Universität Bremen
Faculty 3, Mathematics and Computer Science

Masterthesis

Teaching robots manipulation skills with human controlled robots bodies - Spend a day in Virtual Reality inside a robots body

VR-basierte Methoden zur Vermittlung von Manipulationsfähigkeiten in der Robotik
- Verbringe den Tag in einem Roboterkörper in einer virtuellen Realität.

Alina Hawkin
2926383

Supervisor:	Prof. Michael Beetz PhD
Second Supervisor:	Dr. René Weller
Advisor:	Gayane Kazhoyan
Advisor:	Andrei Haidu

December 7, 2020

Declaration of Authorship

I, Alina Hawkin, declare that this thesis, titled "Teaching robots manipulation skills with human controlled robots bodies -
Spend a day in Virtual Reality inside a robots body" and the work presented in it are my own and has been generated by me as the result of my own original research. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at the University of Bremen.
- 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.

[illegible]

Alina Hawkin

07.12.2020, Bremen

Date

Abstract

Teaching robots how to perform everyday household activities while at the same time, experiencing how it is like to perform these tasks as a robot. This thesis proposes the autogeneration of a skeletal mesh of the PR2 robot and its transfer into a Virtual Reality environment, where a human user can perform everyday household tasks from inside the virtual robots body. In order make the virtual robot move as realistic as possible, an inverse kinematics solver is implemented and used to calculate positions of the arms for the virtual robot. Since the same inverse kinematics solver is used on the virtual robot as on the real one, it can be assured that if the virtual robot was able to perform a certain action and a kinematic solution was found, the same movement will be possible with the real robot in the real world. Furthermore, several adjustments are implemented in order to map the human body to the robots one, including the limits a robots body would introduce. The Goal of this is, that with the help of these limits, a human user will be able to generate much more precise and better usable data for the robot to learn from.

Abstrakt

Einen Roboter lehren, alltägliche Haushalts Tätigkeiten zu verrichten, während man selbst erfährt, wie es ist solche Aufgaben als Roboter zu bearbeiten. Diese Thesis schlägt die automatische Generierung des Animations-Skelets eines Roboters vor, und die Übertragung dessen in eine Virtuale Realität, in der ein Mensch alltägliche Haushalt Tätigkeiten als Roboter verrichten kann. Um die Bewegung des virtuellen Roboters so realistisch wie möglich zu gestalten, wurde der gleiche inverse Kinematik-Solver für den virtuellen Roboter implementiert, wie ihn auch der reale Roboter verwendet. Dies garantiert dem menschlichen Benutzer, dass wenn der virtuelle Roboter eine bestimmte Bewegung mit den Armen ausführen kann, dann wird der reale Roboter es genauso tun können. Es werden außerdem einige Anpassungen implementiert, die den Menschlichen Körper dem des Roboters von den Limitierungen her, annähern. Das Ziel dieser Limitierungen ist es, dass ein Mensch viel präzisere und damit bessere Daten generieren kann, um den Roboter anhand dieser daten dann besser lehren zu können.

Contents

Acknowledgement

I would like to thank Prof. Michael Beetz for introducing me to this research area, which has now accompanied me through two theses, and which I hopefully can continue to work on in the Future. I would also like to thank Dr. Rene Weller for giving me advice when I needed some and calling my attention to aspects of this topic I would have probably otherwise disregarded. I would like to thank the Institute of Artificial Intelligence Bremen for helping me to find a way to finish this thesis, even with regards to the current world situation. I would like to thank my advisors, Gayane Kazhoyan for always helping me find a solution, no matter what the problem might be and Andrei Haidu for answering all of my silly questions. I would also like to thank Sabine Veit for her support throughout all the years - it's been a while. And I would like to thank Patrick Mania and Dominic Kastens who are always there to help me debug the most ridiculous pc issues. Last but not least, I'd like to thank my better half, for keeping me sane during dark transform calculation times and my parents, who always did their best to support me. I couldn't have made it without you guys. Thank you!

1 Introduction

1.1 Motivation

Teaching by demonstration is a very intuitive and practical approach, to pass on a skill from one to another and has been also generally applied to robotics in various ways. Within my bachelors thesis[10] I looked into how Virtual Reality data can be used to teach a robot simple pick and place tasks, with a breakfast setup scenario in mind. The setup used there was based on the RobCoG[1] project which consisted of two human hands which could be controlled via motion controllers by the human user and a replica of the kitchen environment of the PR2 robot.[2]. Since there was no body visualized within the Virtual Reality environment, as a human user performing pick and place tasks, one always had to keep in mind and think about how a robot would perform this exact same task. For example, as a human it is very natural to bend the torso forward when reaching for an object. A robot can only rarely do that. The PR2 robot, on whom this research so far has been tested, has no option of bending the torso. He might however be able to compensate for this with the length of his arms. But now we have the reverse problem, that the robots arms are a lot longer than the humans are[20], and a solution must be found on how this could be accounted for. Another issue was that only the positions of the two hand held controllers could be tracked within the Virtual Reality environment, and the position of the camera. There was no way to track where the human feet were placed. The solution in order to obtain a navigation pose for the robot back then was to remove the height component of the camera position and project it onto the floor, to estimate where the user might have stood. Another disparity were the human hands within VR. The robot has grippers with only two fingers instead of the humans five. This drastically changes the way the robot can grasp and interact with things compared to a human.

