IDL*, genannt, verwendet. IDLs sind Metasprachen, die sich für die Beschreibung von Schnittstellen von Software-Komponenten eignen. Es lassen sich die verfügbaren Prozeduren ohne Abhängigkeit von einer Programmiersprache definieren. Mithilfe einer IDL können für verschiedene Programmiersprachen die benötigten Client-Stubs automatisch generiert werden, falls eine Codegenerierung für die IDL existiert. Der Aufbau der IDL wird in JSON realisiert. Für die unterschiedlichen Servicekomponenten wird eine Unterteilung anhand des Schlüssels *Service* vorgenommen. Dieser weist einen *Identifier* auf, der den Namen der Schnittstelle vorgibt. Die Funktionen der Schnittstelle werden in einem JSON-Array definiert, welches über den Schlüssel *description* angesprochen werden kann. Die Signatur von Funktionen weist einen Namen, eine Parameterliste und einen Rückgabewert auf. Folgende Abbildung zeigt den Aufbau der IDL für die HiL-Simulation.

```
{"service" : "SERVICE_IDENTIFIER",
 "description": [{"name" : "PROCEDURE_NAME_#1",
                  "parameter" :[{ "id"     : "POSITION_OF_PARAMETER_#1",
                                  "name" : "NAME_OF_PARAMETER_#1",
                                  "type"   : "DATATYPE_OF_PARAMETER_#1" },
                                          ⋮
                                { "id"     : "POSITION_OF_PARAMETER_#N",
                                  "name" : "NAME_OF_PARAMETER_#N",
                                  "type"   : "DATATYPE_OF_PARAMETER_#N" }
                               ],
                  "return" : "RETURN_TYPE"
                 },
                          ⋮
                 {"name" : "PROCEDURE_NAME_#N",
                  "parameter":[{ "id"      : "POSITION_OF_PARAMETER_#1",
                                 "name" : "NAME_OF_PARAMETER_#1",
                                 "type"   : "DATATYPE_OF_PARAMETER_#1" },
                                          ⋮
                               { "id"      : "POSITION_OF_PARAMETER_#N",
                                 "name" : "NAME_OF_PARAMETER_#N",
                                 "type"   : "DATATYPE_OF_PARAMETER_#N"}
                              ],
                  "return" : "RETURN_TYPE"
                 }
                ]
}
```

Abbildung 6.3.: Aufbau Schnittstellenbeschreibungssprache

Ein selbstimplementierter Codegenerator kann mit der IDL für die Schnittstellen automatisch Client-Stubs, für die unterstützten Programmiersprachen des Codegenerators, generieren. Auf diese Weise kann die Ansteuerung der HiL-Simulation über das Netzwerk nicht nur in C++ realisiert werden.

6.2.2. Nachrichtendienste

Der Kommunikationskern besitzt zwei Nachrichtendienste, die periodisch Informationen an verbundende Clients versenden. Der erste Nachrichtendienst ist der *MessagePublisher*. Dieser veröffentlicht den Gesamtzustand der HiL-Simulation, siehe Anhang A.2. Der zweite Nachrichtendienst ist der *WorkpieceService*, der Auskunft über heruntergefallene Werkstücke gibt, siehe Anhang A.3. Als Kommunikationseinheit wird der *MessageService* verwendet. Dieser wartet auf eingehende Verbindungen, fügt neue Klienten einer Liste von verbundenen Klienten hinzu und pflegt die Liste bei Verbindungsabbrüchen. Damit die Nachrichtendienste periodisch Nachrichten versenden, implementieren sie das TimerObserver-Interface und werden bei der TimerUnit registriert. In den jeweiligen Aktualisierungsroutinen werden die zu versendenden Nachrichten mithilfe des JsonWriters erstellt. Anschließend werden die Nachrichten über den MessageService an alle verbundenen Clients verschickt.

6.2.3. HiL-Schnittstelle

Die HiL-Schnittstelle der Simulation wird über die verfügbaren GPIO-Pins des Beaglebone Blacks realisiert. Grundlage hierfür bildet die Zuordnung aus Abbildung 3.4. Die Aktoren des Festo-Transfersystems werden als Eingänge und die Sensoren als Ausgänge behandelt. Vor jedem Simulationsschritt müssen die als Eingang definierten GPIO-Pins auf Änderungen überprüft werden. Andernfalls könnten geforderte Steuerbefehle zu spät ausgeführt werden oder sie würden gänzlich missachtet werden, falls das Signal nicht lange genug anliegt. Traten Änderungen auf, müssen diese im DataHandler festgeschrieben werden, damit die aktuellen Daten für den nächsten Simulationszyklus zur Verfügung stehen. Diese Aufgabe übernimmt der *GpioInputListener*. Mithilfe des Actuator-Adapters, wird der Datenbestand im DataHandler aktualisiert. Um die als Ausgang definierten GPIO-Pins nach einem Simulationsschritt zu aktualisieren, wird der *GpioOutputListener* verwendet. Seine Aufgabe besteht in der Anpassung der anliegenden Spannung des jeweiligen Pins. Mithilfe des Sensor-Adapters können die Zustände der Sensoren ausgelesen werden. Um die GPIOs des BeagleBone Black zu kontrollieren, wird die quelloffene Bibliothek *BlackLib* [36] verwendet.

7. Implementation

Die Hauptfunktionalität der HiL-Simulation ist ohne Simulationszyklus nicht möglich. Aus diesem Grund wird in dem folgendem Kapitel der Simulationszyklus vorgestellt. Weil der Simulationscontroller abhängig von dem Zeitgeber ist, wird, anhand von Programmcode, erklärt, wie der Zeitgeber realisiert wurde und wie es möglich ist, dass eine Aktualisierungrate von 100 Millisekunden eingehalten werden kann. Weil die Softwareerstellung nicht ohne Probleme ablief, werden abschließend die größten Hindernisse vorgestellt.

