UNIVERSITY OF BREMEN

BACHELOR THESIS

PyCRAM - Accurate Physics-based Environment for Executing Mobile Pick and Place Plans

---

Author:
Jonas DECH

Supervisor:
Prof. Michael BEETZ
Second Supervisor:
Dr. Daniel GROSSE
Advisor:
Gayane KAZHOYAN

A thesis submitted in fulfillment of the requirements
for the degree of Bachelor of Science

in the

Institute for Artificial Intelligence
Computer Science

December 6, 2019

Declaration of Authorship

I, Jonas DECH, declare that this thesis titled, "PyCRAM - Accurate Physics-based Environment for Executing Mobile Pick and Place Plans" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

UNIVERSITY OF BREMEN

Abstract

Faculty 3
Computer Science

Bachelor of Science

PyCRAM - Accurate Physics-based Environment for Executing Mobile Pick and Place Plans

by Jonas DECH

With the progress in robotic research more complex environments are becoming possible for a robot platform to manage. This creates new challenges for a robot to solve, while working in these environments. This includes both reasoning mechanisms about physical behavior of objects in this environments and the simulation of actions before them being performed in real life. Thus, a lightweight physics-based simulation is warranted. This thesis presents the BulletWorld, a lightweight physics-based simulation, along with its reasoning capabilities to assist with the implementation of a robot in complex environments.

The BulletWorld consists of three modules: the BulletWorld, GUI and Objects. The BulletWorld represents an empty simulation as well as the means to interact with it and to get objects which are loaded inside the simulation. The GUI visualizes the simulation in a window, thus allowing the user to see the situation inside the simulation . The Objects which are loaded inside the simulation are represented by a single class, this allows the user to interact with every object via the same API.
The reasoning capabilities provide the user with the ability to query the simulation in a semantic way, i.e., are two objects in contact with each other or is an object visible from a specific pose.
To integrate this simulation with the existing PyCRAM framework the designator and process modules are used and implemented for the PR2.
This allows the user to implement robot control programs and execute them inside the simulation. This could be, for example, a simple pick and place plan.

Acknowledgements

I want to thank my advisor Gayane Kazhoyan for helping me understand the BulletWorld of CRAM and sharing her ideas with me. As well as all the feedback she gave me.
I also want to thank my dear friend Christian Preißner for spellchecking and correcting this work and making sure everything is readable.
Also thanks to my parents Thomas and Doris Dech for all their motivation and support during the course of my studies. Thanks to my girlfriend Martha Schnieber for all her lovely motivation.
Thanks to my friends Jan Kleinekathöfer, Thomas Lipps, and Lennard Mai for always motivating me to give my best efforts.

Contents

Declaration of Authorship iii

Abstract v

Acknowledgements vii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Approach . . . 2
- 1.3 Contribution . . . 3
- 1.4 Readers Guide . . . 3

2 Related Work 5
- 2.1 Gazebo . . . 5
- 2.2 Actin . . . 5
- 2.3 Webots . . . 6

3 Foundation 7
- 3.1 Robotics Foundation . . . 7
- 3.2 ROS . . . 8
- 3.3 URDF . . . 8
  - 3.3.1 The Structure of an URDF . . . 9
- 3.4 Bullet Physics Engine . . . 9
  - 3.4.1 PyBullet . . . 11
- 3.5 CRAM High-level Robot Control Framework . . . 12
  - 3.5.1 Designators . . . 13
  - 3.5.2 Process Modules . . . 13
  - 3.5.3 CRAM Plan Language . . . 14
  - 3.5.4 Bullet World . . . 14
- 3.6 PyCRAM . . . 15

4 Implementation 17
- 4.1 Architecture . . . 17
- 4.2 Bullet World . . . 18
  - 4.2.1 Current Bullet World . . . 20
  - 4.2.2 GUI . . . 20
  - 4.2.3 Events . . . 21
- 4.3 Objects . . . 22
  - 4.3.1 Attachments . . . 23
  - 4.3.2 Detachments . . . 26
- 4.4 Bullet World Reasoning . . . 26
  - 4.4.1 Stable . . . 26
  - 4.4.2 Contact . . . 27

4.4.3 Visible 27
4.4.4 Occluding 28
4.4.5 Reachable 28
4.4.6 Blocking 29
4.4.7 Supporting 29
4.5 Designators 30
4.5.1 Moving 30
4.5.2 Pick-Up 31
4.5.3 Accessing 31
4.5.4 Looking 31
4.5.5 Opening-Gripper 31
4.5.6 Closing-Gripper 32
4.5.7 Detecting 32
4.5.8 Move-TCP 33
4.5.9 Move-Arm-Joints 33
4.5.10 World-State-Detecting 34
4.6 Referencing 34
4.7 Process Modules 34
4.8 Choosing a Process Module 35
4.9 Control Flow 36

5 Evaluation **39**
5.1 Demo 39

6 Conclusion **45**
6.1 Summary 45
6.2 Discussion 45
6.3 Future Work 46

Bibliography **49**

List of Figures

4.1 The structure of the BulletWorld, GUI, BulletWorldReasoning and Object classes along with the Motion Designators and Process Modules for the PR2. 18
4.2 An empty simulation 21
4.3 PR2 in the kitchen with a few objects 23
4.4 The PR2 with an attached object 25
4.5 A visual representation of the control flow 37

5.1 An UML activity diagram of the demo 39
5.2 The simulation after initializing and loading all objects 40
5.3 The PR2 opening a drawer 41
5.4 The robot after trying to detect the object 42
5.5 The robot after picking up the given object 42
5.6 The robot after placing the object on the table 43

List of Tables

3.1 The types of joints URDF supports 10
3.2 The tools provided by the PyBullet library 12
3.3 The designators available in CRAM 13
3.4 Selection of the most important expressions of CPL 14

4.1 Table of methods the BulletWorld class provides 19
4.2 The methods of the class `bullet_world_reasoning` 24
4.3 A list of all reasoning queries 26
4.4 All available designator 30
4.5 All slots of the moving designator 31
4.6 All slots of the pick-up designator 31
4.7 All slots of the accessing designator 32
4.8 All slots of the looking designator 32
4.9 All slots of the opening-gripper designator 32
4.10 All slots of the closing-gripper designator 33
4.11 All slots of the detecting designator 33
4.12 All slots of the move-TCP designator 33
4.13 All slots of the move-arm-joints designator 33
4.14 All slots of the world-state-detecting designator 34
4.15 All process modules 35

Chapter 1

Introduction

1.1 Motivation

For a few decades robots are capable of performing simple highly structured actions, like working in car production facility. Due to their precise and unique abilities they are frequently employed in laboratories or car production facilities. Such environments are usually static, with no human movement needing to be to be accounted for. Robots are capable to continuously perform the same task they were programmed to do. This is made possible due to target objects or task always being found in the same position.

With the progress in robotics, more complex tasks and environments are becoming possible for a robot to manage. With the growing number of older adults due to demographic change [Statistisches Bundesamt, 2019], the need for robots to work in highly dynamic environments increases. One such environment being a human household. But such environments pose many new challenges due to their dynamic nature. Robots thus need to be able to perceive and navigate in this dynamic environments, while keeping track of their own position as well as proximate objects.

They also need to simulate possible future actions and their outcome. One way to simulate this is by means of physics-based simulation. They offer the advantage of simulating many different scenarios with different positions in a short period of time. This enables the programmer to pick the best scenario and apply it in real-world settings.

One of the important challenges when working in such dynamic environments is to reason about the current and future behavior of objects. However, this can also be solved by the means of physics-based simulation in addition with reasoning capabilities which enable a user to query the simulation in a semantic way.

Robot simulators, such as Gazebo or Webots aim to solve the aforementioned challenges. However, these simulators have the disadvantage of simulating very elaborate scenarios with detailed movements, thus taking longer to simulate. Further, these simulators offer a wide range of tools and functions and are, as a result, not lightweight.

To solve this problem the Cognitive Robot Abstract Machine (CRAM) framework [Mösnelechner, 2016] implements the so-called BulletWorld. This simulation environment abstracts from continuous but irrelevant movements, by teleporting the robot between positions and just simulating critical movements like placing an object. In addition CRAM also offers reasoning capabilities to determine the situation

inside the simulation and react to it.

However, as CRAM is written in Common LISP, a language that is nowadays rarely used, it is not frequently used in programming robots.

As a result the goal of this thesis is to implement a fast lightweight physics-based simulation for a robot in Python, using the PyCRAM plan language and the concept of designators which was previously implemented in PyCRAM [Augsten and Augsten, 2019].

1.2 Approach

To implement this simulation environment along with the reasoning capabilities, PyBullet [Coumans and Bai, 2016–2019] was used. This Python module provides the user with a basic physics simulation. To use this simulation as a full robot simulation, including physics-based reasoning capabilities, a wrapper class needed to be implemented. This wrapper class represents an empty simulation and provides methods to interact with it.

To fill it with objects a second class needed to be implemented, representing objects inside the simulation. The advantage of using one class to represent eveything inside the simulation is that users can interact with everything via the same API whether it is a robot or just a spoon.

On top of this wrapper classes, the reasoning mechanics had to be implemented. That is because the reasoning works with and within the simulation, for this to work properly the simulation has to work in the first place.

The reasoning allows the user to query the simulation in a semantic way, for example: one can query if two objects are in contact with each other or if one object is visible from a specific point like the camera of a robot. The reasoning capabilities provide a wide variety of features which are explained in Section 4.4.

However, every reasoning query works by the same principle: alter the simulation, then check the behavior of all objects. Based on this, evaluate a return value, before the returning the value, altering of the simulation is reversed.

Some commonly used motions a robot may perform in a household context have been implemented for a PR2 robot, using the physics simulation described above. This way an user can easily implement multi-step plans for the PR2 robot, without having to trouble about low-level robot-specific implementation.

For this purpose PyCRAMs [Augsten and Augsten, 2019] motion designator and process modules were used. The latter contain the code to interact with the simulation, whereas the motion designator is a symbolic representation of a movement the robot should perform.
With this, a simple pick and place plan was implemented.

1.3 Contribution

This thesis presents an easy to use physics-based simulation environment for testing robot control programs and simulating different scenarios for the Robot.

The reasoning capabilities allow the user to query the simulation in a semantic way, which allow the control program to react dynamically to changes in the environment.

To use the simulation with an actual robot Willow Garage's Personal Robot 2 (PR2) was integrated with the simulation and a simple pick and place plan was created.

The simulation presented in the following was implemented in Python, a widely used language in robotics.

The work is available at https://github.com/Tigul/pycram

1.4 Readers Guide

This thesis explains how the simulation was integrated with the PyCRAM framework, how each part of it operates and which methods are available to the user.

Chapter 2 This chapter gives an overview of other technologies that are somewhat similar and what differentiates them from this work.

Chapter 3 This chapter provides an overview about every library and technology that is used in this thesis and explains the basics that need to be known in order to understand this thesis.

Chapter 4 This chapter explains in detail the implementation of each component and how they are interlinked with each other.

Chapter 5 This chapter explains the demo that was used to test the implementation of the simulation and how it works.

Chapter 6 This chapter gives an overall conclusion about the motivation of this thesis and the finished implementation, along with an overview of its limitations and future work.

Chapter 2

Related Work

In the world of robotics there are multiple simulation environments for robots available. Examples include the Bullet World of CRAM [Mösnelechner, 2016] which is the ancestor to PyCRAM [Augsten and Augsten, 2019]. This will be explained in chapter 3.5.4.

However, a range of other robotic simulator are readily available, which will be described in the following chapter.

2.1 Gazebo

Gazebo is an open source simulation framework, written in C++, to simulate and test robot programs in an outdoor environment before using them on the real robot [Koenig and Howard, 2004].

The development of Gazebo began in fall 2002 at the university of Southern California by Dr. Andrew Howard and one of his students. The concept arose from the need to simulate robots in an outdoor environment. In 2009, Gazebo was integrated with ROS (Robot Operating System, explained in section 3.2) and the PR2. Since then it became one of the primary tools for simulation in the ROS community.

In 2012, the Open Source Robot Foundation took over the project and has been contributing to its development since.

Gazebo is built around a server-client architecture, the server providing all physics calculations. The client has a graphical interface to visualize the calculations, as well as being able to save the state of the simulation. This includes all objects and their configuration.

Gazebo is composed of different libraries, including one for communication, physics calculations, rendering of the simulation as well as libraries to generate sensor data and to visualize the simulation. The physics library can be changed to utilize different libraries. Though Gazebo normally uses the Open Dynamics Engine (ODE), it is also possible to use the Bullet physics engine, Simbody or the Dynamics Animation and Robotics Toolkit.