These and some other issues can be addressed and potentially solved by introducing a robots body into the Virtual Reality system, within which the human user can perform everyday household activities. With such a robots body, the human user would gain better understanding of the robot's capabilities and therefore be able to generate data which is a lot more suitable for the robot. Also, all the positions of the links and joints of the robots body can now be tracked and recorded. The navigation pose for the robots base does not have to be generated based on the camera anymore but can be obtained from the robots skeletal mesh body.

1.2 Hypothesis

How can a PR2 robot model be generated efficiently for an Virtual Reality environment within the Unreal Engine? How can the robots body limitations be implemented to the model and transferred to the human user? Will the human user within such a virtual robots body be able to generate more suitable data for the robot? The assumption is that scripts could be implemented and used to auto generate a robot model suitable for the Unreal Engine based on the Unified Robot Description Language. It is assumed that an already existing inverse kinematics solver within the Unreal Engine can be used to calculate the robot bodies movement with respect to the robots capabilities and limits.

[1]RobCog: `http://robcog.org/`
[2]RobCoG: `http://robcog.org/` (last accessed: 06.12.2020)

Since the human user will be made more aware of the robots capabilities, the assumption is made that the so generated data will perform better at teaching robots to perform everyday activities than the previous setup[11] without any virtual body.

1.3 Contribution

The contribution of this thesis will be a skeletal mesh model of the PR2 robot, auto-generated from an URDF-file, fully human controllable within a Virtual Reality environment. The movement of the robots arms will use an inverse kinematics solution, so that the robots movement can be kept as realistic as possible. Some solutions will be explored and experimented with, which concern the mapping of the PR2s body to the human body. The goal there is to prevent the human user from performing movements which the robot will not be able to replicate. As a continuation of previous work[10][11], some evaluation of this newly presented approach should be performed, so that it is comparable to the previous work.

1.4 Structure of this Thesis

This thesis will introduce the approach of autogenerating a skeletal mesh for a robot and introducing it into a Virtual Reality environment in order to collect data on how to perform everyday household activities as a robot. To make this as realistic as possible, the same inverse kinematics solution will be used on the virtual robot, as is used on the real one. Furthermore, some limits will be introduced in order to limit the human user to the movement capabilities of the robot.

This thesis is structured as follows:

Ch.2 Related Work: will present similar work done in this field and will also discuss how this approach is different from the already existing ones.

Ch.3 Foundations: within this chapter all tools and programs will be briefly introduced which play a role and are used within this thesis.

Ch.4 Approach and Implementation: in this chapter all the implementation work will be described.

Ch.5 Experimental Evaluation: will contain the evaluation and an overview over the performance of the resulting product.

Ch.6 Conclusion: after giving a brief summary of the work done within this thesis, some of the problems which have been encountered will be discussed, as well as how this research can be continued.

2 Related Work

Having robots which can help us in every day life by performing household activities is probably one of the oldest goals the scientific field of robotics has. It seems to be a very useful goal for many and has been the subject of countless research projects already. The lack of robots performing every day activities in our homes however, proves that it is a rather difficult task to achieve. Many home environments while fulfilling the same functions, look very different. For example, a kitchen will almost guaranteed have a stove, a dishwasher, a fridge and a dinner table, but the positions of these items, how they look like and how they are operated can vary greatly. A fridge can look like any other cupboard in the kitchen, the stove can be electric or gas operated, which would require different kind of handling, the dinner table might be sometimes located in a different room at all. Not to mention that different people store their kitchen items in the most various locations. People will also have different ways of setting up a breakfast table. With all this variations, it is very difficult to develop a robot, which would be able to perform in all of these different environments reliably and which will find all the necessary items on the first try for e.g. a breakfast. A solution for this problem, also a rather old and fundamental idea in its core, is to simply show a robot where the items are, and how one would like the breakfast table to be setup. After all, this is how we, as humans, teach each other also. So why not teach the robots the same way? The difficulty here lies also in how we would teach the robot. Many options have been already looked into by different researchers and will be described further in this chapter. One could teach a robot a new movement or action via kinesthetic learning, teleoperation, simple visual observation, or since the recent development of technology allows it, via Virtual Reality. This very last approach allows the robot to gain absolute data about its environment, since in a Virtual World, every cupboard's position, color, contents and much more can be easily recorded and passed on to the robot as data to learn from. It is also possible to change the environment fairly easily, allowing the collection of data across many different environments without much work. There are several projects which have looked into this approach and many questions and aspects associated with it, since while ”learning from VR” sounds like a simple idea, it can be done in very many different ways.