7.1. Simulationszyklus

Der Simulationscontroller ist ein TimerObserver und wird periodisch alle 100 Millisekunden von dem Zeitgeber aktualisiert. Hierfür wird die Aktualisierungsmethode des TimerObservers benötigt. Diese trägt den Name *update()* und benötigt keine Parameter und besitzt keinen Rückgabewert. Bevor die Welt von Box2D aktualisiert werden kann, müssen mögliche Änderungen der Aktorik, die in dem DataHandler geschrieben worden sind, übernommen werden, siehe Zeile 36 des Programmcodes in Abbildung 7.1. Je nach Betriebsmodus der Simulation haben entweder die GPIO-Schnittstelle oder der Ansteuerungsdienst Werte verändert. Nach der Aktualisierung der Register müssen die Aktoren den neuen Registerwerten angepasst werden, um einen korrekten Simulationszyklus zu gewährleisten, siehe Zeile 37 und 38. Die Weiche muss ihre Position anpassen und das Förderband muss allen Werkstücken einen neuen Kräftevektor zuweisen, je nachdem wie die neue Ansteuerung des Förderbandes ausfällt. Anschließend muss überprüft werden, ob, bedingt durch die Neupositionierung der Werkstücke aus dem letzten Simulationsschritt, Werkstücke der Rampe hinzuzufügen sind. Andernfalls könnte es vorkommen, dass Werkstücke, die sich auf der Rampe befinden, in eine falsche Richtung befördert werden. Erst jetzt darf Box2D den nächsten Simulationsschritt berechnen, siehe Zeile 41. Hierfür wird die *Step*-Funktion auf der Welt aufgerufen. Sie benötigt drei Parameter, die die Dauer des Simulationszyklus, die Genauigkeit der Positionsberechnung und die Genauigkeit der Kollisionsauflösung definieren.

```
33 void SimulationController::update(){
34     std::lock_guard<std::mutex> lock(wkpcLock);
35
36     dataHandler.update();
37     seperator->update(workpieces);
38     treadmill->update();
39     slide->update(workpieces);
40
41     world->Step(SIMULATION_RENDER_LOOP, VELOCITY_ITERATIONS, POSITION_ITERATIONS);
42
43     heightMeasurement->update(workpieces);
44     metalSensor->update(workpieces);
45
46     for(auto & lb : lightBarrier){
47         lb->update(workpiecesOnSlide);
48         lb->update(workpieces);
49     }
50 }
```

Abbildung 7.1.: Aufrufreihenfolge im Simulationszyklus

Nach dem Simulationsschritt müssen die Sensoren aktualisiert werden. Zuerst wird die neue gemessene Höhe des Abstandssensors berechnet, siehe Zeile 43. Danach wird der Metallsensor aktualisiert. Abschließend müssen alle Lichtschranken ihre Werte neu berechnen und im Data-Handler eintragen. Hierfür werden die *Update()*-Funktionen der jeweiligen Lichtschranken in einer For-Schleife aufgerufen, siehe Zeile 46 bis 49. Im Simulationszyklus werden Netzwerkdienste und GPIO-Schnittstelle nicht betrachtet, weil diese beim Zeitgeber registriert werden. Sie werden nach dem neuen Simulationsschritt abgearbeitet. Die Implementation des Zeitgebers wird in dem nächsten Abschnitt vorgestellt.

7.2. Zeitgeber

Für die Implementation des Zeitgebers wurde auf die *Boost.ASIO*-Bibliothek zurückgegriffen[37]. ASIO steht hierbei für asynchrones Input/Output. Boost.ASIO ist eine plattformunabhängige C++ Bilbiothek, die eine asynchrone Verarbeitung von externen Ressourcen wie Netzwerkverbindungen oder Zeitgebern ermöglicht. Programme, die auf der Bibliothek aufbauen, basieren auf sogenannten Input/Output Services, kurz I/O Services. Mithilfe von I/O Services werden Betriebssystemschnittstellen abstrahiert. Um I/O Services zu verwenden muss ein I/O Serviceobjekt verwendet werden. Dieses Serviceobjekt heißt im folgendem Programmcode *ioTimer* und ist ein *boost::asio::deadline_timer*. Mithilfe des DeadlineTimers kann eine registrierte Funktion nach Ablauf einer Zeitspanne ausgeführt werden. Diese heißt im untenstehenden Programmcode *execute* und wird vom Betriebssystem aufgerufen, sobald die Zeitspanne des Timers abgelaufen ist. Der eigentliche Zeitgeber der Simulation ist ein Thread, dessen Routine