2.2 Actin

Actin is a proprietary robot simulation library that is developed by Energied Technologies. It is written in C++ and was initially released in 2005.

Actin was developed as a simulation and control software for the Johnson Space Center [Comstock, Lockney, and Glass, 2005].

This library provides a wide range of features that help with testing robots in the simulation. Its features include kinematic simulation, meaning Actin is capable of simulating forward and inverse kinematics in real-time, as well as providing attachments to other robots. Furthermore, Actin includes mechanics for visualizing the simulation which allow for highly detailed camera feedback. The rendering also utilizes NVIDIAs ray tracing technology for an even more realistic visualization. Actin also provides the simulation with various sensors including cameras, stereo vision, and range sensors.

The library has build-in support for network communication for either TCP or UDP, which allows the front- and backend components to be installed on different systems.

With these features there are a lot of possible use cases. One could, for example, use the simulator to evaluate the hardware of a robot with the targeted task, to determine if the robot is ideal for the given task.

Another possible use case is the evaluation of vision systems, as the rendering capabilities allow to simulate a vision system before using it in the real world. [Energid Technologies, 2004 – 2019]

2.3 Webots

Developed by the Swiss Federal Institue of Technology in 1996, Webots is a free open source robot simulator written in C++.

Even though it is available without additional costs, user support and consulting services are offered at 38 Euro and 188 Euro, respectively.

The targeted use of Webots include the simulation of autonomous cars, validation of robotic research results, and the training of human pilots.

To accomplish these goals, Webots provides a variety of features, including an accurate physics simulation, which is an extended version of the Open Dynamics Engine. Further, Webots contains vast libraries of robot models, objects, sensors and actuators to use in the simulation as well as an import/export system to import models and maps into the simulation; movies, screenshots and 3D animations, may also be exported.

To utilize these features, Webots has several APIs for different programming languages, including C, C++, Python, Java, Matlab, and ROS. In addition to that, Webots is platform independent: meaning runs on Windows as well as Linux and Mac OS [Swiss Federal Institute of Technology, 1996 – 2019].

Chapter 3

Foundation

In this chapter all required technologies that are needed to understand this thesis will be explained. This includes information about robotics and the Robot Operating System (ROS) as well as the Unified Robot Description Format (URDF), which is used in this thesis to represent the robot and it is environment.

Furthermore, there will be sections about the Bullet Physics Engine, which is a lightweight physics simulation library, and its Python wrapper PyBullet, which is used in this thesis.

The last sections of the chapter will be about CRAM, the Cognitive Robot Abstract Machine, which is a high-level robot control framework that enables the user to write highly dynamic robot control plans and the Python implementation of CRAM, PyCRAM, which currently does not support all features of CRAM.

3.1 Robotics Foundation

Because this thesis is submitted in the field of robotics it is important to understand the problems and methods of this research field.

Robotics is a branch of engineering and science which deals with the creation and control of robots. It is a rapidly growing field that emerged in the 1960s, with the first robots being master-slave arms that copied the movements of the human arm. These robots were used to handle nuclear material [Bruno and Oussama, 2016].

With the development of integrated circuits and miniaturized components, the robots could perform complicated tasks which lead to their use in the car manufacturing and other industry branches.

Throughout the years, robots have become part of the everyday life: whether it is in a warehouse transporting goods or in a household as a vacuum cleaner.

However, in order for these robots to function properly, they also need to be controlled. For this purpose, a wide variety of frameworks have been developed that run on the hardware of the respective robot. As the complexity of the control program increases along with the complexity of the environment, these control programs need to be highly structured.

Simulation plays a huge roll in the development and control of robots. In the development it is very helpful to be able to simulate the robot and the environment, so that the control programs can be simulated and tested before running on the real robot. In controlling the robot, the future actions of the robot can be simulated before performing them. This way it is possible to simulate the outcome of the action and determine if it is successful or fails.

3.2 ROS

The Robot Operating System is a framework that helps creating robot control programs. The aim of ROS is to simplify the process of creating robot software, by offering a wide variety of tools, libraries and conventions across both multiple platforms and programming languages.

ROS was developed with the idea that research groups all over the world can share their work and compliment each other. This way, every research group can contribute their work to the ROS ecosystem and also use the work of others to improve their own work.

ROS provides a lot of services, e.g., hardware abstraction, package management or messages between processes. The latter is key for the ROS ecosystem as it allows the programs of different research groups to communicate with each other via a unified interface and across different programming languages. This communication works a system called ROS nodes[Open Source Robot Foundation, 2007–2009].

3.3 URDF

URDF stands for Unified Robot Description Format, which is the most popular way of describing a robot in the ROS world. To have the description of a robot in this format is useful as one could, for example, use this representation to calculate the inverse kinematics. To perform this task, program needs an accurate representation of the robot, otherwise the results will not translate well into the real world.

Another use of the URDF is to load the robot into a simulated environment to test its actions before they are performed on the real robot. This way the robot can, for example, perform an action multiple times in the simulation, which enables the user to find the best way to perform the respective task before performing it in the real world.

One of the reasons URDF is popular is due to it being the standard representation of robots by ROS and all the software being linked to the ROS ecosystem.

An URDF is based on a XML notation that consists of link and joint tags. Links describe the single parts of the robot and the joints connect these parts. For example, one "finger" of a robot is a link, this link is connected to the "hand" of the robot by a joint.

While the main purpose of URDFs is to represent robots, it is also possible to represent entirely different things, like a kitchen, in which the robot operates. For example, a table would consist of 5 links, one for each leg and one for the tabletop. All

links are connected by joints. In addition, there could be a virtual joint which has no visual representation and functions as the root link of this table.

3.3.1 The Structure of an URDF

The link tag contains two other tags, the `visual` and `collision` tag. While the visual tag contains all tags that are taken into account for the visualization, the collision tag contains all tags that are needed to be accounted for the collision of this tag.

Both types have a `geometry` tag which specifies the exact dimensions of this link. There are a few possible ways to specify these dimensions. The first involves the utilization of primitive shapes like box, cylinder or sphere. The second one is to use a mesh file of type .dae, .obj, .stl or .mtl. A mesh is a polygon representation of an object.

The separate links are connected via joints. These joints have a type attribute, which determines how the joint behaves. There is a total of six different types of joints which are listed and explained in Table 3.1.

The different URDF joints and links can be used to represent a household environment in the simulation, specifically to describe furniture. A dining table would consists of five different links which are needed to describe this object. The surface and the four legs would be connected by fixed joints. On the other hand, a refrigerator door may use a revolute joint, whereas a drawer uses a prismatic joint.

All joints, except the fixed joint, have an upper and lower limit which specifies how far they can be moved.

In addition to the type and upper and lower limits, every joint also has a `parent`, `child`, and `origin` tag.

The parent and child tags determine the two links this joint connects. Here, parent link is the main link, whereas the child will be connected to this link. Typically, an URDF is a tree structure in which the nodes represent links and the edges represent joints. In other words, every link is the child of another link, except the root link. In an URDF there can only be one root link, even if the URDF describes multiple structures, like the furniture in a kitchen. The furniture root link may either be connected to a virtual point in space without a physical body, i.e., having a virtual link. Such information is key as all other positions are defined on this basis.

The origin tag determines the position of the child link; it will be given as coordinates in xyz format. The coordinates are relative to the origin of the parent link. This may sound inconvenient at first, but describing the position this way has the advantage that whole structures can be moved just by editing the origin of one joint [Open Source Robot Foundation, 2018].

3.4 Bullet Physics Engine

The Bullet Physics Engine is a C++ library for simulating physical interactions and determining the outcome of these interactions. To achieve this goal, there are several

TABLE 3.1: The types of joints URDF supports

Joint Type	Description
Fixed	A fixed joint cannot move in any direction, it is completely immoveable and mainly used to bind static parts together. For example, if one would want to describe the furniture of a kitchen with an URDF file, most of the joints would be fixed.
Continuous	A continuous joint has an upper and lower limit of positive and negative infinity and an axis around which it rotates. The continuous joint is used to describe, for example, wheels on a robot, as they can rotate endlessly in both directions.
Prismatic	A prismatic joint can only move along a single axis. It has upper and lower limits and an axis at along which it can move.
Revolute	The revolute joint is similar to the continuous joint, as it also rotates around one axis. However, this time there are upper and lower limit to the movement. With this joint one can, for example, describe the elbow of a human like robot.
Planar	The planar joint is similar to the prismatic joint but it can move along two axes. It also has, like the prismatic joint, an upper and lower limit. This joint can, for example, be used to describe the shoulder joint of a human like robot.
Floating	The floating joint is similar to the planar and prismatic joint, but it can move along all three axes.

different tools that can be provided by the library. The tools Bullet provides are collision detection as well as rigid and soft body dynamics. Bullet does not provide any tools for rendering the simulated rigid bodies or calculating kinematics. This has to be done by plugins.

Bullet was mainly developed by Erwin Coumans and Yunfei Bai, together with a big open source community. The targeted use of the Bullet library is the use in games, visual effects, and robotic simulations.

The rigid body dynamics Bullet offers simulate the impact of forces to linked rigid bodies; i.e., bodies that will not deform under the impact of external forces. For example this is used to simulate the robot, which is composed of many individual rigid bodies that are connected to each other by joints; which, in Bullet terminology, are called constraints.

To let bodies inside the simulation interact with each other, Bullet uses a concept called constraints. The constraints are Bullet's way to bind bodies together, so the movement of one body influences the other. these constrains work like a virtual joint of an URDF: they can either be of type fixed, prismatic, point2point (which is a planar joint) or of type gear (which is a continuous joint). A constraint can either be between two different bodies or between two parts of the same body. In both cases one needs to specify from which part of the first body to which part of the second

body the constraint should work.

An important tool of Bullet to detect and describe the interaction of different objects and bodies is through collision detection. This tool is specifically used to determine if two bodies intersect with each other. Different algorithms may be used to determine this scenario.

Bullet uses collision shapes for the collision detection. These collision shapes are assigned to the body and define the shape in which this body should collide with other collision shapes. The shape of the collision shape and the body do not have to match each other. For example, when simulating a robot and using the results in real-life, the collision shape could be bigger than the body to include a security margin. This way, the robot would stop before colliding with another object.

The collision detection can either be discrete or continuous: In other words, one can either query the collision of two bodies at one specific moment in the simulation, which would be a discrete collision detection. The other possible way would be to observe the collision of the two bodies constantly, which would be a continuous collision detection [Coumans and Bai, 2015 – 2019].

3.4.1 PyBullet

PyBullet is a Bullet physics-engine based Python module, which specialized in physics simulation for, i.a., robotics, games or machine learning. It provides support for loading bodies from URDF, SDF or MJCF files.

PyBullet provides a wide range of tools to interact with the physics simulation. Table 3.2 shows a selection of these tools with a short description.

In addition to these tools, PyBullet also provides bindings to rendering, with a CPU renderer and visualization. The rendering will be performed by TinyRenderer, which is a simple and lightweight renderer that only generates an output file but omits the visualization. OpenGL3 is used for the visualisation [Dmitry V. Sokolov, 20015 – 2019].

There is also support for virtual reality headsets like the HTC vive or the Oculus rift.

Another important feature of PyBullet is the built-in cross-platform client-server model which works via a variety of ways. For example, shared memory, UDP or TCP, which allows for network wide working with the same physics simulation. This is especially useful if there are more than one agent within the simulation.

For everything physics related PyBullet wraps the Bullet C-API. Other features like the virtual reality support or the rendering and the support for URDF, SDF and MJCF files are exclusively to the PyBullet framework and are implemented on top of the Bullet API [Coumans and Bai, 2016–2019].

TABLE 3.2: The tools provided by the PyBullet library

Tool	Description
Forward dynamics simulation	Simulates the movement of a body with the given torque and motors.
Inverse dynamics computation	Computes the torque each motor must deliver to achieve a given acceleration of a joint. The torque is computed using the recursive Newton Euler algorithm.
Forward kinematics	Simulates the position of the end-effector after a list of joint poses is applied.
Inverse kinematics	Computes the joint poses needed to achieve a given position of the end-effector. The joint poses are computed using an improved version of Samuel Buss' inverse kinematics library.
Collision detection	Determines if two bodies are in contact with each other. For this to be determined, one may use the following: closest points, overlapping pairs, etc.
Ray intersection queries	Computes the first object that intersects with a line from a given start to a given end point.