The closest related work to this thesis is its predecessor, *”Towards robots executing observed manipulation activities of humans”*[3] and the resulting paper *”Learning Motion Parameterizations of Mobile Pick and Place Actions from Observing Humans in Virtual Environments*[4]. In this previous work, it was looked into if data acquired from Virtual Reality of every day household activities like setting up a breakfast table, can be used to teach a robot to perform the same task. In the Bachelors-thesis, the necessary chain of frameworks was developed, from collecting the data in VR, to loading it into a Knowledge base[7] and using it within the CRAM[14] planning tool to execute the household task within simulation[13]. In the following work[11] this approach was developed further by generalizing the collected data, making it independent of the exact environment it was collected from. More data was collected within different virtual kitchen environments to be able to further generalize. It was also extensively tested, with two different robots

[3][10]: Alina Hawkin. “Towards robots executing observed manipulation activities of humans”. Bachelor Thesis. Institute of Artificial Intelligence, University of Bremen. (Visited on 04/23/2018)

[4][11]: Gayane Kazhoyan et al. “Learning Motion Parameterizations of Mobile Pick and Place Actions from Observing Humans in Virtual Environments”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020

within the bullet world simulation[13] and in the real world on the PR2 robot. The success rate of the robot being able to perform the simple breakfast table setup based on this data was very high, even though there were not that many positions to reference from. For example, with the previous approach, only the hands of the human user can be tracked and the headset. There are no feet positions recorded of where the human user stood while performing a task, so that a navigation position had to be estimated based on the location of the head mounted display. The grasping of objects with the human hands is very different than what the robot can do with his two fingers, leading to impossible or rather hard grasping configurations. There are no visual ques for the human user as to how big the robots base actually is, which leads to the human user standing very close to furniture. If the robot were to use that pose just as is, he would collide with the furniture, which is why some offsets needed to be applied to the gathered poses.

In this thesis, keeping the previous work in mind, some of the issues described above should be resolved. By generating a robot body for the Virtual Reality environment, it will be possible to fully track that body, including the positions of the base and all links and joints. Better data could be generated simply by the human user being more aware of the robots limits.

The *VirtualHome*[15][5]-project created a data-set, which contains descriptions of everyday household activities in natural language and in the form of so called *programs* which are symbolic representations of these activities, defining every step in sequence, which is needed to perform the said activity. This data-set was aimed to be used by robots, in order to be able to perform these everyday activities and various household environments. The *VirtualHome* itself, and the name giver to this project, is a 3D simulator within the Unity[6] game engine used to simulate these household activities. An virtual agent within this simulation was successfully used to perform these activities based on the previously generated programs. Furthermore, by placing multiple cameras within the virtual environment, it was possible to generate a video data-set of the performance of the everyday activities, which again can be used for further learning by robots. Based on this data the *VirtualHome* project team was able to show that an agent within the simulation was able to perform a household task given only a natural language description of it and a model, allowing the agent to learn from the provided programs and videos. Based on this work, these *programs* were developed further in the authors following work[7] now also using *Activity Sketches*, which describe the central essence of an activity, which then can be used with given environmental constrains to generate a program, specifically designed to perform the given activity within the limits and capabilities the given environment provides.

This project is similar to this thesis' idea in the sense that both projects work towards creating a data-set a robot can learn from, however the techniques applied and the desired results are different. While the *VirtualHome*[15] uses natural language descriptions obtained from crowd sourcing as the basis if their task descriptions and their *program*

[5][15]: Xavier Puig et al. *VirtualHome: Simulating Household Activities via Programs.* 2018. arXiv: `1806.07011 [cs.CV]`

[6]Unity game engine: `https://unity.com/`

[7][12]: Yuan-Hong Liao et al. "Synthesizing Environment-Aware Activities via Activity Sketches". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).* 2019

generation, this thesis aims more towards generating a data-set based on the general positions of objects within the household environment and the generalization that can be performed based on these. The task description or the so called *programs*, are assumed to already be there in the form of *CRAM-plans* (which will be described in the Foundations section). Also the focus of this thesis for the data generation aspect is to replicate the robots body in Virtual Reality so that the in the future obtained positions are more suitable for the robot, since the human and robot body both differ largely in size and capability. The use of VR also compared to the video data set of the *VirtualHome*-project would allow for absolute knowledge about the environment, without having to deal with the problem of occlusion of the task by objects or the actor himself.

The *ROS Reality*[8][9] framework provides a connection between the ROS[10] framework and any Virtual Reality hardware supported by Unity[11]. It was developed and used to teleoperate a robot with the help of VR, based on kinetetic teaching. This means that instead of being inside the robots body, like presented in this thesis, the human user is outside the robots body, but is able to guide the robot by grasping the joints of the VR Robot and moving them manually into a goal position, like one would do with kenestatic learning. It was further made possible to set an end-effector target goal with the VR controller, send it via ROS to the robots own inverse kinematics solver, and visualize the result to the human operator based on if a solution for the target pose was found or not. The *ROS Reality* project also included a URDF parser, which auto-generates the robot within the Unity game engine by assembling simple game objects and connecting them with joints, as described in the URDF.

In this thesis a model of the PR2 robot will also be generated for the use within the Unreal Engine (which will be introduced in the upcoming Foundations chapter), but instead of just being assembled with game objects, it will be a skeletal mesh, which allows the use of the Unreal Engine build in kinematics and animation tools. The approach of kinestetic teaching provides the benefit that the robots links and joints can be posed very precisely. However, this way of moving the robot is a lot slower than just controlling the robots manipulators directly. Also, this thesis' approach differs from the *ROS Reality* one also in the fact that teleoperaton is not the end goal. While it probably would be possible to integrate it also, it might be done so in future work but not in this one. Also while sending one goal pose to the robots inverse kinematics solver and waiting for the robot to reach that pose is an intuitive way to teleoperate the robot, the way it currently seems to be implemented is that only one goal is being sent at a time, and then the human operator needs to wait for the computed result and the visualization. It is not performed continuously in real time, in the sense that the robot does not track the movement of the VR motion controller continuously using inverse kinematics, which is the goal of this thesis.