in *run()* definiert wird. Seine Hauptaufgabe besteht in dem Aufrufen aller Aktualisierungsmethoden der registrierten TimerObserver, siehe Zeile 39 und 61 bis 64. Um zu gewährleisten, dass die Aktualisierungsintervalle in einem Zeitrahmen von 100 Millisekunden passieren, muss überprüft werden, ob der Thread mit der Abarbeitung der Aktualisierungsroutinen der TimerObserver fertig ist. Hierfür muss eine Symbolisierung benutzt werden. Die Wahl fiel auf einen Mutex, weil hiermit kritische Bereiche abgebildet werden können und es die Möglichkeit auf Bedingungsvariablen gibt. Bevor der Thread die erste While-Schleife betritt, sperrt er den Mutex. Solange der Boost.ASIO Timer den boolschen Wert *started* nicht verändert hat, siehe Zeile 47, ruft der Thread *timerCondition.wait(lock)* auf, siehe Zeile 37.

```
23 void TimerUnit::execute(boost::system::error_code const& errorCode) {
24     if (errorCode != boost::asio::error::operation_aborted) {
25         wakeUp();
26         if(running){
27             ioTimer->expires_at(ioTimer->expires_at() + boost::posix_time::milliseconds(interval));
28             ioTimer->async_wait(boost::bind(&TimerUnit::execute, this, boost::asio::placeholders::error));
29         }
30     }
31 }
32
33 void TimerUnit::run(){
34     std::unique_lock<std::mutex> lock(timerLock);
35     while (running) {
36         while (!started) {
37             timerCondition.wait(lock);
38         }
39         updateObserver();
40         started = false;
41     }
42 }
43
44 void TimerUnit::wakeUp(){
45     if(timerLock.try_lock()){
46         timerCondition.notify_all();
47         started = true;
48         timerLock.unlock();
49     }else{
50         if(criticalTimer){
51             violationHandler.reportTimingViolation(criticalTimer, timeNotConverted);
52         }
53         started = true;
54     }
55 }
56
57 void TimerUnit::registerObserver(Observer observer){
58     registeredObserver.push_back(observer);
59 }
60
61 void TimerUnit::updateObserver(){
62     for(auto& obs : registeredObserver){
63         obs->update();
64     }
65 }
```

Abbildung 7.2.: Programmcode TimerUnit

Hierdurch wird der Thread suspendiert und der Mutex wird freigegeben. Wenn der Boost.ASIO Timer in der *execute* Methode *wakeUp()* aufruft, wird, falls der Mutex nicht gesperrt ist, das entsprechende *notifyAll()* auf der Bedingungvariable aufgerufen. Hierdurch wird der sus-

pendierte Thread aufgeweckt, wodurch er wieder die TimerObserver abarbeitet. Falls der Mutex noch gesperrt ist, ist dies ein Hinweis darauf, dass der Thread noch am Bearbeiten der Aktualisierungsmethoden der TimerObserver ist. Ist dies der Fall, wird in Zeile 51 dem *TimingViolationHandler* eine Zeitüberschreitung gemeldet.

7.3. Probleme

Im Laufe der Implementation mussten grundlegende Designentscheidungen verworfen werden, weil verwendete Bibliotheken nicht für ARM-Architekturen kompiliert werden konnten. Weiterhin entstanden ungeahnte Verhaltensänderungen der Software durch Versionsunterschiede der verwendeten Boost-Bibliotheken. Folgender Abschnitt stellt die Probleme vor.

7.3.1. RPC-Bibliotheken

In der ersten Iterationsphase der HiL-Simulation sollte für die Kommunikation zwischen Client und Ansteuerungsdienst das Kommunikationsframework *Apache Thrift* [38] verwendet werden. Thrift bietet die Möglichkeit RPCs für verschiedene Programmiersprachen mithilfe einer IDL zu generieren. Ein Codegenerator verwendet die selbstdefinierte thriftkonforme IDL und generiert für Client- und Serverseite die notwendigen Stubs. Die zur Verfügung stehende Version Thrift 0.10 unterstützte über 20 verschiedene Programmiersprachen [38]. Jedoch bestand noch kein Crosscompilersupport, weshalb der Versuch Thrift zu verwenden scheiterte.

Bevor eine eigene RPC-Lösung implementiert wurde, musste der Versuch unternommen werden die Bibliotheken *JSON-RPC 2.0* [39] und *XML-RPC for C and C++* [40] für ARM-Architekturen zu kompilieren. Dies schlug aufgrund von strikteren Compilerüberprüfungen und Fehlern in den Makefiles fehl.

7.3.2. Ansteuerung der GPIO-Pins

Die GPIO-Pins können statisch oder dynamisch konfiguriert werden. Für eine statische Konfiguration werden Devicetree-Overlays verwendet [41]. Dies sind Konfigurationsdateien, die beim Bootvorgang des BeagleBone Black angewendet werden. Die Konfiguration erfolgt im Kernelspace. Um die GPIO-Pins dynamisch im Userspace zu konfigurieren, wird das virtuelle Dateisystem *Sysfs* des Linux-Kernels verwendet, welches eine Schnittstelle zu Kerneldatenstrukturen zur Verfügung stellt [42]. Um GPIO-Pins zu konfigurieren, müssen die entsprechenden Dateien des gewünschten GPIO-Pins verändert werden. Diese liegen unter *sys/class/gpio/* und

bilden eigene Unterordner. Zunächst muss der gewünschte GPIO-Pin exportiert werden, um ihn zu verwenden. In dem entsprechendem Unterordner, zum Beispiel *gpio66*, befinden sich mehrere Dateien. Mit der Datei *direction* kann bestimmt werden, ob der GPIO-Pin als Eingang oder Ausgang konfiguriert werden soll. Über die Datei *value* kann der an dem Pin anliegende Wert ausgelesen oder definiert werden. Um diesen Konfigurationsmechanismus in C++ zu realiseren, werden *Input/OutputFileStreams* verwendet. Hiermit werden die entsprechenden Daten beschrieben oder gelesen. Dadurch wird eine Konfiguration der GPIO-Pins zur Laufzeit der HiL-Simulation ermöglicht.

In der Implementationsphase fiel auf, dass einige GPIO-Pins, wenn sie über die HiL-Simulation angesprochen wurden, nicht funktionierten. Dieses Verhalten zeigte sich auch in der Verwendung von Testskripten. Wurde hingegen die Konfiguration der GPIO-Pins als *SuperUser* vorgenommen, funktionierten die GPIO-Pins. Dieses Problem besteht noch immer, wodurch drei GPIO-Pins der HiL-Schnittstelle nicht beschreibbar sind.