3.5 CRAM High-level Robot Control Framework

CRAM stands for Cognitive Robot Abstract Machine which refers to a framework for high-level robot controlling, written in Common LISP. It was developed by Lorenz Mösenlechner for his PhD thesis and is currently expanded by the CRAM team of the Institute of Artificial Intelligence of the University of Bremen, Germany.

The aim of CRAM is it to create robot control programs for highly dynamical environments, such as a human household. Unlike static environments, like an industrial facility, dynamic settings pose the challenge of changing positions of needed objects over time.

As mentioned above, CRAM is used for high-level robot control; these control programs are called plans in the CRAM framework. Plans are a structured course of actions a robot needs to perform to achieve a given task. The goal of CRAM is that the plans are independent of the robot they are being executed on. To achieve this goal, CRAM uses a concept called process modules which are explained in Section 3.5.2. To access the process modules another concept called designator is used. Designator are a way to parameterize an action that should be performed. Designators will be explained in detail in Section 3.5.1.

The CRAM framework also implements a reasoning engine which should be used for every reasoning task that may arise during plan execution or the resolving of designators. This engine consist of a prolog interpreter, written in Common Lisp. It is used for resolving designators because, depending of the current situation of the robot, the solution of a designator can vary. If, for example, the robot should pick up an object and needs to choose an arm to do so, it may be possible that only one arm

can reach the object.

3.5.1 Designators

Because robot control plans need to be highly flexible, the parameters of this plan needed for execution can only be decided on when they are needed. For example, the pose of a robot to perform a specific action can only be decided right before the action. This is the case because in a highly dynamic environment, there may be obstacles that were created during the execution of the plan [Kazhoyan and Beetz, 2017].

In CRAM plans these, parameters are described by designators. Currently, there are four types of designators, which are listed and described in Table 3.3.

The designators consist of key-value pairs, with each pair the possible resoultions

TABLE 3.3: The designators available in CRAM

Designator	Description
Location	Describes a location. This location will be generated by a generator function and then checked by a verification function.
Action	Describes an action like picking or placing a object on a table.
Object	Describes an object with its name, type and position.
Motion	Describes the low-level controlling of the robot, like the joint movements or the perception.

of a designator are more restricted. For example, a location designator would look like this:

```
1 (a location (part-of kitchen) (on counter-top))
```

This describes a location in the kitchen on a counter. The pair `(part-of kitchen)` limits the possible solutions to the kitchen and the second pair limits this further to only locations which are on top of the counter. There are still infinite possible solutions because every point that fulfills these restrictions is a valid location.

3.5.2 Process Modules

Process module are the concept which makes CRAM usable for an arbitrary robot. Process modules are called by the motion designators and contain the code for the actual movement of the robot. These process modules can be changed based on the robot the plan should be executed on, without the need to adapt the high-level plan to the robot platform.

The aim of this concept is that it, for example, is not relevant if a robot has one, two or three arms, the plan should still execute as intended. The only thing that needs to be changed are the low-level motion commands in the process modules and the reference of the motion designator.

The decision, which process module should be used, is done by a group of prolog statements. These statements reason about the current situation of the robot and try to choose the best fitted process module.

3.5.3 CRAM Plan Language

The CRAM Plan Language (CPL) is a domain specific language based on Common Lisp. It was designed to help designing the CRAM plans by native supporting parallel execution of commands, special exception handling mechanisms (which allow to retry the failed code with new parameters), and the usage of objects between different threads while ensuring the integrity of the data.

The advantage of using Common Lisp is that it is easily extensible with macro mechanisms, which make it easy to implement a domain specific language like CPL. This makes the language more flexible and allows it to adapt to various situations.

Table 3.4 shows the most important and most common used expressions of the CPL. Another important feature that the CPL provides are Fluents, which are thread-safe

TABLE 3.4: Selection of the most important expressions of CPL

Expression	Description
par	Executes an arbitrary amount of expressions, and in multiple threads. It fails if throws an exception.
seq	Executes all expressions sequentially. It terminates if one throws an exception.
try_all	Executes all expressions parallel in different threads, but does not terminate if one throws an exception.
pursue	Executes all expressions parallel in different threads and terminates if one of the expressions terminates.

objects. The need for thead-safe objects arose because of CPLs multi threading abilities.

A fluent is an object which stores a value, which can either be a numerical or lexical value. This value can be modified by all threads while the fluent ensures the integrity of the stored data.

This can be very useful, for example, to store the current position of the robot. As the position is needed by a lot of components in different threads, making it available available across all threads while still ensuring the integrity of the position is critical [vgl. Mösnelechner, 2016].

3.5.4 Bullet World

CRAM provides several tools for testing plans, such as the Bullet World along with its reasoning mechanics. The BulletWorld is a physics-based simulation based on the Bullet physics engine. Using this simulation, plans can be executed and tested. To assure a close representation of the real world, CRAM supports loading bodies

from URDF, collada, obj, and stl files. This gives the user the ability to rebuild the environment in which the real robot operates, thus making it possible to transfer the results of the simulation to the real world.

The included reasoning mechanics allow for the user to gain insight into the current state of the simulation through requesting semantic queries about the current relation of objects in the simulation. By gaining this insight, the user is able to plan the behavior of the robot accordingly.

There is a wide variety of different reasoning queries, such as a query to check if two objects are in contact with each other or whenever an object is visible for the robot.

To verify all requests to the Bullet World, only requests via prolog queries are allowed. This ensures that the status of the simulation is valid at all times and no false situation is created.

3.6 PyCRAM

PyCRAM is the Python re-implementation of CRAM, developed by Andy and Dustin Augsten [Augsten and Augsten, 2019].

Currently, PyCRAM provides the CRAM Plan Language, designators and process modules. Most of the features work similar to CRAM, but due to the great difference between Python and LISP a few mechanics needed to be adapted. An example includes the omitted prolog reasoning engine. Thus, the reference of the designators works fundamentally different. In addition, the decision which process module to choose differs between CRAM and PyCRAM.

As a result of the lacking prolog engine the reference of the designators cannot use prolog queries. Instead, nested if statements are used to check and return the effective designator.

Further, PyCRAM has no integrated simulator, meaning all code that is written has to be tested on the real robot.

The following chapter thus illustrates the implementation of a lightweight physics-based simulator with which a user can write plans using PyCRAM plan language and motion designators, which are performed inside the simulation instead of the real robot. Furthermore, the simulation allows the robot to do physics-based reasoning at runtime.

Chapter 4

Implementation

My contribution to the PyCRAM framework is the implementation of a robot simulation environment based on PyBullet, which allows fast simulation of actions a robot could perform as well as the corresponding impact on objects involved in these actions. The simulation environment is called BulletWorld corresponding to the name of the underlying library.

To check the situation of the simulation reasoning mechanisms were added, which allow to query the simulation. Query examples include methods to determine if one object is stable at the current simulation time or whether or not an object is reachable. The methods and mechanisms related to the implementation are presented in the following sections.

4.1 Architecture

Figure 4.1 shows all implemented components and how they interact with each other.

Starting from the low-level, BulletWorld is the main class of the simulation which initializes and manages all aspects of the simulation. It relies on the GUI class to initialize the simulation. The BulletWorld with all its methods will be explained in detail in Section 4.2.

The GUI class is a wrapper to initialize the simulation and maintain it. How it works will be explained in section 4.2.2.

To represent objects in the simulation the Object class is used. Any instance of this class represents one object in the simulation. After initialization, each instance is registered in the corresponding BulletWorld. The Object class with all its methods and how they work is explained in section 4.3.

The BulletWorldReasoning class provides the user with reasoning capabilities; for this purpose it relies on the BulletWorld to provide the object instances or to interact with the simulation. All available reasoning queries will be explained in Section 4.4.

The BulletWorld, Object and GUI class can be considered wrapper for the PyBullet library. In addition, to this they provide methods for management and interactions. The BulletWorldReasoning class uses these classes to provide its reasoning capabilities.

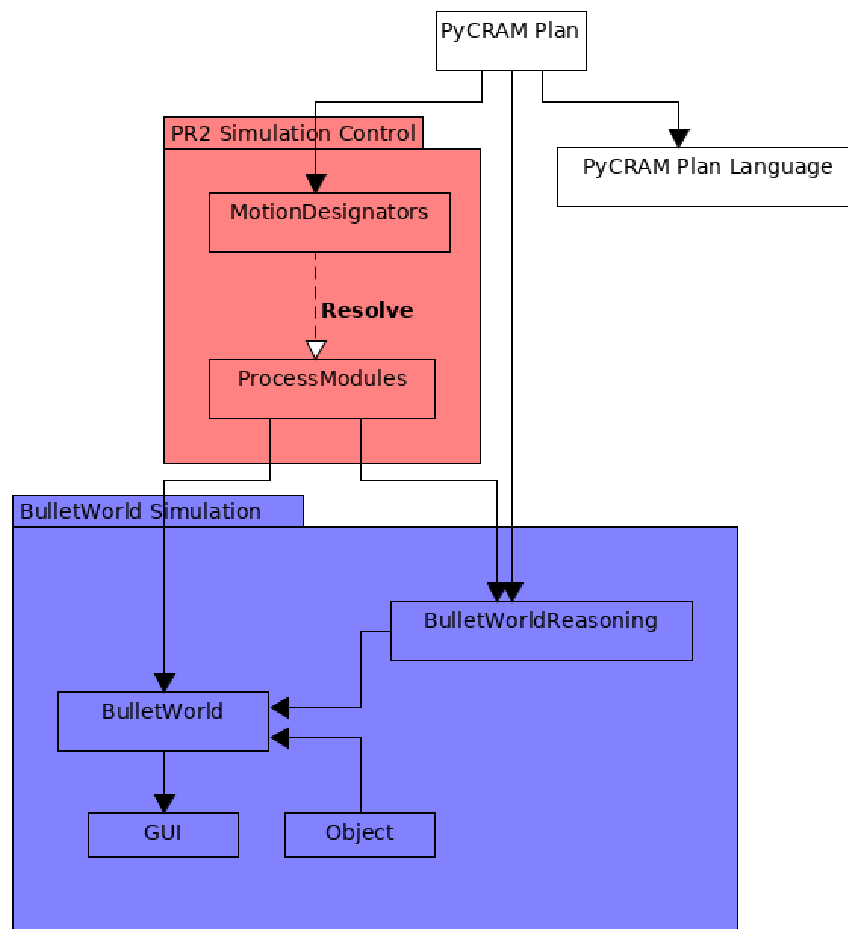


FIGURE 4.1: The structure of the BulletWorld, GUI, BulletWorldReasoning and Object classes along with the Motion Designators and Process Modules for the PR2.

To control the robot inside the simulation, the motion designators and process modules are used. The Motion Designator provide the user with an easy interface, while the Process Modules execute the actual control code which moves the robot inside the simulation.

All implemented motion designators are listed and explained in Section 4.5.

A list with all process modules can be found in Section 4.7.

4.2 Bullet World

The BulletWorld is the environment in which all actions will be simulated. It can be considered a wrapper around the physics client of PyBullet. The BulletWorld class provides mainly methods controlling the simulation as well as getting objects that are present within the simulation.

Because PyBullet allows more than one physics client it is also possible with the BulletWorld. In other words, the user can have multiple instances of the BulletWorld, each with different scenarios. A BulletWorld instance can be either of type direct or gui. With type direct there is no visible feedback of the actions in the simulation and the simulation can only be affected by methods that change the status of an object. In gui mode, however, there is a window in which the simulation is displayed and the objects can be dragged around with the mouse.

Because of technical restrictions it is only possible to have one graphical simulation, the rest will have to be executed in direct mode.

Table 4.1 shows the methods provided by the BulletWorld class.

There are methods to get objects from the BulletWorld either by their name, type or id, to set the gravity, and to simulate the BulletWorld for a given time. Furthermore, all event references are stored so that all events are unique for each instance of the BulletWorld. Currently, the only events available are for attachment, detachment, and manipulation.

TABLE 4.1: Table of methods the BulletWorld class provides

Method	Description
get_objects_by_name	Returns a list of all objects with the given name.
get_object_by_id	Returns the object corresponding to the given ID.
get_attachment_event	Returns the instance of the event which is called when two objects are attached.
get_detachment_event	Returns the instance of the event which is called when two objects are detached.
get_manipulation_event	Returns the instance of the event which is called when the environment was manipulated.
set_realtime	Sets the realtime in the simulation to True or False.
set_gravity	Sets the gravity of the simulation to the given vector.
simualte	Runs the simualtion for a given amount of seconds.
exit	Terminates the simulation.