A very close field to Virtual Reality is of course Augmented Reality, which has been used

[8][19]: D. Whitney et al. "ROS Reality: A Virtual Reality Framework Using Consumer-Grade Hardware for ROS-Enabled Robots". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 1–9. DOI: `10.1109/IROS.2018.8593513`

[9][16]: Eric Rosen et al. "Testing Robot Teleoperation using a Virtual Reality Interface with ROS Reality". In: Mar. 2018

[10][5]: Open Source Robotics Foundation. *Bullet world demonstration.* URL: `http://cram-system.org/tutorials/intermediate/bullet_world` (visited on 04/09/2018)

[11]Unity game engine: `https://unity.com/`

in conjunction with Virtual Reality[4][12] to enhance the embodiment element teleoperation of a robot. While oftentimes in teleoperation the output of the 3D camera of a robot is used as the main source of observation of what the robot is currently doing by simply being mapped onto the headset of the human teleoperator, often times this visual representation is not enough and can lead to performance errors being caused by the end effector moving outside the field of view of the camera, or the low resolution of the 3D camera. The paper[4] describes how by the use of AR the immersion aspect can be further increased by providing additional information to the teleoperator, in form of a 3D model representing the robots position even if it moves out of the field of view of the camera, or by providing other visual ques like visualizing the amount by which the gripper is closed with bars or by visualizing paths for the end effector to the target the robot should manipulate.

While the above mentioned work is once again targeting the problem of teleoperation, some of the findings are very interesting and could be applied or developed within the future work of this thesis. For example, the paper[4] states that the users of this system reported an increased sense of embodiment and that the learning curve of teleoperating the robot was significantly reduced. It remains to be seen to what degree these kinds of visualizations can be applied to this project.

Visualizing parts of the robot within Virtual Reality in order to obtain better pick and place data has also been a point of research in Zihe Xu's masters thesis.[13]. Instead of adding the full robot body to the Virtual Reality, only some elements were added. The overall goal was to make the human user aware of the limitations of the robot. The field of view of the human via the VR headset was adapted to the same range and size as the robots. A robot usually perceives one image, analyses it and then performs the task based on that data a trigger based visualization was implemented also. A square robot base is attached to the humans position within Virtual Reality so that the human user is made aware of the size of the robots base and therefore can avoid collisions between the robot and the furniture. A visual que in the form of an arrow is also added to notify the human user before a collision can occur. The potential collision object is also highlighted visually. In order to compensate for the disparity in length of human vs. robots arms, Zihe measured the arms of the human user and added an offset, so that the outstretched length of the human arms would match the ones of the robot. Another problem between the two bodies is that the human user can bend forward in order to gain more arm reachability, while the robot cannot. In order to limit this, an approach with an additional tracker mounted on the chest of the user was tried, also providing a visual que if a bending motion was performed. However the sensor proved to be rather unreliable and often times got occluded by the humans arms, which influenced the obtained data negatively. The grippers used here were not the PR2s grippers, even though the limitations applied to the perception and interaction between the human user and the virtual reality environment were largely PR2 inspired, other, more simple parallel grippers were used instead.

Zihes work was very largely focused on the perception side. Both, in the ways of how limitations to the field of view were applied as well as how visual ques can be implemented to make the human user more aware of the robots limits. This thesis however will focus

[12][4]: F. Brizzi et al. “Effects of Augmented Reality on the Performance of Teleoperated Industrial Assembly Tasks in a Robotic Embodiment”. In: *IEEE Transactions on Human-Machine Systems* 48.2 (2018), pp. 197–206. DOI: 10.1109/THMS.2017.2782490

[13][20]: Zihe Xu. “Designing Human-controlled Robots in VR for Learning Everyday Manipulation Tasks”. Master Thesis. Institute of Artificial Intelligence, University of Bremen. (Visited on 12/10/2019)

more on replicating the PR2s body completely within the Virtual Reality environment. Both theses have in common that the role model for the applied limitations is the PR2 robot and the body differences between robot and human will be addressed in both also, but different ways, presenting different solutions to the same initial problem.

3 Foundations

The following section will introduce all the tools, frameworks, plugins which have been used within this thesis. Some of the tools and plugins have been adapted or changed in the process. All changes will be further described in the **Implementation**4 section. All software which has been adapted, is open-source. The order of mention within this chapter corresponds to order of use and implementation.

3.1 ROS - Robot Operating System

The Robot Operating System[14](ROS) is a framework which provides many tools and libraries necessary to develop robot software. It supports the most commonly used programming languages, allowing to implement the necessary algorithms with the for them most efficient language. This is made possible by a standardized communication protocol which needs to implemented within the nodes which need to communicate via ROS. The entire system is similar in concept to building blocks, where researchers all over the world can share their solutions to robotic problems. ROS helped the preexisting problem, in which every robotics research lab or institute had to develop and implement essentially the same algorithms over and over again, adapting them to their robot, since there was no network or system which would be able to share these commonly used algorithms in a generic way. With the modular design that ROS provides, one can easily download many tools and various implementations, adapt a few parameters if at all, and run them without much hassle. This modular design helped bring robotics research forward across the globe.

3.1.1 URDF

The Unified Robot Description Format(URDF)[15] is a widespread commonly used XML based format, containing robot descriptions. It is also used by ROS as a standard within the ROS community. It essentially describes all the robots links and joints in relation to one another, specifying their respective limits, forces and sizes. Based on this format, the Virtual Reality PR2 model will be generated.