7.3.3. Programmoptionen

Die HiL-Simulation kann in zwei verschiedenen Modi betrieben werden. Klientenmodus und GPIO-Modus. Zusätzlich sollen die verwendeten Ports der Netzwerkdienste verändert werden können. Hierbei dürfen nur die registrierten Ports von 1024 bis 49151 verwendet werden. Die HiL-Simulation verwendet einen synchronen Logger, um Ereignisse und Fehler aufzuzeichnen. Hierfür muss die Möglichkeit gegeben sein, dass die Log-Nachrichten gespeichert werden oder direkt auf der Konsole ausgegeben werden. Um die elektrische Signalisierung zu testen, werden zwei Testroutinen verwendet, siehe Kapitel 8. Um Einfluss auf die Simulation beim Start des Programmes zu haben, werden Boost.ProgramOptions verwendet. Hiermit ist ein Auslesen der Kommandozeile und eine einheitliche Formatierung der möglichen Optionen über eine Hilfsausgabe möglich, siehe Abbildung 7.3.

```
debian@beaglebone:~$ ./FestoSim -h
Options:
  --version                             Display FestoSim version
  -h [ --help ]                         Print help messages
  -i [ --idl ]                          Print Interface Definition Language
                                        with commons
  -v [ --verbose ]                      The initialization is talkative, Logs
                                        are printed on the console and NOT
                                        saved.
  -c [ --client ]                       Run FestoSim in Client Mode. By default
                                        the control will happen via GPIO
  -m [ --portStatusMessage ] arg (=8889)
                                        Specify the port for the status-message
                                        publisher
  -p [ --portSimulationControl ] arg (=8888)
                                        Specify the port for the networkaccess
  -m [ --portFallenWkpcPublisher ] arg (=8900)
                                        Specify the port for the fallen
                                        workpiece publisher
  --gpioButtonTest                      Routine to press/ set high every
                                        registered button/cable
  --gpioLedTest                         Routine to set every registered output
                                        on high

Example :
  FestoSim with no options
     The default parameters will be used.
     - Port 8888 for the networkaccess
     - Port 8889 for the messagePublisher
     - The simulation is controlled over the connected GPIOs
     - Logs will be saved under [PathToFestoSim]/festoLogs/

 Logs :
     The log files can be found in the folder where FestoSim lays/
     The logs will be recreated with every new start of FestoSim.
     That means that you have to save the log files if you want to analyze them.
```

Abbildung 7.3.: Boost ProgramOptions

Für die Kompilierung der HiL-Simulation wird die Boost Version 1.62 verwendet. Bei Verwendung der HiL-Simulation mit Boost Version 1.58 entstehen Segmentierungsfehler, die bei dem Auslesen der Kommandozeile entstehen. Aus diesem Grund muss für ein lauffähiges Kompilat die Boost-Bibliothek Version 1.62 verwendet werden.

8. Validierung der Simulation

Das Testen der HiL-Simulation auf eine korrekte Funktionsweise ist unabdingbar. Das Problem beim Testen besteht in der Vorgehensweise. Für die Testabdeckung der Funktionalität der HiL-Simulation kann ein Black-Box-Test verwendet werden, bei dem nur das nach außen sichtbare Verhalten überprüft wird. Hierfür können die Anforderungen, definiert in Abschnitt 3, herangezogen werden. Für die Programmabläufe der Simulation und Netzwerkdienste muss das innere Verhalten betrachtet werden. Hierfür eignen sich White-Box-Tests bei denen die innere Funktionsweise bekannt ist und diese mit in die Testszenarien einfließt.

Um für Black-Box-Tests einen visuellen und haptischen Eindruck beim Testen zu erzeugen, wurde ein Testboard für die HiL-Simulation entworfen, siehe Abbildung 8.1. Mithilfe des Testboards werden die verwendeten GPIO-Pins an Leuchtdioden und Knöpfe angeschlossen. Hierdurch entfällt das Überprüfen der GPIO-Werte im Filesystem des BeagleBone Black. Um die möglichen Nachrichten der RPC-Schnittstellen zu testen, kann das Netzwerk-Analyse Werkzeug *Wireshark* zusammen mit dem Netzwerkwerkzeug *Netcat* benutzt werden. Für die Einsicht auf die internen Abläufe der Software können die generierten Lognachrichten der HiL-Simulation und der Debugger verwendet werden.

Abbildung 8.1.: Testboard für die Überprüfung von Sensorik und Aktorik

Für die Validierung der Funktionsweise der HiL-Simulation, wurden Testszenarien mit Erwartungshaltungen entwickelt. Weil die HiL-Simulation einen hohen Funktionsumfang aufweist, wurde für ein vereinfachtes Testen ein Programm mit grafischer Benutzeroberfläche implementiert, siehe Abbildung 8.2. Um alle Steuerbefehle zu testen, wurde das Versenden der Befehle, siehe hierzu Anhang A.1, auf bestimmte Tasten der Tastatur gelegt. Die grafische Benutzeroberfläche wird mithilfe der Statusnachrichten der HiL-Simulation aktualisiert.