The following example will demonstrate the method set_gravity: The method takes a vector of type x,y,z as parameter and defines it as the gravity in the simulation. Setting the same gravity as in the real world (a constant vector with a average magnitude of 9.81 $\frac{m}{s^2}$ along the z axis) in the simulation would look like this:

```
1 world.set_gravity([0, 0, -9.8])
```

The simulation in PyBullet works in steps, where every step equals approximately $\frac{1}{240}$ seconds. Some methods of PyBullet need a simulation step to work: For example, the collision detection may only function if the step_simulation method was executed at least once before.

The simulate method takes the amount of seconds that should be simulated as a float, so that it is also possible to simulate fractions of a second. Next, the method loops the exact number of steps to simulate the given time.

Another possible way is to increase the time one step simulates. However, this would come with a loss in precision.

Simulations can be ended using the exit method. This sets the current_bullet_world to an active BulletWorld and collects the threads. It is mainly used for working with multiple Bullet Worlds because problems may arise if one of the instances is not correctly terminated. Otherwise, the threads would not be collected and continue to allocate RAM. When working with only one it is not necessary to use the exit method, though it is recommended to do so.

4.2.1 Current Bullet World

The current_bullet_world variable is the default world for all methods. It will always point to the last initialized BulletWorld. To ensure that the current_bullet_world always points on a valid BulletWorld every, instance saves the previous current_bullet_world and resets it once it is finished; making this effectively a single linked list.

4.2.2 GUI

If the BulletWorld instance is run in the gui mode a window will show up, providing insight into the simulation. This is shown in Figure 4.2.

However, because of the way PyBullet works, this window is not persistent. Meaning, it will close as soon as the script finishes. This is a problem as the user cannot observe the situation within the simulation after all statements are executed.

To solve this problem, a new thread will be created when the BulletWorld is instantiated. This thread will be permanently kept active by an endless loop that checks whether the simulation is still active. If it is, the thread will sleep for 10 seconds and then be checked again.

```
1 def run(self):
2         if self.type == "GUI":
3                 self.world.client_id = p.connect(p.GUI)
4         else:
5                 self.world.client_id = p.connect(p.DIRECT)
6
7         while p.isConnected(self.world.client_id):
8                 time.sleep(10)
```

This code is executed in the new thread.

The if-else statement decides if the new BulletWorld is in gui or direct mode and

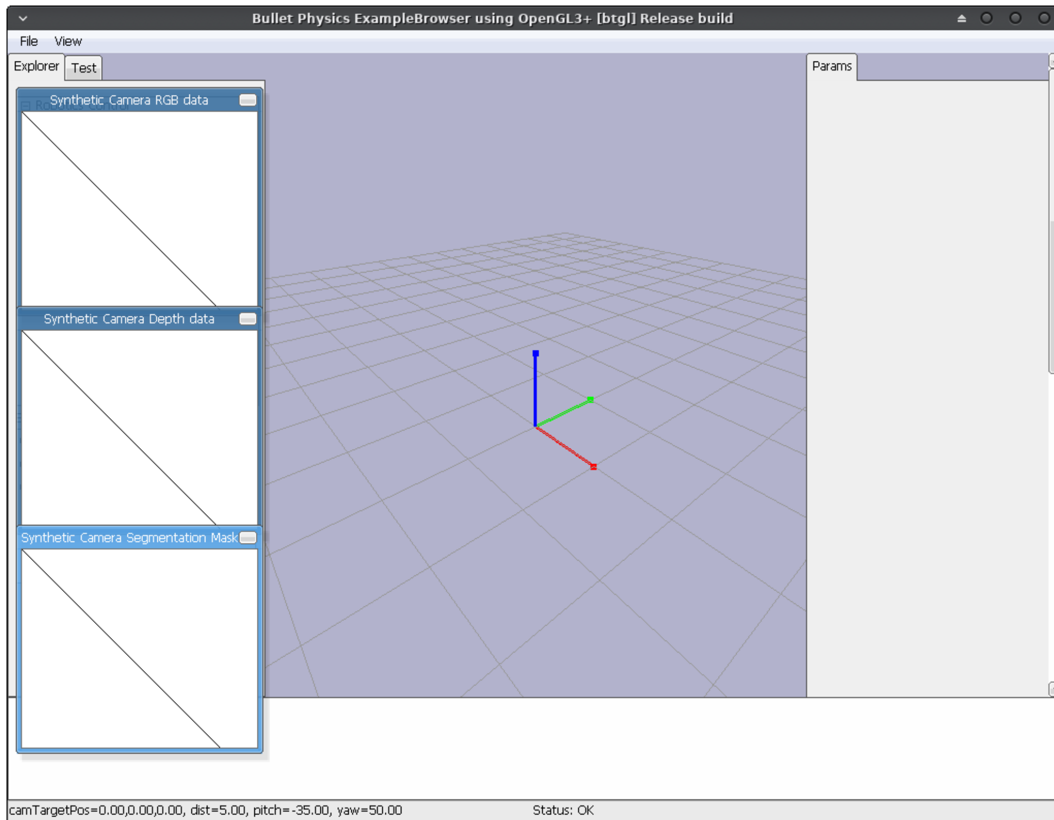


FIGURE 4.2: An empty simulation

sets the id of the corresponding instance. Next, the loop checks if the simulation is still running and lets the thread sleep if that is the case. This will continue until the simulation is terminated.

4.2.3 Events

Events are used in case a scenario occurs to which multiple services need to respond. In PyCRAM, an own implementation of events is used. The reason for this addition is that Python does not offer a standard implementation of events.

The implementation is in the class Event. Using this addition, the user can create an instance of this class and register or remove an arbitrary number of handlers, in addition to calling the event. When the event is called all registered handler are called.

For an easier use the operators += and -= are overloaded. With the += operator it is possible to add a handler to the event.

```
1 attachment_event += handler_function
```

The -= operator works similar; however, it removes the handler from an event. This is shown in the following code example:

```
1 attachment_event -= handler_function
```

Two ways may be used to call an event: the first is to call the `fire` method of the event. The second is to call the event directly. This looks like this:

```
1 attachment_event()
```

4.3 Objects

The Object class represents anything inside the simulation regardless of whether it refers to a robot, the environment (like a kitchen) or an item (such as a mug). This makes the working with objects easier because one does not have to specify the nature of the given object. On the downside, this approach limits the functionality of the methods because the methods cannot be designed with a special use case in mind.

```
1 Object(name, type, path, position=[0, 0, 0], orientation=[0, 0, 0, 1],
      world=None, color=[1, 1, 1 ,1])
```

The constructor of the object class requires only two arguments, the rest is optional. The first is the name of this object: it may be arbitrary as it is only used to identify the object.

The type of the object specifies the exact type of this object: For example, the type of a kitchen would be "environment". This makes it simple to cluster objects and easily identify a specific group of objects. One instance in which this may be helpful is when excluding environmental objects, such as the kitchen or the floor, from reasoning queries.

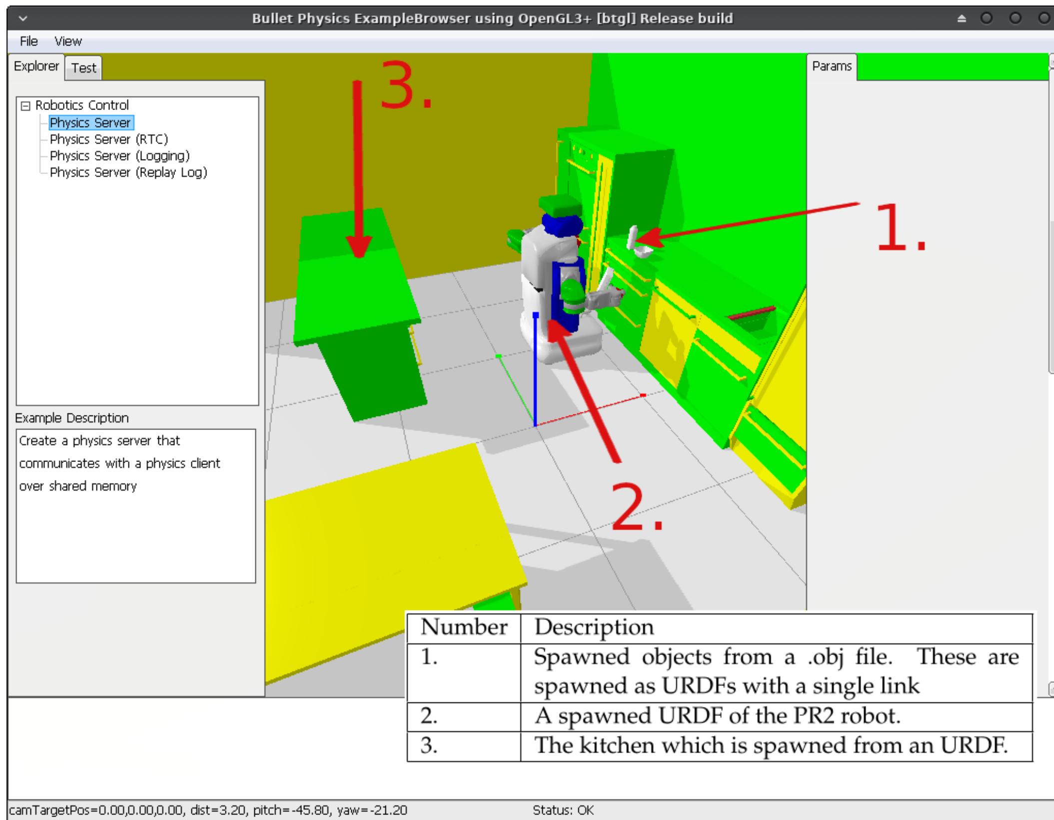
The path argument is the path to the file, which describes this object. An object can either be generated from an URDF, stl or obj file. If the given file is of stl or obj format an URDF file with one link, that contains this file as mesh, will be generated and spawned. This needs to be performed as PyBullet does not directly support stl or obj files. In this process it is possible to define an individual color so that meshes can be spawned with different colors: For URDF files this is not necessary as the URDF format supports independent colors for every link.

The position and orientation arguments define the position and orientation in which the object should be spawned. These are represented as lists of x, y, z in world coordinate frame for the position and as a list representing a quaternion for the orientation. A quaternion is a list of 4 numeric values that describe the rotation around each axis as well as a normalization value.

The world argument specifies a BulletWorld in which the object should be spawned. If no world is given, the current_bullet_world is used. The argument is set to None and not directly to the reference because the standard values will only be evaluated once and then be set. This is problematic when dealing with object references instead of static values. As a workaround for this problem the current_bullet_world will be set in the constructor if the world argument is None.

The color argument only works when an obj or stl file is provided. The color is given as a list of RGBA.

Figure 4.3 shows a PR2 and a kitchen inside the simulation. Everything spawned is an instance of the Object class.



Number	Description
1.	Spawned objects from a .obj file. These are spawned as URDFs with a single link
2.	A spawned URDF of the PR2 robot.
3.	The kitchen which is spawned from an URDF.

FIGURE 4.3: PR2 in the kitchen with a few objects

The methods the class provides are show in Table 4.2:
The attach and detach methods will be explained in the next section.

The other listed methods are mostly used for either querying the state of one object or to manipulate this state. Most of the methods are simple getters and setters, such as the method set_position_and_orientation changes the position of an object in the simulation or the method get_pose returns the base position of an object in world coordinate frame.

The two methods get_joint_id and get_link_id return the internal id of a specific joint or link. This is necessary because PyBullet does not use the names of joints or links but instead abstracts these by an id. That is required in all PyBullet methods when referring to a link or joint. Because of the time it takes to traverse all links, especially when the model is very complex (e.g., a kitchen), two dictionaries will be created in the constructor which translate the name of one joint or link to the corresponding id.

4.3.1 Attachments

The attachments are mainly used to simulate pick-up actions, because the robot and the object it picks up are two independent objects. As they are also simulated independently, attachments are needed. The attach method creates a virtual fixed joint between the two objects that should be attached to each other.

TABLE 4.2: The methods of the class `bullet_world_reasoning`

Method	Description
attach	Attaches this object to the other given object.
detach	Detaches this object from the other given object. This method only works if they were previously attached.
get_position	Returns the position of this object in world coordinate frame.
get_pose	An alias for `get_position`.
get_orientation	Returns the orientation of this object as quanternion.
set_position_and_orientation	Sets the position and orientation of this object.
set_position	Sets the base position of this object, the position must be given as a list of xyz.
set_orientation	Sets the orientation of this object, the position must be given as a list representing a quanternion.
get_joint_id	Returns the unique ID of the given joint name.
get_link_id	Returns the unique ID of the given link name.
get_link_position	Returns the position of a given link of this object, the link is specified by its name.
get_link_orientation	Returns the orientation of a given link of this object, the link is specified by its name.
get_link_position_and_orientation	Returns the position and orientation of a given link as a list.