3.2 Blender

Blender[16] is an open-source 3D-modeling software, which provides many tools centered around 3D computer graphics. Beside 3D-model creation, it can be used for animation, rigging, skinning, visual effects generation, texturing, particle, fluid and smoke simulation, sculpting, rendering, even video editing and many more. It is a very versatile tool which can always be extended by many available plugins and python scripts. It includes a python console which allows for element inspection or direct, on-the-go scripting. It is the most commonly used open-source 3D editing tool. In this thesis, Blender was used to generate the PR2s Virtual Reality model.

[14]ROS: Robot Operating System `https://www.ros.org/` (last accessed: 06.12.2020)
[15]URDF: `http://wiki.ros.org/urdf` (last accessed: 06.12.2020)
[16]Blender: `https://www.blender.org/` (last accessed: 06.12.2020)

3.2.1 Skeletal Meshes

Skeletal Meshes are 3D animated objects which contain a skeleton, which allows the mesh fitted around it to be moved and animated. The intuitive idea of a human skeleton within a human body describes the core principle quite accurately. An animation skeleton consists of bones. Each bone within blender has a head and a tail, which are described by 3D-coordinates. Bones can be connected to one another, but they don't have to be. These connections, if they exist, represent essentially the joints of the skeleton, while the bone body itself can be seen as the link. Bones within skeletons are usually surrounded by one or multiple meshes. The movement of the bones, moves the meshes around them or deforms them, depending on the influence radius given to the bone. By programming bone-movement, be it via key-frames and interpolation or forward/inverse kinematics, the entire mesh can be animated and brought to life.

Constraints can be applied to automate the movement of a skeletal mesh or restrict it. For example, certain motion-limits between bones can be set up, or inverse-kinematics chains can be created for pieces of the model, which allow for a more automatic and environment responsive animations.

3.2.2 phobos-Plugin

Phobos[17] is a plugin for Blender, which has been developed by the DFKI (Deutsches Forschungszentrum für Künstliche Intelligenz, Bremen)[18], to assist in the creation of 3D-robot-models for simulators like Gazebo[19] or Mars[20]. It can read in a *urdf* or a *sdf* file, and generate a model out of it.

3.3 Unreal Engine

The Unreal Engine[21] is a games development engine created by Epic Games, which contains very many versatile tools needed to create all sorts of digital games. It allows for physics based and particle based animation, basic skeletal control animation, 3D world map creation and the development of game logic. It is one of the most popular game engines at the moment and is fairly well known for its stunning graphics. Game and animation logic can be programmed with the help of Blueprints, which is a visual way of programming which consists of connecting the correct nodes to one another following an execution path. Alternatively programming can be done with C++. The Blueprint coding style is more beginner friendly, but is of course more limited than the C++ approach, which is more powerful but which also contains many engine-specific language caveats and generally benefits from experience with C++ itself. Both languages can be used for either purpose, although it is strongly encouraged to use Blueprints for animation and it is made nearly impossible to not do so. Unreal Engines functionality can be further

[17][17]: Kai von Szadkowski and Simon Reichel. "Phobos: A tool for creating complex robot models". In: *Journal of Open Source Software* 5.45 (2020), p. 1326. DOI: 10.21105/joss.01326. URL: https://doi.org/10.21105/joss.01326

[18]DFKI Bremen: https://robotik.dfki-bremen.de/en/startpage.html (last accessed: 06.12.2020)

[19]Gazebo Simulator: http://gazebosim.org/

[20]MARS Simulator: https://robotik.dfki-bremen.de/de/forschung/softwaretools/mars.html (last accessed: 06.12.2020)

[21]Unreal Engine: https://www.unrealengine.com/ (last access: 06.12.2020)

increased by the use of Plug-ins, which can introduce new features to the game itself or the development process.

In this thesis the Unreal Engine will be used to setup and control the Virtual Reality environment in which the virtual robot is supposed to be manipulated and controlled by the human.

3.4 RobCog - Robot Commonsense Games

The Robot Commonsense Games (RobCoG)[22] [23] project is largely the baseline for this thesis in Unreal Engine. Its objective is to collect commonsense knowledge for robots, based on everyday activities human users can perform within a Virtual Reality environment. The information about the performed tasks, e.g. positions of objects interacted with, how they were interacted with etc. can be logged symbolically. Within the RobCoG environment, the virtual world with which a human user interacts is the same kitchen environment which can be found in the laboratory of the Institute of Artificial intelligence Bremen.[24] The Head Mounted Display or the HTC Vive[25] Headset represents the head of the human user. It has no visual representation in VR at all and is mapped to the camera which defines the field of view for the human user. The user interacts with this world via two HTC Vive VR motion controllers, which in the Virtual Reality are represented as two human hands. So if one were to look in a mirror within VR, one would only see a pair of floating hands. The hands are not attached to any sort of body and are directly mapped to the motion controllers. There are no options or features currently build in to offset the hands from the motion controller position, at least to my knowledge, since that is not desired in this case nor necessary. The hands can be open or closed. They have physics bodies setup within the Unreal Engine for them, meaning that if they collide with another body in VR, they will not go through that body, but rather be pushed away by it. This means also that it is possible to interact with the environment without needing to grasp an item. For example, by hooking the open hand behind a handle, one would still be able to pull open a drawer. Attached to the motion controllers are also two red arrows, one for each hand. These represent the location of the motion controllers independently of the physics. For example, if one pushes with the hand in VR against a wall, the hand will not go through the wall, but the motion controller in the real world will, since there is no way of preventing this from happening. To visualize this for the user, the red arrow will always follow the motion controller, even if it means going through the wall. The mesh of the hand will be left behind, until the user is back in the open space where the mesh can attach itself back to the motion controller location.