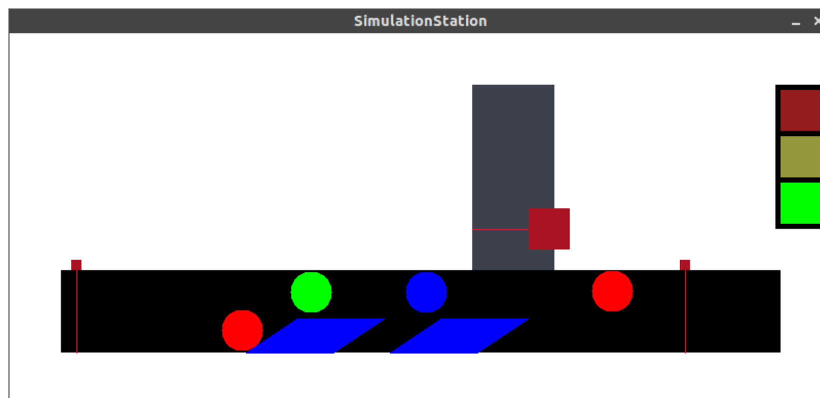

Abbildung 8.2.: Programm für die Überprüfung der HiL-Simulation

Für die Überprüfung der korrekten Ansteuerung der GPIO-Pins, wurden zwei Testroutinen geschrieben. Die GPIO-Pins teilen sich in Eingänge und Ausgänge. Die Testroutine für die Eingänge iteriert über eine Liste von registrierten GPIO-Pins und wartet auf eine steigende Flanke an dem momentan zu testenden Pin. Die Testroutine für die Sensoren verhält sich gegensätzlich. Es werden die registrierten GPIO-Pins der Ausgänge beim Drücken der Eingabetaste einzeln auf High und Low gesetzt. Falls es einen Fehler in der Verkabelung zweier Boards gab, kann dies mithilfe der Testroutinen herausgefunden werden. Mithilfe des Testboards und der Testroutinen wurde das in Abschnitt 7.3.2 erwähnte Problem aufgedeckt.

9. Fazit

Das Ziel dieser Bachelorarbeit bestand in der Entwicklung eines Prototypen einer HiL-Simulation eines Festo-Transfersystems. Der entwickelte Prototyp weist einen größeren Funktionsumfang, als die vorher genutzte Simulation auf und bietet eine genauere Kollisionsbehandlung. Durch Probleme der Ansteuerung der GPIO-Schnittstelle können drei GPIO-Pins nicht verwendet werden. Aus diesem Grund kann die HiL-Simulation momentan nur beschränkt verwendet werden.

Für die HiL-Simulation wird eine diskrete zeitorientierte Simulationsform verwendet. Durch eine Analyse des Referenzsystems konnte ein detailliertes Simulationsmodell erstellt werden. Anhand der definierten funktionalen und nicht funktionalen Anforderungen wurde der Funktionsumfang und die Funktionsweise festgelegt. Durch eine Abstraktion einer dreidimensionalen Welt in zwei Dimensionen, können Kollisionsüberprüfungsroutinen simpler realisiert werden. Mithilfe der verwendeten Physik-Engine wird eine hochauflösende Kollisionsbehandlung garantiert, die durch optimierte Kollisionsberechnungen effizient arbeitet. Eine Abstraktion eines virtuellen Benutzers des Festo-Transfersystems wurde über einen Netzwerkdienst realisiert. Durch die Verwendung von entfernten Prozeduraufrufen gestaltet sich die Benutzung des Netzwerkdienstes wie die Verwendung einer lokal verfügbaren Prozedur. Über eine Schnittstellenbeschreibungssprache und JSON als Nachrichtenformat werden die vom Netzwerkdienst angebotenen Prozeduren sprachenunabhängig verfügbar. Mithilfe eines Testboards und einer grafischen Darstellung wurde die Funktionsweise des Prototypen überprüft. Die Software läuft stabil auf dem BeagleBone Black.

Um den Prototypen zu erweitern, bietet sich eine statische Pinbelegung des BeagleBone Blacks mit DeviceTreeOverlays an. Auf diese Weise werden die benötigten GPIO-Pins beim Bootvorgang konfiguriert, wodurch eine Fehlerquelle der GPIO Nutzung vermieden werden kann. Zusätzlich sollten die digitalen Höhenwerte analog zur Verfügung stehen. Um eine realistischere Simulation zu erzeugen, sollten Reibungskoeffizienten und Materialeigenschaften in den Kollisionsberechnungen berücksichtigt werden.

Der Ausblick für weitere Arbeiten liegt in der Erweiterung der HiL-Simulation um oben genannte Eigenschaften und die Realisierung einer Möglichkeit mehrere HiL-Simulationen miteinander zu verbinden, sodass aneinandergereihte Festo-Transfersysteme simuliert werden können. Eine weitere Arbeit könnte in der Virtualisierung des Prototypen bestehen. Wenn die GPIO-Schnittstelle über Software abstrahiert werden kann, könnte die Simulation in einer virtuellen Umgebung betrieben werden, wodurch der BeagleBone Black überflüssig wird. Mithilfe einer Erweiterung der RPC-Schnittstellen könnte die HiL-Simulation als Webservice zur Verfügung gestellt werden, wodurch Studenten online ihre erstellte Ansteuerungssoftware validieren könnten.