This way of utilizing attachments has several advantages: Firstly, fewer code has to be written which results in fewer room for errors. Only a few transformations, to calculate the relative poses between the two objects, have to be performed. The second advantage is that the attachments are more precise this way, because they are resolved by PyBullet directly.

Nonetheless, there are also disadvantages to this way of using the attachments. This includes e.g., that the virtual joint is only resolved if the simulation is running. However, this is the opposite of the use that was intended (to teleport between positions) The simulation should only be running when performing physics-related tasks.

To keep track of all attachments, every object has a dictionary with the attached object as keys and the unique id of the attachment as value. The id is returned by the PyBullet method which creates the virtual joint. This dictionary is identical for both objects of the attachment, meaning, it is irrelevant on which object the detachment method is called because both know the unique id of the attachment.

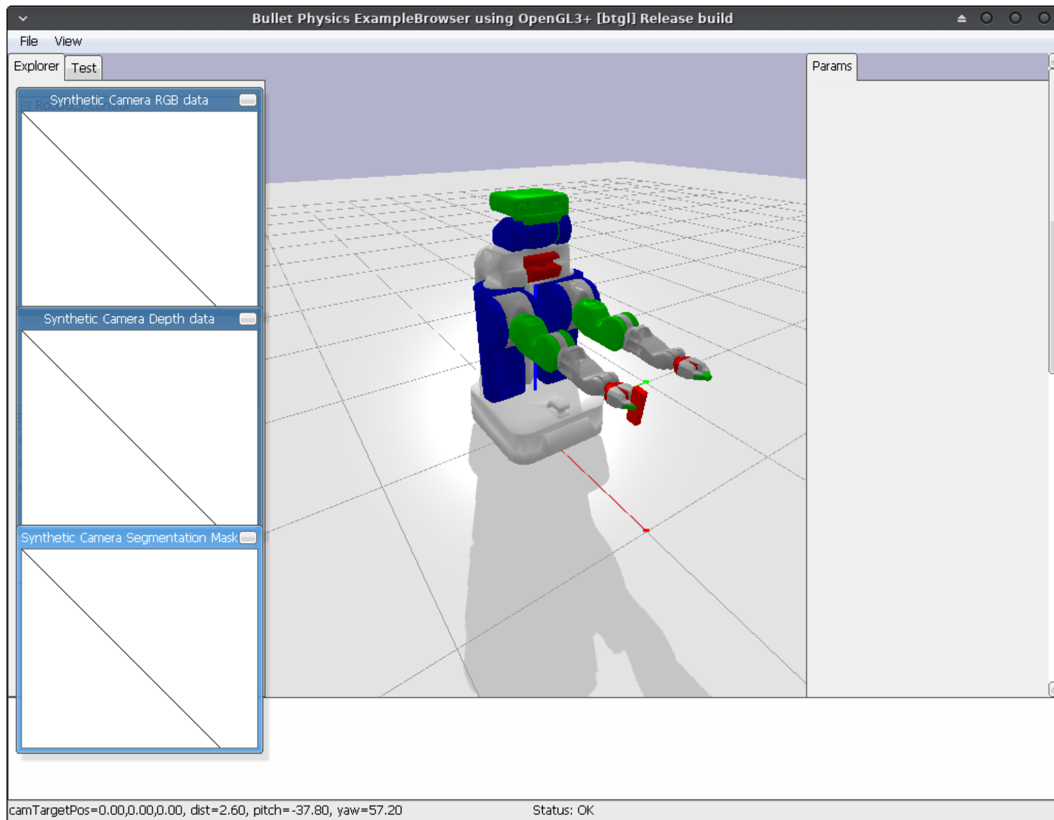


FIGURE 4.4: The PR2 with an attached object

To attach two objects to another, the user just needs to call the attach method on one of the objects and provide the other object as a parameter.

If robot and fork are the names of the objects, the attach method looks like this:

```
1 robot.attach(fork)
```

This would be enough to attach the objects, but without further parameters the attachment would be between the base position of the robot and the fork. This is not ideal as a robot would hold a fork with its hands and we want to simulate the attachment this way. For that purpose it is possible to specify the links between which the attachment should be created.
This would look like this:

```
1 robot.attach(fork, "r_gripper_tool_frame", None)
```

The robot with an attached object can be seen in Figure 4.4. This creates the joint between the "r_gripper_tool_frame" link of the robot and the base position of the fork. The "None" instead of a link name means that the base position should be used. This is mainly for attaching objects which are not created from a stl or obj file as these do not have links which can be provided.

Lastly, it is important to mention is that it is not possible to call the `attach` method again if there is already an attachment between the given objects. The reason for this is that it would create a second joint between the objects and override the entry in the dictionary of all attachments and create an irremovable joint. If the user were

to call the attach method with an existing attachment, it would just return without having altered anything.

4.3.2 Detachments

The detachments can be considered straight-forward because only the virtual joint needs to be removed. With the removal, the two objects become again independent of one another.

To detach two objects, the user just needs to call the detach method on one object and provide the other object as a parameter.

```
1 robot.detach(fork)
```

Because of the dictionary with all attachments, the method can look up the unique joint id, if there is one, and delete the joint; thus making the objects independent again.

4.4 Bullet World Reasoning

This section takes a look at the reasoning that is provided by PyCRAM and explains the different types. The reasoning consists of semantic queries one can utilize to get a better understanding of the situation inside the simulation. For example, there are methods to determine if two objects are colliding or if one object is visible from a specific point (like the visual sensor of a robot). In the following sections these queries will be referred to as reasoning queries.

Table 4.3 shows a list of all available reasoning queries with a short description.

TABLE 4.3: A list of all reasoning queries

Query	Description
Stable	Checks if the given object is stable in the simulation. An object is considered stable if it will not change its position if the simulation is running.
Contact	Checks if two given objects are in contact with each other.
Visible	Checks if the given object is visible from a given position.
Occluding	Returns a list of objects that occlude the object.
Reachable_object	Checks if a given object is reachable for a given robot.
Reachblae_position	Checks if a given position is reachable for a given robot.
Blocking	Returns a list of objects that are in contact with the robot if it were to reach the object.

4.4.1 Stable

This reasoning query determines if an object is stable inside the simulation, i.e., that the object does not change over a longer period of time. This is the case if the object is, for example, placed on top of a table or a counter.

To accomplish this, the coordinates of the objects will be saved at the beginning for later comparison. In addition, the current state of the world will also be saved so it can be restored later. Next, the world will be simulated for an equivalent of 2 seconds and the current coordinates will be saved. Because of how precise PyBullet works, the coordinates cannot be directly compared. Instead they will be rounded to three decimal places that correspond to the precision of 1 millimeter. Now, the previous saved state will be restored and the result of the comparison will be returned.

This reasoning query returns "True" if the object did not move while it was simulated (i.e., stable) and returns "False" in any other case.

```
1 stable(bottle)
```

For being utilized, the reasoning query only requires the object, and optionally, the world when multiple worlds are used.

4.4.2 Contact

This reasoning query is similar to a collision detection. It returns "True" if the two given objects are in contact and "False" in any other case. To achieve this, the collision detection get_contact_points method of PyBullet can be used. To determine if there are contact points the simulation needs to step one time (the mechanics of the simulation are explained in Section 4.2).

The usage of this method is shown in the following:

```
1 contact(kitchen, bottle)
```

All that is needed are the two objects for which the contact should be determined, and optionally, the world when multiple worlds are used.

4.4.3 Visible

This reasoning query determines if an object is visible from a specific position, e.g., the camera mount of a robot. The reason why the position and not the robot needs to be given is that the framework should be able to work with all kinds of robots. Thus, no assumptions about the robot can be made. For easier use, the object class has a method called get_link_position which, when given a link name, returns the exact position of this link in the world coordinate frame.

Because this reasoning query uses images in its process, they need to be rendered from the simulation. Details regarding the used rendering software are explained in Section 3.4.1

To determine if the object is visible it needs to be rendered two times. The first time only the object alone will be rendered and all visible pixels are counted. This sets the maximal number of pixel that are visible from this position. The second time all objects will be rendered and the visible pixels are counted again. This value is the actual number of pixels that are visible from the given position.

To determine if the object is visible, the method checks if the actual number divided by the maximum number of pixels is greater than 0.8. This corresponds to at least 80 % of the object being visible. Before dividing the number it will be checked if the maximum number of pixels is greater than zero, to prevent a division by zero. If

the maximum number of pixels is zero "False" will be returned, because the object is determined to be not visible.

To use the reasoning query, the user needs to specify the object and a position of the camera. This would look like this:

```
1 visible(bottle, robot.get_link_position("camera_mount_link"))
```

4.4.4 Occluding

This reasoning query returns a list of objects that are between one given object and the camera position. This is useful if, for example, a robot wants to get something out of a shelf and check if anything is in front of that object.

This reasoning uses rendered images. Details on the used rendering software are given in Section 3.4.1.

This reasoning query is similar to the query of visible in the way that the scene needs to be rendered multiple times. First, the object is again rendered alone. This time, the pixels are not counted but, instead, the position of pixel that belong to the given object are saved as tuples in a list. The second time, the whole scene is rendered and the previously saved positions of the pixels belonging to the object are checked. If the object is still visible in this pixel nothing happens, but if another object is visible this object is occluding the given object and will be saved in a list and then returned.

This reasoning query needs, like visible, an object and the position of the camera. The call is very similar to the one above.

```
1 occluding(bottle, robot.get_link_position("camera_mount_link"))
```

4.4.5 Reachable

The reachable reasoning query determines, like the name suggests, if a given object is reachable by a robot. This reasoning query needs an object that is a robot and the link name of one gripper of this robot. The name of the gripper needs to be given because PyBullet should work with arbitrary robots. Optionally, one can set a threshold for how close the gripper needs to be to the object for the reasoning query to return "True". The standard threshold is one millimeter.

Because this reasoning query alters the state of the simulation, it will be saved before the predicate and restored afterwards.

To check if an object is reachable, the same steps as in a real scenario will be performed. In other words, first, the inverse kinematics will be calculated and applied. Afterwards, the distance between gripper and object will be calculated if it is less than the threshold, the reasoning query returns "True".

For calculating the inverse kinematics, the ik solver of PyBullet is used. This solver is an improved version of Samuel Buss' inverse kinematics library.

The usage would look like this:

```
1 reachable(bottle, robot, "r_gripper_tool_frame")
```

In some cases, it can be useful to determine if a position, not an object, is reachable. One such case is, for example, if one wants to check if the robot can reach the target position when placing an object. For this purpose, there are two reachable reasoning queries: one that takes an object and another that takes a position in the world coordinate frame.

In theory, it would be possible to not only use the tool frame but every other link described in the URDF. So, it would also be possible to use the link of the arm. The reasoning query would still work if the arm can reach the object.

4.4.6 Blocking

This reasoning query determines the objects that would block the robot if it was to reach for the given object. It is very similar to the reachable predicate in the way that it needs an object, which is a robot, and a link name of one gripper.

Because this is also a reasoning query that changes the state of the simulation, it will be saved before the execution and restored afterwards.

To begin, the reasoning query calculates again the inverse kinematics for the robot to reach the object. Then, this will be applied to the robot. Afterwards, it will be checked for every object in the world if it is in contact with the robot and the list with objects that are in contact will be returned.

This approach is not ideal because, for example, the floor is also considered an ordinary object and the robot is always in contact with the floor. Thus, lists of blocking objects would always contain the floor.

However, this problem is difficult to solve as assumptions about the objects would have to be made. For example, one could check the name for something like "floor" or "plane" but there is no guarantee that the user would name the floor like this. To solve this problem, the type attribute of objects is used: For example, one could set the type of objects, such as the floor or other environment, to "environment" and filter for these after the reasoning query is finished.

Again, this call is similar to the one of visible and would look like this:

```
1 blocking(bottle, robot, "r_gripper_tool_frame")
```

4.4.7 Supporting

The supporting reasoning query determines if one object supports the other. For example, if a bottle is sitting on a table than the table is supporting the bottle. For this to be true there are two conditions that have to be met.

Firstly, the second object needs to be above the first object and, secondly, the two objects need to be in contact.

To check these conditions, first the z coordinates of the objects are compared. If the one of the second object are greater than those of the first, the condition is met. For the second condition the contact predicate is used. If booth conditions are met the predicate returns "True" and "False" in any other case.