For grasping physics meant also that while closing the hand by pulling the trigger of the motion controller, the fingers would collide with the object the user is grasping and therefore not pass through the object visually, providing a more realistic grasping experience. However, the fingers always perform only one grasping motion. So while it mostly looks realistic, for many objects it also looks a bit off. For example, since

[22]RobCoG: `http://robcog.org/` (last accessed: 06.12.2020)
[23][8]: Andrei Haidu. *Robot Commonsense Games.* URL: `http://www.robcog.org/games.html` (visited on 04/08/2018)
[24]IAI Bremen: `https://ai.uni-bremen.de/` (last accessed: 06.12.2020)
[25]HTC Vive: `https://www.vive.com/eu/` (last accessed: 06.12.2020)

the grasping motion consists of the fingers just coming together to the middle of the palm, one cannot perform a very controlled parallel pincer-grasp. This is mostly relevant for grasping the spoon from a flat surface. Also the grasping works by attaching an object to the hands palm-area, since a sphere is defined there with which an object can overlap and then be attached to. When grasping the spoon, since this point is relatively close to the palm of the hand, one has to really push the hand against the surface on which the spoon is located, fingers outward, so that one can grasp with the palm. This is very unrealistic but then again, spoons are very small and thin objects compared to a bowl and cup. They are really currently hard to grasp in VR with this approach.

3.5 KDL and Eigen

KDL[26] is a third party library for C++ or Python, developed by the Open Robot Control Software (Orocos)[27], providing forward and inverse kinematic solutions. It reduces the motion modeling problem to essentially a geometric one. It is widely used within robotics and is also implemented within CRAM[28]

3.6 USemLog - Logging Data in Virtual Reality

The USemLog[29][30] plugin for Unreal Engine allows to record all actions performed within the Virtual Reality environment. Each object can be tracked, with its position, size and other properties as well as any interaction an object might have with another. Hence, if an object e.g. a bowl is standing on a surface e.g. a table, it can be recorded that they both share a *contact* event. Then once the bowl is picked up by a user, it no longer is sharing this contact event but is rather being *acted on* by a user. All this sub-symbolic and symbolic information can be recorded in .owl and .json files, which then can be used by KnowRob (explained below), to infer this information and to base a robot's decision making onto these previous experiences.

3.7 CRAM - Cognitive Robot Abstract Machine

CRAM[31] is an extensive framework for developing robotics applications. It contains reasoning mechanisms, a lightweight simulator(bullet world[13]) and many more tools, which support the development of cognition-enabled control programs for robots. It defines high level symbolic plans for manipulation activities which can be applied to many robots. It supports already various robots and can be used for any robot which supports ROS. In previous work, the *Cram-Knowrob-VR*[32] package was developed, which provides

[26]KDL: `https://orocos.org/kdl.html` (last accessed: 06.12.2020)

[27]Orocos: `https://docs.orocos.org/` (last accessed: 06.12.2020)

[28]CRAM KDL implementation: `https://github.com/cram2/cram/blob/master/cram_pr2/cram_pr2_low_level/src/kinematics-trajectory.lisp`

[29]USemLog: `https://github.com/robcog-iai/USemLog` (last accessed: 06.12.2020)

[30][9]: Andrei Haidu and Michael Beetz. *Automated Models of Human Everyday Activity based on Game and Virtual Reality Technology.* (Visited on 04/01/2018)

[31][14]: Lorenz Mösenlechner. "The Cognitive Robot Abstract Machine". Dissertation. München: Technische Universität München, 2016, `http://cram-system.org/`

[32]CRAM KVR package: `https://github.com/cram2/cram/tree/boxy-melodic/cram_knowrob/cram_knowrob_vr`

KnowRob queries and generalization mechanisms for processing data collected in Virtual Reality. It also contains robot plans and evaluation tools for this type of data. In this thesis it will be used to evaluate the newly gained PR2-body-based data.

3.8 KnowRob

KnowRob[33][18][3] is a knowledge base for robots which aims to assist robots in performing everyday household activities by providing them with common knowledge. KnowRob works mainly based with prolog-queries to provide answers to a robots questions. It can contain knowledge about the current environment, e.g. where is a spoon typically located in a kitchen? What can be done with a cooking pot? How should a cooking pot be handled, once it has been on the stove? All these questions and more is common knowledge for humans but not for robots. Even more important is the generation of this knowledge. It can be provided via experiences generated by robots while performing every day tasks. Or knowledge can also be provided by learning from humans. In a Virtual Reality environment, the full state of the world is known and can be logged in its entirety, giving even more importance to this kind of data. Even though a new version of KnowRob has been released[3], in this thesis, still an old version will be used, in the hopes of avoiding any issues later down the line while testing and evaluating the newly acquired data.