A. Anhang

A.1. JSON-Nachricht aller Steuernachrichten - RPC Zugriff


```
{"Payload":{"Action":"WP_ADD_AT","Parameter":{"WorkpieceType":"WORKPIECE_FLAT","x":20.0,"y":20.0,"z":0.0}},"Service":"USER_SERVICE","Timestamp":"04-06-2018 02:07:56"}
{"Payload":{"Action":"WP_ADD_AT","Parameter":{"WorkpieceType":"WORKPIECE_METAL","x":20.0,"y":50.0,"z":0.0}},"Service":"USER_SERVICE","Timestamp":"04-06-2018 02:07:58"}
{"Payload":{"Action":"WP_ADD_AT","Parameter":{"WorkpieceType":"WORKPIECE_NORMAL","x":20.0,"y":35.0,"z":0.0}},"Service":"USER_SERVICE","Timestamp":"04-06-2018 02:07:59"}
{"Payload":{"Action":"WP_ADD_AT","Parameter":{"WorkpieceType":"WORKPIECE_CODED_1","x":20.0,"y":35.0,"z":0.0}},"Service":"USER_SERVICE","Timestamp":"04-06-2018 02:08:00"}
{"Payload":{"Action":"WP_ADD_AT","Parameter":{"WorkpieceType":"WORKPIECE_CODED_2","x":20.0,"y":20.0,"z":0.0}},"Service":"USER_SERVICE","Timestamp":"04-06-2018 02:08:01"}
{"Payload":{"Action":"WP_FLIP","Parameter":{"int":0}},"Service":"USER_SERVICE","Timestamp":"04-06-2018 02:08:01"}
{"Payload":{"Action":"WP_FLIP","Parameter":{"int":0}},"Service":"USER_SERVICE","Timestamp":"04-06-2018 02:08:02"}
{"Payload":{"Action":"WP_FLIP","Parameter":{"int":0}},"Service":"USER_SERVICE","Timestamp":"04-06-2018 02:08:03"}
{"Payload":{"Action":"WP_REMOVE_AT","Parameter":{"int":1}},"Service":"USER_SERVICE","Timestamp":"04-06-2018 02:08:04"}
{"Payload":{"Action":"WP_REMOVE_AT","Parameter":{"int":2}},"Service":"USER_SERVICE","Timestamp":"04-06-2018 02:08:04"}
{"Payload":{"Action":"WP_REMOVE_AT","Parameter":{"int":3}},"Service":"USER_SERVICE","Timestamp":"04-06-2018 02:08:05"}
{"Payload":{"Action":"LB_BEGINNING"},"Service":"SENSOR_SERVICE","Timestamp":"04-06-2018 02:08:10"}
{"Payload":{"Action":"LB_HEIGHT_UNIT"},"Service":"SENSOR_SERVICE","Timestamp":"04-06-2018 02:08:13"}
{"Payload":{"Action":"LB_BEGINNING"},"Service":"SENSOR_SERVICE","Timestamp":"04-06-2018 02:08:14"}
{"Payload":{"Action":"VALID_HEIGHT"},"Service":"SENSOR_SERVICE","Timestamp":"04-06-2018 02:08:16"}
{"Payload":{"Action":"LB_SWITCH"},"Service":"SENSOR_SERVICE","Timestamp":"04-06-2018 02:08:17"}
{"Payload":{"Action":"METAL_SENSOR"},"Service":"SENSOR_SERVICE","Timestamp":"04-06-2018 02:08:17"}
{"Payload":{"Action":"SWITCH_OPENED"},"Service":"SENSOR_SERVICE","Timestamp":"04-06-2018 02:08:18"}
{"Payload":{"Action":"LB_RAMP"},"Service":"SENSOR_SERVICE","Timestamp":"04-06-2018 02:08:19"}
{"Payload":{"Action":"LB_END"},"Service":"SENSOR_SERVICE","Timestamp":"04-06-2018 02:08:19"}
{"Payload":{"Action":"LED_GREEN_OFF"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:21"}
{"Payload":{"Action":"LED_GREEN_ON"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:23"}
{"Payload":{"Action":"LED_GREEN_OFF"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:25"}
{"Payload":{"Action":"LED_YELLOW_OFF"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:25"}
{"Payload":{"Action":"LED_YELLOW_ON"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:27"}
{"Payload":{"Action":"LED_RED_OFF"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:27"}
{"Payload":{"Action":"LED_RED_ON"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:28"}
{"Payload":{"Action":"LED_Q1_OFF"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:28"}
{"Payload":{"Action":"LED_Q1_ON"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:29"}
{"Payload":{"Action":"LED_Q2_OFF"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:30"}
{"Payload":{"Action":"LED_Q2_ON"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:30"}
{"Payload":{"Action":"LED_RESET_ON"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:31"}
{"Payload":{"Action":"LED_RESET_OFF"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:31"}
{"Payload":{"Action":"LED_START_ON"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:32"}
{"Payload":{"Action":"LED_START_OFF"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:32"}
{"Payload":{"Action":"CURRENT_TIMESLICE"},"Service":"MAINTENANCE_SERVICE","Timestamp":"04-06-2018 02:08:33"}
{"Payload":{"Action":"CURRENT_TIMESLICE"},"Service":"MAINTENANCE_SERVICE","Timestamp":"04-06-2018 02:08:33"}
{"Payload":{"Action":"STATUS_SIMULATION"},"Service":"MAINTENANCE_SERVICE","Timestamp":"04-06-2018 02:08:35"}
{"Payload":{"Action":"BTN_ESTOP_PRESS"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:37"}
{"Payload":{"Action":"BTN_ESTOP_RELEASE"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:39"}
{"Payload":{"Action":"BTN_STOP_RELEASE"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:40"}
{"Payload":{"Action":"BTN_STOP_PRESS"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:41"}
{"Payload":{"Action":"BTN_START_RELEASE"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:42"}
{"Payload":{"Action":"BTN_START_PRESS"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:43"}
{"Payload":{"Action":"BTN_RESET_RELEASE"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:43"}
{"Payload":{"Action":"BTN_RESET_PRESS"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:44"}
{"Payload":{"Action":"VALID_HEIGHT"},"Service":"SENSOR_SERVICE","Timestamp":"04-06-2018 02:08:44"}
{"Payload":{"Action":"VALID_HEIGHT"},"Service":"SENSOR_SERVICE","Timestamp":"04-06-2018 02:08:45"}
{"Payload":{"Action":"RESET_SIMULATION"},"Service":"MAINTENANCE_SERVICE","Timestamp":"04-06-2018 02:08:46"}
{"Payload":{"Action":"SHUTDOWN_SIMULATION"},"Service":"MAINTENANCE_SERVICE","Timestamp":"04-06-2018 02:08:47"}
{"Payload":{"Action":"ENGINE_RIGHT_ON"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:54"}
{"Payload":{"Action":"ENGINE_RIGHT_OFF"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:52"}
{"Payload":{"Action":"ENGINE_LEFT_ON"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:53"}
{"Payload":{"Action":"ENGINE_LEFT_OFF"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:54"}
{"Payload":{"Action":"SWITCH_CLOSE"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:56"}
{"Payload":{"Action":"SWITCH_OPEN"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:57"}
{"Payload":{"Action":"ENGINE_SLOW_ON"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:57"}
{"Payload":{"Action":"ENGINE_SLOW_OFF"},"Service":"CONTROL_SERVICE","Timestamp":"04-06-2018 02:08:58"}
["answer",{"type":"integer","value":0}]
["answer",{"type":"boolean","value":true}]
["answer",{"type":"boolean","value":false}]
["answer",{"category":"GPIO","message":"GPIO mode is used.","type":"error"}]
["answer",{"category":"Not implemented","message":"SHUTDOWN_SIMULATION not supported.","type":"error"}]
```