The usage of this predicate needs the two objects, which are or are not supporting each other:

```
1 supporting(kitchen, bottle)
```

It can, for example, be used to find all objects that are placed on a table. For this, the user needs to traverse through all objects and check if they are supported by the table.

4.5 Designators

To use the Bullet World with the PR2 a wide variety of designator with the corresponding process modules were implemented. In this section, all available designator will be listed and explained together with all required and optional slots.

Table 4.4 shows an overview of all available designator, with a short description of each one.

TABLE 4.4: All available designator

Designator	Description
Moving	Moves the robot to a given position and checks if the robot is in collision with other objects.
Pick-Up	Picks up an object with the given type and attaches it to the gripper.
Place	Places the given object and detaches it from the robot.
Accessing	Opens a drawer, to access the objects within.
Looking	Moves the head to look at a given position.
Opening-Gripper	Opens the gripper of the given arm.
Closing-Gripper	Closes the gripper of the given arm.
Detecting	Tries to find an object of the given type within the field of view.
Move-TCP	Moves the tool center point of the given arm to a given position.
Move-Arm-Joints	Moves the joints of one or both arms to a given or pre-defined position.
World-State-Detecting	Detects an object with the belief state of the world. The robot does not need to see the object.

4.5.1 Moving

The moving designator is used to move the robot to the designed position. It is also possible to specify an orientation, but it is not mandatory to do so.

Table 4.5 shows all slots for this designator along with a short description and whether it is mandatory or optional.

TABLE 4.5: All slots of the moving designator

Slots	Description	Required
Type	Defines the type of this designator.	Yes
Target	Defines the destination for the robot.	Yes
Orientation	Defines the orientation of the robot.	No

4.5.2 Pick-Up

The pick-up designator is used to pick up an object with the given arm. Although the arm slot is not necessary for the execution of the designator, it is encouraged to use it for better results.

Table 4.6 shows all available slots with a short description and if this slot is mandatory.

TABLE 4.6: All slots of the pick-up designator

Slot	Description	Required
Type	Defines the type of this designator.	Yes
Object	The object that should be picked up.	Yes
Arm	The arm with which the object should be picked up. If no arm is given, the left is used.	No

4.5.3 Accessing

The accessing designator is used to open drawers to access the objects within. The drawer handle and joint need to be specified because no semantic knowledge about the drawer is available.

Table 4.7 shows all slots with a short description and if this slot is necessary.

4.5.4 Looking

The looking designator is used to move the head of the robot, to look at a given position. The position has to be in world coordinate frame. Alternatively, it is possible to specify an object for the robot to look at.

Table 4.8 gives all slots along with a short description and if this slot is necessary. Even thought neither target, nor object are mandatory, one of them has to be provided.

4.5.5 Opening-Gripper

This designator is used to control the grippers of the robot. It can either open the left or the right gripper.

TABLE 4.7: All slots of the accessing designator

Slot	Description	Required
Type	Defines the type of this designator.	Yes
Drawer-Handle	The name of the drawer handle, how it is specified in the URDF. This is the point which the robot grips to open the drawer.	Yes
Drawer-Joint	The name of the prismatic joint the drawer is connected to.	Yes
Part-of	The object of which the drawer is a part of.	Yes
Distance	The distance, how wide the drawer should be opened.	No
Arm	The arm with which the robot should open the drawer.	No

TABLE 4.8: All slots of the looking designator

Slot	Description	Required
Type	Defines the type of this designator.	Yes
Target	The position for the robot to look at.	No
Object	The object at which the robot should look.	No

Table 4.9 shows all available slots with a short description and if the slots are necessary.

TABLE 4.9: All slots of the opening-gripper designator

Slot	Description	Required
Type	Defines the type of this designator.	Yes
Gripper	Specifies the gripper which should be opened, either left or right.	Yes

4.5.6 Closing-Gripper

This designator is used to control the gripper of the robot. It can either close the left or right gripper of the robot.

Table 4.10 shows all slots with a short description and if the slot is necessary.

4.5.7 Detecting

This designator is used to detect an object that is in front of the robot and returns it to the user.

Table 4.11 shows all available slots with a short description and if the slot is necessary.

TABLE 4.10: All slots of the closing-gripper designator

Slot	Description	Required.
Type	Defines the type of this designator.	Yes
Gripper	Specifies the gripper which should be closed, either left or right.	Yes

TABLE 4.11: All slots of the detecting designator

Slot	Description	Required
Type	Defines the type of this designator.	Yes
Object-Type	The type of the object that should be detected.	Yes

4.5.8 Move-TCP

This designator is used to move the Tool Center Point of one of the arms of the robot. The arm does not have to be given. If no arm is given, the left is used.

Table 4.12 shows all slots with a short description and if the slot is mandatory.

TABLE 4.12: All slots of the move-TCP designator

Slot	Description	Required
Type	Defines the type of this designator.	Yes
Target	The target to which the tcp should be moved.	Yes
Arm	The arm of which the tcp should be moved.	No

4.5.9 Move-Arm-Joints

This designator is used to manipulate the joints of the robot arms. The user can either do this by giving a list of joint values, that will then be applied to the joints or to use the pre-defined joint states. This can be used by giving, instead of a list, the string "park".

Table 4.13 shows all available slots with a short description and if it is necessary. Even though neither the left, nor right arm is mandatory to be specified, one of them has to be provided.

TABLE 4.13: All slots of the move-arm-joints designator

Slot	Description	Required
Type	Defines the type of this designator.	Yes
Left-Arm	Defines the configuration for the left arm.	No
Right-Arm	Defines the configuration for the right arm.	No

4.5.10 World-State-Detecting

This designator is used to detect an object within the state of the world. Thus, the robot does not need to see the object to detect it. This can be useful as, in some cases, the robot is not able to see an object but will still able to grasp it. For example, this is the case if the robot wants to grasp object from the lowest level of a shelf.

Table 4.14 shows all slots with a description and if the slot is required.

TABLE 4.14: All slots of the world-state-detecting designator

Slots	Description	Required
Type	Defines the type of this designator.	Yes
Object-Type	The type of the object that should be detected.	Yes

4.6 Referencing

To be used in the process modules, the motion designator needs to be referenced. In this process some parameters are being transformed or missing ones are inferred. The result of this is a dictionary which contains the needed key, value pairs to execute this process module.

The reference of a designator uses a cascade of if-statements that determine which parameters are given and which need to be inserted. Listing 4.1 shows how this is done for the moving motion designator.

```
1 if desig.check_constraints([('type', 'moving'), 'target']):
2         if desig.check_constraints(['orientation']):
3                 solutions.append(desig.make_dictionary([('cmd', 'navigate'
                      ), 'target', 'orientation']))
4         solutions.append(desig.make_dictionary([('cmd', 'navigate'), '
              target', ('orientation', BulletWorld.robot.get_orientation())])
              )
```

LISTING 4.1: The code to reference the motion designator for moving the robot

4.7 Process Modules

To just have the designator is not enough for the robot to work with the simulation, because the designator is just the access point for the low-level code that moves the robot in the simulation.

This low-level code is implemented in the process modules, which are called through the designator. The process modules are as atomic as possible, each one is responsible for one movement and checking that the robot is in a valid position; in other words, that the robot is not colliding with anything and is in the right position.

Every process module is responsible for one physical resource; for example, there is a process module for moving the robot, for the gripper, moving the head, etc. Usually, the motion designator will be called from an action designator, but these are not yet implemented. So, motion designators were used for every action to be performed.

Table 4.15 shows all available process modules with a short description.

TABLE 4.15: All process modules

Process Module	Description
Pr2Navigation	Moves the robot to a specific position and checks if it is in collision with objects in the environment.
Pr2PickUp	Moves the gripper to the given object and attaches it to the robot.
Pr2Place	Moves the gripper to the given target location and detaches the object from the robot.
Pr2Accessing	Moves the gripper to the given drawer and opens it a given distance.
Pr2MoveHead	Moves the head joints to look at a given position in the world coordinate frame.
Pr2MoveGripper	Opens or closes the gripper.
Pr2Detecting	Tries to detect an object from a given type. The object has to be visible for the robot.
Pr2MoveTCP	Moves the tool center point of a given arm.
Pr2MoveJoints	Sets the joints of one arm either to a given list of poses or to the pre-defined parking position.
Pr2WorldStateDetecting	Tries to detect an object of a given type from the world state. The object does not need to be visible for the robot.

4.8 Choosing a Process Module

At the top-level are the motion designators which contain the desired motion and the parameters needed to perform this motion. This motion designator is then handed over to the process module which determines, by the type, which process module to choose. The code to choose the right process module for a motion designator can be seen in Listing 4.2.

```
1 def available_process_modules(desig):
2         if desig.check_constraints([('type', 'moving')]):
3                 return pr2_navigation
4
5         if desig.check_constraints([('type', 'pick-up')]):
6                 return pr2_pick_up
7
8         if desig.check_constraints([('type', 'place')]):
9                 return pr2_place
```

```
10
11          if desig.check_constraints([('type', 'accessing')]):
12                  return pr2_accessing
13  ...
```

LISTING 4.2: An example of a code to determine a process module for a motion designator. The provided code is equal for every other process module.

After choosing the right process module it will be executed and move the robot. Listing 4.3 shows the process module responsible for moving the robot.

```
1   class Pr2Navigation(ProcessModule):
2           def _execute(self, desig):
3                   solution = desig.reference()
4                   if solution['cmd'] == 'navigate':
5                           robot = BulletWorld.robot
6                           robot.set_position_and_orientation(solution['
                                target'], solution['orientation'])
7                           for obj in BulletWorld.current_bullet_world.
                                objects:
8                                   if btr.contact(robot, obj):
9                                           if obj.name == "floor":
10                                                  continue
```

LISTING 4.3: The process module for moving the robot from one position to another.

The first statement references the designator. This can be seen in Listing 4.1. After that, it will again be checked that this process module and the referenced designator match. Next, the robot object will be retrieved from the BulletWorld and its position will be changed. Furthermore, it will be checked if the robot is in collision with anything, except the floor.

4.9 Control Flow

Now that all parts of the BulletWorld, BulletWorld Reasoning, motion deisgnator and process modules are explained, the control flow of all parts and how they interact with each other can be introduced.

The following bullet points explain step by step the control flow, while Figure 4.5 shows this explanation in a visual way.

A typical control flow in this architecture would be like this:

1. The user executes a designator in the high-level plan.
2. The right process module for this designator will be chosen.
3. The process module references the motion designator.
4. The solution for the motion designator is returned.
5. The process module queries the BulletWorld for the required objects.
6. The BulletWorld returns the required objects.
7. The process module calls the BulletWorldReasoning.

8. The BulletWorldReasoning then queries the BulletWorld for the objects.
9. The BulletWorld returns the required objects.
10. The BulletWorldReasoning finishes the reasoning query and returns the result to the process module.
11. The process module then interacts with the objects in the simulation according to the results of the reasoning query.

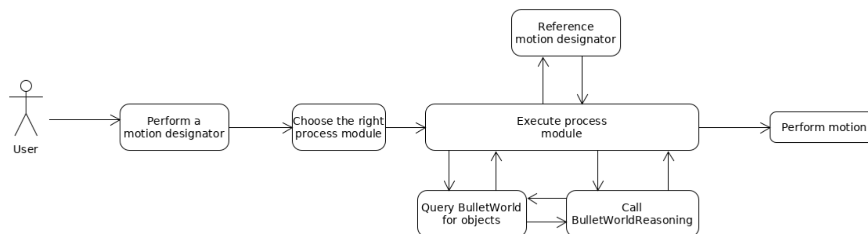


FIGURE 4.5: A visual representation of the control flow

Chapter 5

Evaluation

5.1 Demo

To test the implementation of the simulation and reasoning capabilities, a simple demo was implemented. This demo uses the simulation and various reasoning queries to realize a simple pick and place plan, which is then used to set the table in a kitchen.

For this scenario, the robot needs to pick up the objects from the counter and place them in the right position on the table. Both the object types and placing positions are hand-written, because inferring the types of objects that are being required or finding suitable placing poses are not part of the current thesis project.