[33]KnowRob: `http://www.knowrob.org/`

4 Approach and Implementation

4.1 Architecture Overview

The following section will describe the implementation work performed within this thesis. It is structured based on the main tools used. First it will be described how Blender and phobos were used to generate a skeletal mesh of the PR2 robot. After this, the Unreal Engine will be introduced and how some basic functions of the PR2 robot were implemented. Then the various implementations of inverse kinematics nodes of Unreal Engine will be discussed, since they have been tested and experimented with in quite some depth. Also the resulting inverse kinematics solution will be presented. Then it only remains to describe which other features got implemented within Unreal Engine to help and get the virtual robot to move as realistic as possible and how the differences between the human and robot bodies were addressed. The last subsection will then discuss how the data can be recorded within Virtual Reality and what needed to be done to achive that.

4.2 Implementation

4.2.1 Blender and Phobos: generating a skeletal mesh from a URDF file

The following section will provide an overview of the implementation, changes and settings to the Phobos-Plugin and Blender, which were necessary in order to generate a rigged skeletal mesh model of the PR2-Robot based on the URDF file. It will also include details about the import of the URDF file, as well as the resulting FBX export and the settings necessary for the Unreal Engine import.

4.2.1.1 preparation of the URDF-File

The URDF-Format has clear rules on how the file should be set up in order to describe the robot properly, but there does not exist a standard which would enforce the use of the same XML-Tags throughout the file. This results into different Tags being used for the same things within one file.

The Phobos plugin however, does not support all of these different tags, which leads to errors during the import process of the URDF-file. In order to prevent these errors, the problematic tags have to be removed or replaced with parse-able ones. One example of such parsing errors if the references ot the meshes within the URDF files. These references use the ROS notation, meaning they refer to the file path based on the package. However, phobos and Blender do not have a native ROS integration, so either one needs to install a plugin to reference all these paths accordingly, or one can search-replace the *package://* prefix with the according adress. The easiest way to achieve this was to use the search-replace function of any editor and to try to import the file into Blender via Phobos a few times, to see which tags could not be parsed.

The following table provides an overview of the necessary changes:

The *<gazebo>* references needed to me commented out completely, since otherwise phobos would throw errors on import.

original tag	**phobos required change**
package://	/home/blender/Desktop/pr2_common
<parent>	<parent link =
</parent>	/>
<child>	<child link =
</child>	/>
<axis> <xyz> 0 1 0 </xyz> </axis>	<axis xyz="0 1 0"/>

Table 1: *Summary of the URDF adaptation which needed to be performed in order for phobos to parse the URDF correctly.*

After all affected tags were replaced or removed, it was possible to successfully import the file into Blender.

The original PR2 description file which was used can be found here: `https://github.com/code-iai/iai_pr2/blob/master/iai_pr2_description/robots/pr2_calibrated_with_ft2.xml`. The resulting URDF file, after performing all the adaptations can be found here: `https://github.com/hawkina/phobos/blob/rigging/models/iai_pr2_inorder.urdf`

4.2.1.2 Skeletal Mesh generation with phobos

After importing the URDF file into Blender via phobos, the result looks like the PR2 robot but it is unfortunately build wrong. It is not one coherent skeletal mesh. Instead, every link is its own object, containing a mesh, an armature and a bone. A coherent and usable skeletal mesh however, would have only one armature, containing all the bones with the same parent-child relationships as described in the URDF file. Since this is not the case from the get go, adaptation was needed.
The fastest way to do so seemed to start building the necessary adaptation on top of the phobos plugin, since it already imports the URDF and converts it into a python collection, one might as well use this collection as a starting point. Before a skeleton can be created, it must be understood how skeletal bones work within Blender:

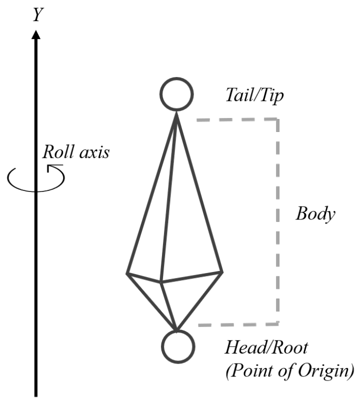


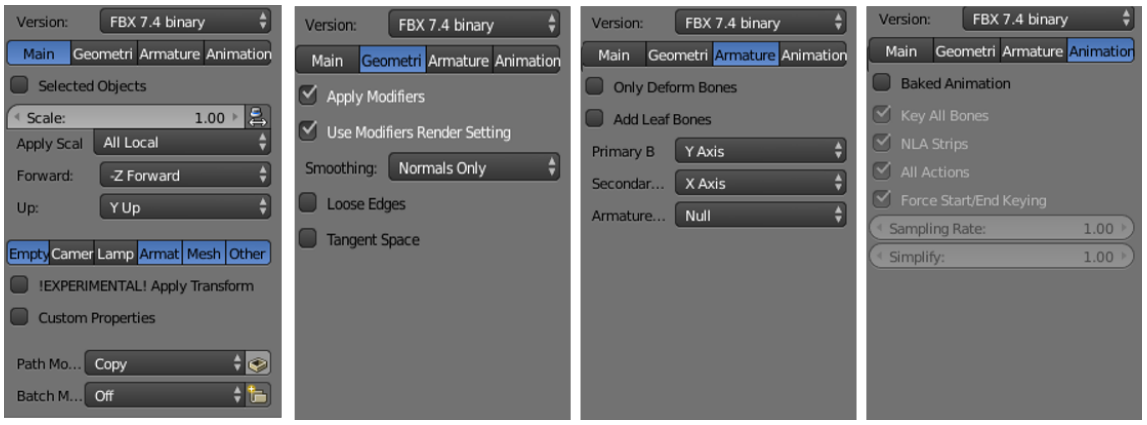
Figure 1: *Schematic drawing of a Blender bone.*