Abbildung A.1.: RPC Nachrichten

A.2. JSON-Nachricht des Gesamtzustandes


```
actors:
  SWITCH_STATUS:              false
  button:
    BTN_ESTOP_STATUS:         false
    BTN_RESET_STATUS:         false
    BTN_START_STATUS:         false
    BTN_STOP_STATUS:          false
  engine:
    ENGINE_LEFT_STATUS:       false
    ENGINE_RIGHT_STATUS:      true
    ENGINE_SLOW_STATUS:       true
    ENGINE_STOP_STATUS:       true
misc:
  SIMULATION_REFRESH_RATE:    100
  TIMESTAMP:                  "17-01-2018 11:56:43"
  TIMING_VIOLATION:           false
ports:
  A_BITMASK:                  237
  B_BITMASK:                  233
  C_BITMASK:                  160
sensors:
  HEIGHT:                     12.3163118362427
  METAL_SENSOR:               false
  led:
    LED_Q1_STATUS:            false
    LED_Q2_STATUS:            false
    LED_RESET_STATUS:         false
    LED_START_STATUS:         false
  lightBarrier:
    LB_BEGINNING:             false
    LB_END:                   false
    LB_HEIGHT_UNIT:           true
    LB_RAMP:                  false
    LB_SWITCH:                false
  trafficLight:
    LED_GREEN_STATUS:         true
    LED_RED_STATUS:           true
    LED_YELLOW_STATUS:        true
workpieceSize:                5
workpieces:
  0:
    fellAtEnd:                false
    id:                       4
    orientation:              0
    position:
      x:                      130.70964050293
      y:                      20.0499496459961
    type:                     "WORKPIECE_CODED_2"
  1:
    fellAtEnd:                false
    id:                       3
    orientation:              0
    position:
      x:                      181.806365966797
      y:                      35
    type:                     "WORKPIECE_CODED_1"
  2:
    fellAtEnd:                false
    id:                       2
    orientation:              0
    position:
      x:                      229.407821655273
      y:                      52.1094741821289
    type:                     "WORKPIECE_NORMAL"
  3:
    fellAtEnd:                false
    id:                       1
    orientation:              0
    position:
      x:                      286.57373046875
      y:                      58.1720199584961
    type:                     "WORKPIECE_METAL"
  4:
    fellAtEnd:                false
    id:                       0
    orientation:              0
    position:
      x:                      400.300109863281
      y:                      62.6721343994141
    type:                     "WORKPIECE_FLAT"
```


Abbildung A.2.: JSON - Gesamtzustand

A.3. JSON-Nachricht aller heruntergefallenenen Werkstücke


```
fallenWorkpieces:
  0:
    fellAtEnd:      false
    fellAtFront:    true
    id:             0
    orientation:    0
    type:           "WORKPIECE_METAL"
  1:
    fellAtEnd:      true
    fellAtFront:    false
    id:             1
    orientation:    0
    type:           "WORKPIECE_METAL"
  2:
    fellAtEnd:      false
    fellAtFront:    true
    id:             3
    orientation:    0
    type:           "WORKPIECE_METAL"
  3:
    fellAtEnd:      true
    fellAtFront:    false
    id:             2
    orientation:    0
    type:           "WORKPIECE_FLAT"
  4:
    fellAtEnd:      false
    fellAtFront:    true
    id:             4
    orientation:    0
    type:           "WORKPIECE_FLAT"
size:               5
```


Abbildung A.3.: JSON-Nachricht aller heruntergefallenenen Werkstücke

Literaturverzeichnis

[1] MathWorks. URL: https://de.mathworks.com/discovery/hardware-in-the-loop-hil.html (aufgerufen am 25.04.2018).

[2] FESTO, . URL: http://www.festo-didactic.com/de-de/lernsysteme/elektrotechnik-elektronik-sps/mps-transfersystem/mps-transfersystem-bewegt-die-mechatronik-und-elektrotechnikausbildung.htm?fbid=ZGUuZGUuNTQ0LjEzLjE4LjExNjkuNjgyNw (aufgerufen am 24.04.2018).

[3] di-soric Group. URL: https://www.di-soric.com/de/Einweglichtschranken-di-soric-30668.html (aufgerufen am 24.04.2018).

[4] FESTO, . URL: https://www.festo.com/net/SupportPortal/Files/405119/SOEL-RTD-Q50-PP-S-7L_2013-01c_8024628F3.pdf (aufgerufen am 24.04.2018).

[5] BALLUFF GmbH. URL: https://wiki.induux.de/Induktive_Sensoren (aufgerufen am 24.04.2018).