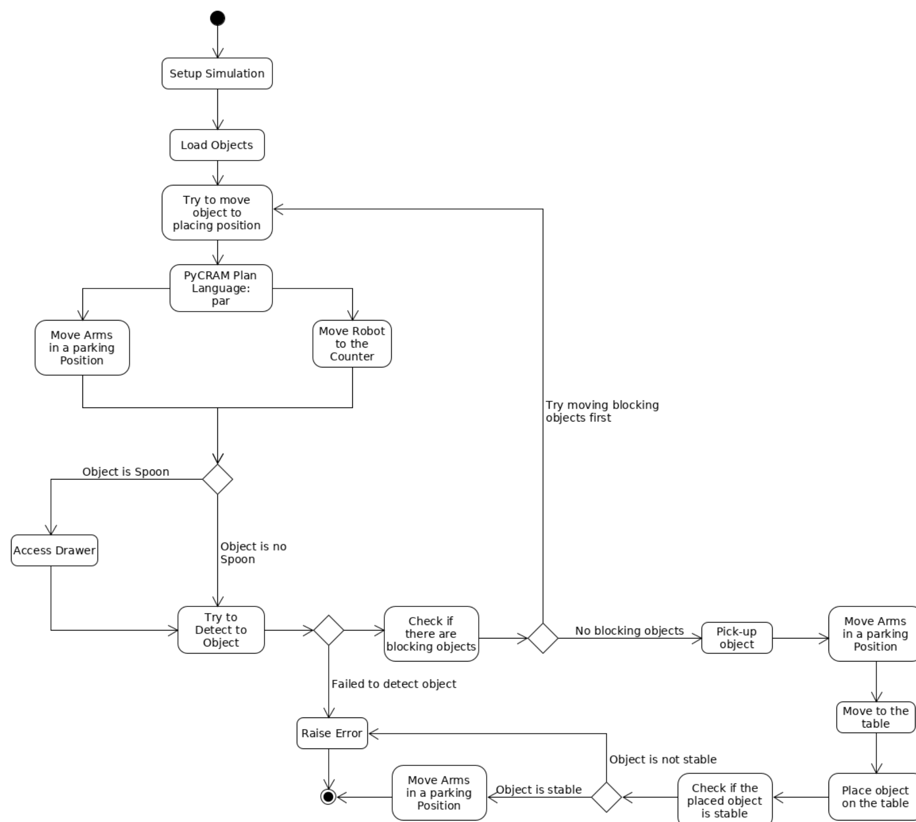


FIGURE 5.1: An UML activity diagram of the demo

Figure 5.1 shows the sequence of events in the demo.
Firstly, the simulation is initialized, which includes instantiating the BulletWorld and setting the gravity, and the objects are loaded. After all preparations are complete, the execution of the pick and place plan can begin. This is shown in Figure 5.2. The code that initializes the demo scenario is provided in Listing 5.1.

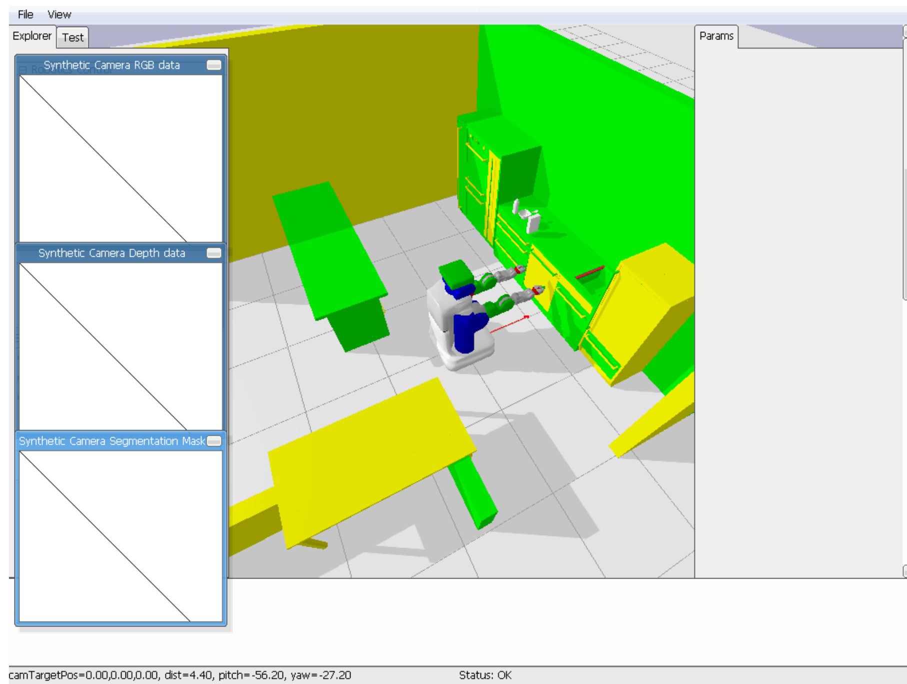


FIGURE 5.2: The simulation after initializing and loading all objects

```
1  world = BulletWorld()
2  world.set_gravity([0, 0, -9.8])
3  plane = Object("floor", "environment", "../plane.urdf", world=world)
4  robot = Object("pr2", "robot", "../pr2.urdf")
5  kitchen = Object("kitchen", "environment", "../kitchen.urdf")
6  milk = Object("milk", "milk", "../resources/milk.stl", [1.3, 1, 1])
7  spoon = Object("spoon", "spoon", "../resources/spoon.stl", [1.4, 0.8, 1])
8  cereal = Object("cereal", "cereal", "../resources/breakfast_cereal.stl",
       [1.3, 0.6, 1])
9  bowl = Object("bowl", "bowl", "../resources/bowl.stl", [1.3, 0.8, 1])
10 BulletWorld.robot = robot
```

LISTING 5.1: The code to initialize the demo scenario

To begin, the robot moves to the counter and, in parallel, moves its arms in a parking position. After arriving at the counter, it will be checked if the object to be picked up is the spoon. If so, the robot will open the drawer to access the spoon within. This can be seen in Figure 5.3. The motion designator for this movement can be seen in Listing 5.2.

```
1  ProcessModule.perform(MotionDesignator([('type', 'accessing'), ('drawer-
       joint', 'sink_area_left_upper_drawer_main_joint'), ('drawer-handle', '
       sink_area_left_upper_drawer_handle'), ('arm', 'left'), ('distance',
       0.3), ('part-of', kitchen)]))
```

LISTING 5.2: The motion designator to open a drawer

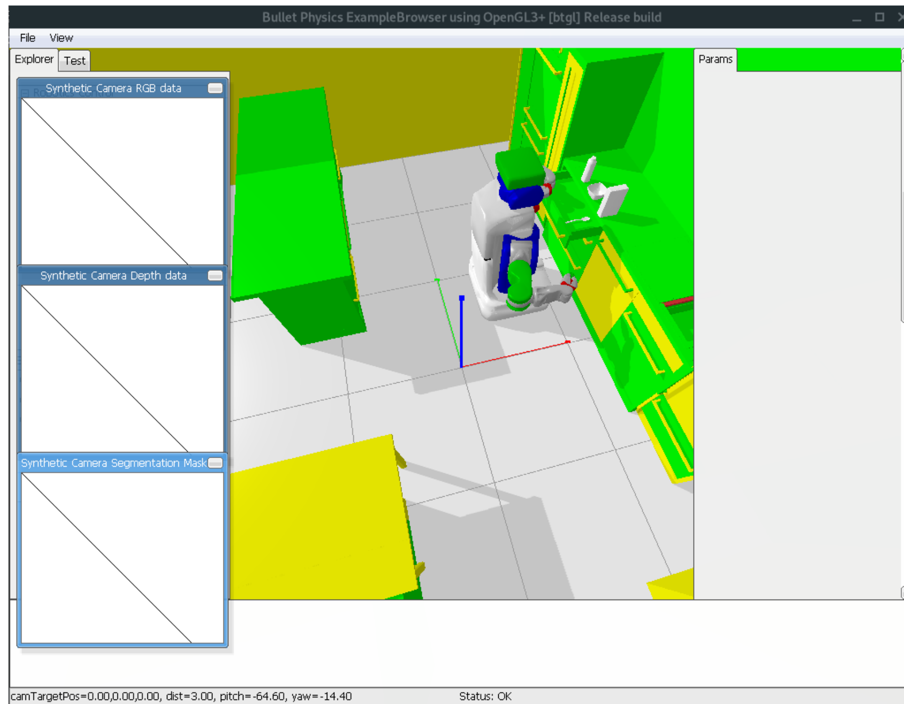


FIGURE 5.3: The PR2 opening a drawer

Next, the robot will try to detect an object in front of it with the given type. If this action fails, the plan raises a PerceptionError and terminates. Figure 5.4 shows the robot after the detecting. The code to move the robot is shown in Listing 5.3.

```
1 with par as s:
2         ProcessModule.perform(MotionDesignator([('type', 'move-arm-joints'
              ), ('left-arm', 'park'), ('right-arm', 'park')]))
3         ProcessModule.perform(MotionDesignator([('type', 'moving'), ('
              target', [0.65, 0.7, 0]), ('orientation', [0, 0, 0, 1])]))
4 ProcessModule.perform(MotionDesignator([('type', 'looking'), ('target',
      [1.3, 0.6, 1])]))
5 det_obj = ProcessModule.perform(MotionDesignator([('type', 'detecting'), (
      'object', object_type)]))
```

LISTING 5.3: The motion designators to move the robot to the counter and park its arms

After successfully detecting the object it will be checked if there are blocking objects. For this purpose, all environmental object are filtered out. If after this no blocking objects are found, the object will be picked up and the plan continues. If there are blocking objects, the plan will be recursively called with the blocking object. The robot, after having picked up the object, is shown in Figure 5.5. The code for determining blocking objects and picking up objects is shown in Listing 5.4.

```
1 block = btr.blocking(det_obj, BulletWorld.robot, gripper)
2 block_new = list(filter(lambda obj: obj.type != "environment", block))
3
4 if block_new:
5         move_object(block_new[0].type, targets[block_new[0].type][0],
              targets[block_new[0].type][1])
```

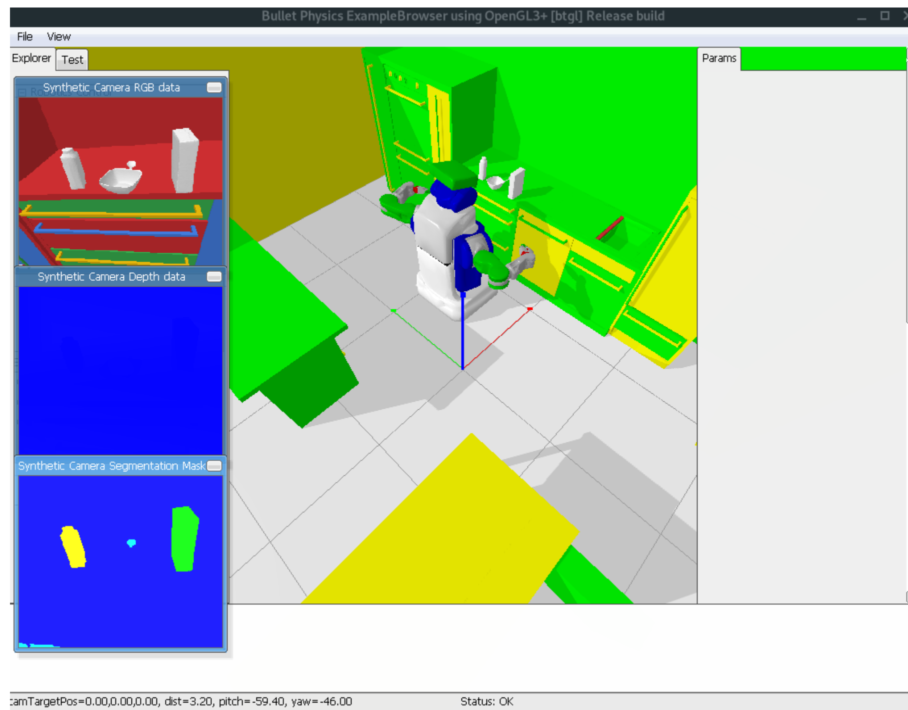


FIGURE 5.4: The robot after trying to detect the object

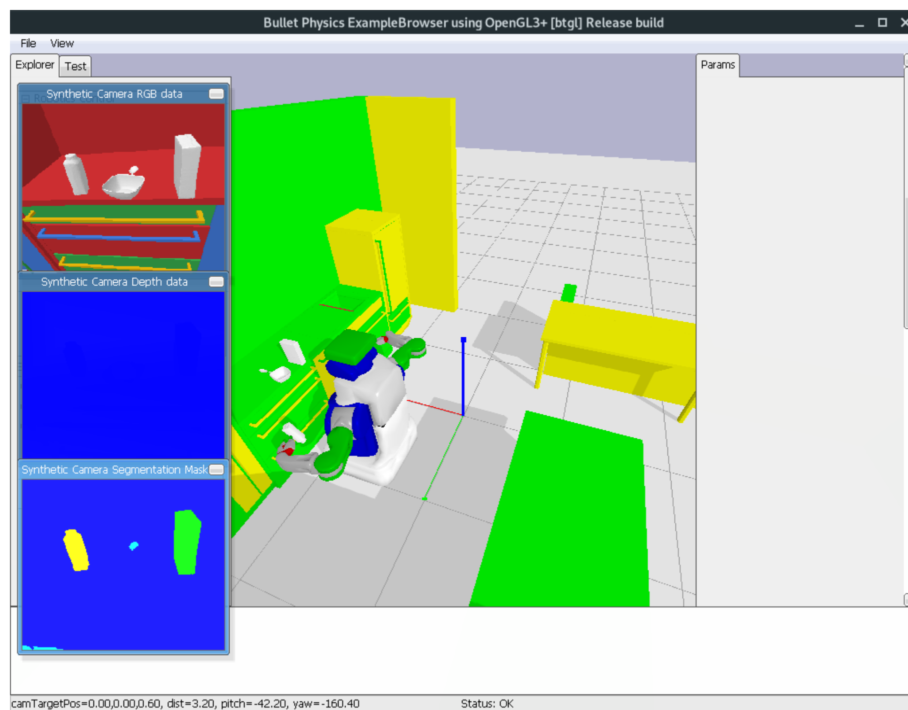


FIGURE 5.5: The robot after picking up the given object

```
6           ProcessModule.perform(MotionDesignator([('type', 'moving'), ('
                target', [0.65, 0.7, 0]), ('orientation', [0, 0, 0, 1])]))
7
8   ProcessModule.perform(MotionDesignator([('type', 'pick-up'), ('object',
        det_obj), ('arm', arm)]))
```

```
9
10 ProcessModule.perform(MotionDesignator([('type', 'move-arm-joints'), ('
       right-arm', 'park')]))
```