[6] FESTO, . URL: http://www.festo-didactic.com/de-de/lernsysteme/elektrotechnik-elektronik-sps/mps-transfersystem/die-transferstrecke-kombinieren-und-erweitern.htm?fbid=ZGUuZGUuNTQ0LjEzLjE4LjExNjkuNjgzMA (aufgerufen am 24.04.2018).

[7] reichelt elektronik GmbH & Co. KG. URL: https://www.reichelt.de/Einplatinen-Computer/BEAGLEBONE-BLACK/3/index.html?ACTION=3&GROUPID=8242&ARTICLE=143499 (aufgerufen am 30.04.2018).

[8] Raspberry Pi Foundation, . URL: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ (aufgerufen am 24.05.2018).

[9] Prof. Dr. Robert Gillenkirch. URL: https://wirtschaftslexikon.gabler.de/definition/system-50117/version-273342 (aufgerufen am 24.05.2018).

[10] Prof. Dr. Jean-Paul Thommen. URL: https://wirtschaftslexikon.gabler.de/definition/modell-39245/version-262658 (aufgerufen am 24.05.2018).

[11] RiskNet. URL: https://www.risknet.de/wissen/rm-methoden/monte-carlo-simulation/ (aufgerufen am 24.05.2018).

[12] Ulrich Hedtstueck. Springer, 2013. ISBN 978-3-642-34870-9.

[13] D. Crockford. The application/json media type for javascript object notation (json). RFC 4627, RFC Editor, July 2006. URL http://www.rfc-editor.org/rfc/rfc4627.txt. http://www.rfc-editor.org/rfc/rfc4627.txt.

[14] W. Eddy. Tcp syn flooding attacks and common mitigations. RFC 4987, RFC Editor, August 2007.

[15] Jon Postel. Transmission control protocol. STD 7, RFC Editor, September 1981. URL http://www.rfc-editor.org/rfc/rfc793.txt. http://www.rfc-editor.org/rfc/rfc793.txt.

[16] C. Hornig. A standard for the transmission of ip datagrams over ethernet networks. STD 41, RFC Editor, April 1984.

[17] J. Postel. User datagram protocol. STD 6, RFC Editor, August 1980. URL http://www.rfc-editor.org/rfc/rfc768.txt. http://www.rfc-editor.org/rfc/rfc768.txt.

[18] Nvidia PhysX. URL: https://developer.nvidia.com/physx-source-github (aufgerufen am 05.06.2018).

[19] Bullet 3 Github. URL: https://github.com/bulletphysics/bullet3 (aufgerufen am 05.06.2018).

[20] Erin Catto. URL: hthttps://github.com/erincatto/Box2D (aufgerufen am 25.05.2018).

[21] Open Source Initiative. URL: https://opensource.org/licenses/Zlib (aufgerufen am 09.05.2018).

[22] Erin Catto. Box2d v2.3.0 user manual. 2007.

[23] Emanuele Feronato. URL: http://www.emanueleferonato.com/2013/11/12/unity3d-goes-also-2d-with-unity-4-3-release/ (aufgerufen am 09.05.2018).

[24] Thomas Larsson Linus Kaellberg. Fast computation of tight fitting oriented bounding boxes.

[25] Kai Shiu Chong. URL: https://gamedevelopment.tutsplus.com/tutorials/collision-detection-using-the-separating-axis-theorem--gamedev-169 (aufgerufen am 12.05.2018).

[26] Prof. Dr. Hans Werner Lang. URL: http://www.iti.fh-flensburg.de/lang/algorithmen/geo/convex.htm (aufgerufen am 12.05.2018).

[27] Serlo.org. URL: https://de.serlo.org/mathe/funktionen/kurvendiskussion/nullstellen/mitternachtsformel (aufgerufen am 26.05.2018).

[28] Eric Rowell et al. URL: http://bigocheatsheet.com/ (aufgerufen am 28.05.2018).

[29] Randy Gaul. URL: http://www.randygaul.net/2013/08/06/dynamic-aabb-tree/ (aufgerufen am 28.05.2018).

[30] Digitalrune. URL: http://digitalrune.github.io/DigitalRune-Documentation/html/cd2fc090-d3fd-4a0f-a7d3-1759241c8545.htm (aufgerufen am 29.05.2018).

[31] Prof. Dr. rer. nat. Thomas Lehmann. Software engineering 2 - embedded software modellierung /entwurfsmethodik hal. 2015.

[32] SourceMaking.com. URL: https://sourcemaking.com/design_patterns/adapter (aufgerufen am 30.05.2018).

[33] Phillip Johnston. URL: https://embeddedartistry.com/blog/2017/4/6/circular-buffers-in-cc (aufgerufen am 31.05.2018).

[34] Guenther Bengel. Springer Verlag, 2014. ISBN 978-3-8348-1670-2.

[35] Niels Lohmann. URL: https://github.com/nlohmann/json (aufgerufen am 04.06.2018).

[36] Yiğit Yüce. URL: http://blacklib.yigityuce.com/ (aufgerufen am 04.06.2018).

[37] David Abrahams Beman Dawes. URL: https://www.boost.org/ (aufgerufen am 07.06.2018).

[38] Apache Software Foundation, . URL: https://thrift.apache.org/ (aufgerufen am 07.06.2018).

[39] Matt Morley. URL: http://www.jsonrpc.org/specification (aufgerufen am 07.06.2018).

[40] Eric Kidd. URL: http://xmlrpc-c.sourceforge.net/ (aufgerufen am 07.06.2018).

[41] D. Molloy. *Exploring BeagleBone: Tools and Techniques for Building with Embedded Linux*. Wiley, 2014. ISBN 9781118935125.

[42] Linux Programmer's Manual, sysfs - a filesystem for exporting kernel objects.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 12. Juni 2018 Thomas Drauschke