LISTING 5.4: The code for determine blocking objects and picking up object.

After transporting the blocking object out of the way, the robot will move the original object to the table and place it on the given placing position. When being placed, it will be checked whether the object is stable. If it is, the plan terminates. If not, a ReasoningError will be raised.

The robot, after having placed the object and checked object stability, is shown in Figure 5.6. The code needed to perform this action is shown in Listing 5.5.

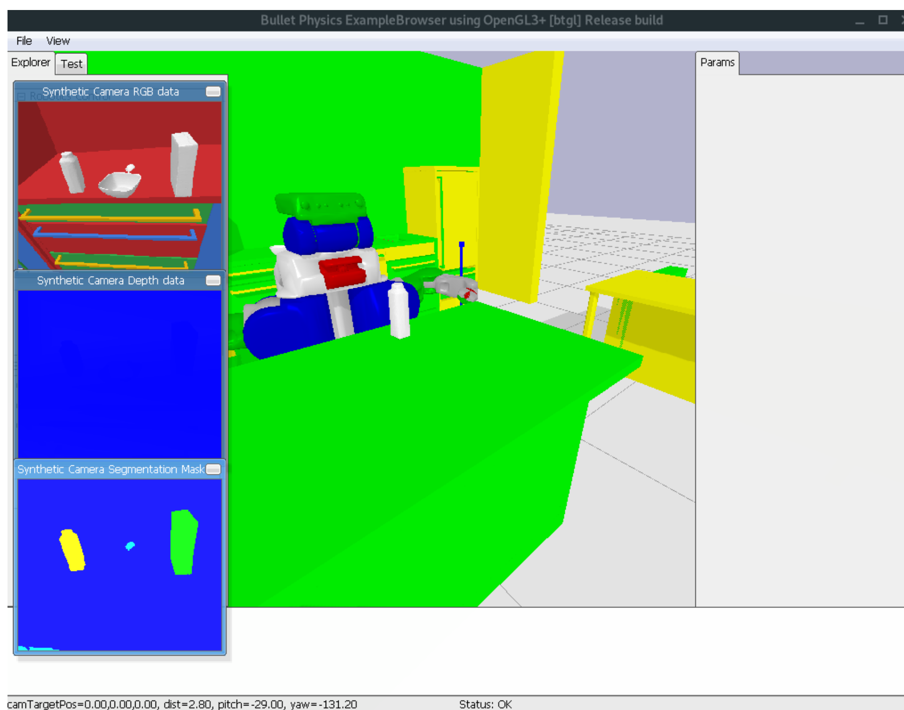


FIGURE 5.6: The robot after placing the object on the table

```
1  ProcessModule.perform(MotionDesignator([('type', 'moving'), ('target',
       [-0.3, 1, 0]), ('orientation', [0, 0, 1, 0])]))
2
3  if btr.reachable_pose():
4          ProcessModule.perform(MotionDesignator([('type', 'place'), ('
               object', det_obj), ('target', target), ('arm', arm)]))
5
6  ProcessModule.perform(MotionDesignator([('type', 'move-arm-joints'), ('
       left-arm', 'park'), ('right-arm', 'park')]))
7  print("placed: ", object_type)
8
9  if not btr.stable(det_obj):
10         raise btr.ReasoningError
```

LISTING 5.5: The code to determine if the target position is reachable and checking if the objects is stable after placing it.

Chapter 6

Conclusion

6.1 Summary

With this thesis, PyCRAM was extended with simulation and reasoning capabilities as well as the required code to include the PR2 into the simulation and control him.

Because this library should be used by others, the greatest emphasis while developing the BulletWorld and Object class was on the user experience. The goal was to make both the use of the BulletWorld, and the process of loading objects into it, as undemanding as possible.

With this in mind, no wrapper class for positions in the world were created; instead, lists were used. The same goes for the orientation of the robot.

The reasoning methods are also implemented with the user experience in mind. Every method is static, so only the required arguments are needed to fulfill the task. There is a wide variety of different reasoning methods, including the contact between two objects, visibility of an object for the robot or blocking objects when the robot tries to grip an object.

All these methods work by the same principle: save the world state, alter the simulation in a way to fulfill the given task, check if the state of the simulation satisfies the task, reset the world state to the saved one and, lastly, return the result.

The motion designators allow the user to easily parameterize a motion that should be performed. In some cases it is not even necessary to provide all parameters because the designator can either infer them by itself or use standard values. In addition to this, the process modules allow for a platform independent implementation by wrapping the low-level code and allow the user to decide which platform the plan should be executed on. Thus, the demo presented in Section 5.1 can also be executed on the real PR2 with the right process modules.

6.2 Discussion

In the BulletWorld everything inside the simulation is an instance of the Object class. Thus, the user can interact with the entirety via the same methods. This prevents multiple cases when handling objects. But because it is useful in some cases to difference between different types of objects, the Object class has a type attribute which allows the clustering of objects.

Although more a conceptional than an implementation limitation of the approach to simulate motions before executing them is that one needs an accurate representation of both the robot and the environment. Because, if the results cannot be used in the real world, it does not make sense to simulate them in the first place. However, most manufacturer do provide an exact model of their robots, enabling its simulation.

A prominent disadvantage of representing everything inside the BulletWorld as an instance of the Object class is that it can be difficult to distinguish between objects of interest and ones that can be ignored. One example for this is the reasoning query blocking, because it returns every object with which the robot is in contact with when it reaches for the target object; this includes, e.g., the floor. This can be evaded by using the type attribute of the objects and filtering for objects, like the floor. If the user wants to exclude one particular object, the object name can also be used.

Another limitation of the Object class is that the methods provided are very limited and only provide basic management and information capabilities. Any further mechanics have to be implemented by the user, because the later use cannot be foreseen yet.

Because PyCRAM should work with any robot but some of the reasoning queries need the robot object, the user needs to set the robot variable in BulletWorld class. This can be done when initially setting up the simulation, making it accessible from every point of the program.

Theoretically, is it possible to initialize multiple BulletWorld instances and run them in parallel. But because this was not a priority during the development, it might currently be unstable and not all methods may work. However, this can be fixed in a future version and will also be discussed in Section 6.3.

A limitation of the attachments included is that they only work while the simulation is active: This is the case because the virtual joint that is created for the attachment is only resolve in a simulation step. For the attachment to work properly, the simulation needs to be run every time the robot moves, either itself or its arms.

6.3 Future Work

While the BulletWorld and its reasoning work satisfactorily and can easily be used, there are a few problems that either need to be fixed or were more work needs to be invested, in order for the simulation work properly.

Firstly, though working as intended, the attachments' virtual joints currently need a few simulation steps to be resolved. This is problematic because the simulation should only run when determining physics-related questions and not while the robot moves from one position to another.

This can be fixed by saving the id of the links that the attachments use and moving the object to the corresponding link when the robot or its arms move. While implementing this, one needs to be aware that attachments are bilateral, and the id

must be saved in both objects.

Moreover, the capability to use multiple BulletWorlds at the same time needs to be improved. While this is rudimentarily implemented, it was no priority in the presented development process. Thus, it may be the case that not all methods work properly with multiple BulletWorlds. Furthermore, the management of the current_bullet_world is not very stable and may break if one BulletWorld is not ended properly. While currently the described steps (Section 4.2) need to be undertaken, future work may improve the safe termination of the simulation.

To implement this, one needs to ensure that every call to the PyBullet library contains the parameter to specify the targeted BulletWorld. In addition to this, the management of the current_bullet_world needs to be improved to ensure it always contains a valid BulletWorld.

Regarding the reachable and blocking reasoning query, no orientation of the end effector is taken into account. Thus, it is currently not possible to grasp from a specific side. The inverse kinematic solver offers the possibility to specify a target orientation. The only addition to be made is the creation of a dictionary which translates the grasp side to the orientation.

To make PyCRAM more attractive for others to use, it would be a good idea to integrate other robot platforms with PyCRAM. This includes implementing the process modules and motion designator for the respective robot platform.

This thesis is only a small step to a complete reimplementation of CRAM in Python as there are still a range of features and functions that need to be implemented. Examples include the addition of more types of designators, location distribution to distribute regions in space or the sampling of possible positions.

The next logical implementation aspect would be the projection, meaning the simulation of actions before performing them in the real world, as projection without a simulation is impossible. With the projection it would be also helpful to implement task trees which are used for introspection, to determine what action failed during the execution and to find another way around the problem. The aforementioned additions may assist in the development of a user-friendly re-implementation of CRAM in Python, to successfully simulate robots in dynamic environments.

Bibliography

Augsten, Andy and Dustin Augsten (2019). “PyCRAM - Python-based concurrent reactive programming language for autonomous mobile manipulation”. Universität Bremen.

Bruno, Siciliano and Khatib Oussama (2016). *Springer Handbook of Robotics -*. 2. Aufl. Berlin, Heidelberg: Springer. ISBN: 978-3-319-32552-1.

Comstock, Douglas, Daniel Lockney, and Coleman Glass (2005). “A structure for Capturing Quantitative Benefits from the Transfer of Space and Aeronautics Technology”. In:

Coumans, Erwin and Yunfei Bai (2015 – 2019). *Bullet User Manual*. https://github.com/bulletphysics/bullet3/blob/master/docs/Bullet_User_Manual.pdf.

— (2016–2019). *PyBullet, a Python module for physics simulation for games, robotics and machine learning*. http://pybullet.org.

Dmitry V. Sokolov (20015 – 2019). *Tiny Renderer*. https://github.com/ssloy/tinyrenderer.

Energid Technologies (2004 – 2019). *Actin Simulation Capabilities*. https://www.energid.com/actin/simulation-capabilities.

Kazhoyan, Gayane and Michael Beetz (2017). “Programming Robotic Agents with Action Descriptions”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Vancouver, Canada, pp. 103–108. DOI: 10.1109/IROS.2017.8202144.

Koenig, Nathan P. and Andrew Howard (2004). “Design and use paradigms for Gazebo, an open-source multi-robot simulator”. In: *IEEE*.

Mösnelechner, Lorenz (2016). “The Cognitive Robot Abstract Machine”. PhD thesis. Technische Universität München.

Open Source Robot Foundation (2007–2009). *About ROS*. https://www.ros.org/about-ros/.

— (2018). *Building a Movable Robot with URDF*. https://wiki.ros.org/urdf/Tutorials/Building%20a%20Movable%20Robot%20Model%20with%20URDF.

Statistisches Bundesamt (2019). *Bevölkerung im Erwerbsalter sinkt bis 2035 voraussichtlich um 4 bis 6 Millionen*. BevölkerungimErwerbsaltersinktbis2035voraussichtlichum4bis6Millionen.

Swiss Federal Institute of Technology (1996 – 2019). *Webots*. https://cyberbotics.com/.