Every bone has a head which represents its point of origin, as well as a bone tail, which represents its end. Bones can be connected to one another but they don't have to. Bones are organized in a hierarchy, via parent-child relationships. A bones child is always connected by its head to the parents tail. Ever bone must have a parent, but not every bone has a child. A childless bone is regarded as a leaf bone. The resulting structure of bones is called an armature or outside of Blender, a skeleton. The armature object within Blender, contains a set of bones which describe the origin or reference position of each bone. The armature has a second set of bones, identical to the first one, which are the pose-able bones. These are used for animation and to define constraints. The armature object also contains all the meshes associated with the bones. In other scenarios, usually the entire body consists of one mesh and the bones therefore define the areas in which the body or in this case the mesh can be deformed, depending on the radius of influence and the given weight of the bone. The connections between the bones represent the joints, just like within a human skeleton. Since the PR2 robot consists of many meshes, and each mesh is associated to a bone, the influence are of the bones can be disregarded, since a bone will always influence just the associated mesh.

Within the scripts of phobos, the main function that was adapted mostly is `createLink` within the `links.py` file. As the name suggests, it is responsible for generating links, which in this case are the bones we wish to create. When this function is called for the first time, it creates an empty scene object, called `pr2_empty`, since it should only be the reference pose. Then the armature object gets created as well as the root bone, which in the case of the PR2 is the `base_footprint`. Since it is the very first bone and since two positions are needed for each bone and the root bone does not have a parent, it gets a fixed length of 0.001, by getting a pose for the tail of $x = 0, y = 0, z = 0.001$. If no length would be given and the bones head ends up exactly where the tail is, the bone would disappear again since Blender does not allow bones of length 0. The entire collection is being iterated over, creating a bone for every child based on the position of the parent. Every bone that would have had the length 0, gets a very tiny length assigned to it. The reason is that the PR2 has a few bones in the same place. For example, the *shoulder_lift_link*, *upper_arm_roll_link* and the *upper_arm_link* all have an origin of 0 and would all result in being non-existant if that rule would not be applied. But since most of the PR2s joints are revolute and move only along one axis, the idea to map them each to one bone seems close. Since the poses are always given relative to the parent, the transforms need to be calculated accordingly.
After all the bones are generated, the meshes are loaded and mapped to the bones. An armature modifier is applied to every mesh, as well as a vertex group needs to be created, for the bone to be properly mapped to the skeleton. A rather PR2 specific rule, is that the meshes for the fingers initially appear at the wrong place. So for example, for the right gripper both fingers will appear where the right finger is supposed to be, one of them will have a little offset. This is due to the fact that it is the same mesh is being used for both of them, so to fix this it needs to be rotated.

4.2.1.3 Export of the FBX Model from Blender to Unreal Engine

Exporting and FBX from Blender to Unreal Engine is a project of its own. Both systems treat FBX files slightly different and finding the correct settings takes some time. The settings which worked in the end are the following:



Version: FBX 7.4 binary
Main | Geometri | Armature | Animation
☐ Selected Objects
Scale: 1.00
Apply Scal: All Local
Forward: -Z Forward
Up: Y Up
Empty | Camer | Lamp | Armat | Mesh | Other
☐ !EXPERIMENTAL! Apply Transform
☐ Custom Properties
Path Mo...: Copy
Batch M...: Off

Version: FBX 7.4 binary
Main | Geometri | Armature | Animation
✓ Apply Modifiers
✓ Use Modifiers Render Setting
Smoothing: Normals Only
☐ Loose Edges
☐ Tangent Space

Version: FBX 7.4 binary
Main | Geometri | Armature | Animation
☐ Only Deform Bones
☐ Add Leaf Bones
Primary B: Y Axis
Secondar...: X Axis
Armature...: Null

Version: FBX 7.4 binary
Main | Geometri | Armature | Animation
☐ Baked Animation
✓ Key All Bones
✓ NLA Strips
✓ All Actions
✓ Force Start/End Keying
Sampling Rate: 1.00
Simplify: 1.00

Figure 2: *Blender FBX export settings.*

Also, since both programs treat the files rather differently there were a few other issues. In the beginning, the export failed because all the meshes had the same names as the bones. They were both referred to as e.g. *base_link* etc. Within Blender that was not an issue, since there is a differentiation between the meshes and the bones within the armature. They are different objects of different classes. However Unreal sees meshes as bones and adds them as such to the Skeletal tree. There is a setting within the FBX import of Unreal Engine which supposedly avoids this but it this case it did nothing. Therefore, for Unreal, there were multiple objects with the same name, which of course could not be imported properly.
After iterating over all the meshes and prefixing them simply with the term *mesh*, this problem was solved. However the skeleton still did not behave correctly. It seemed like the bones which are supposed to control a specific mesh have an offset to one another. Basically the wrong bone is controlling the wrong mesh. After some research, it was found that the Unreal Engine has a very different understanding of bones compared to Blender. In Blender, every bone has a head and tail, therefore two positions which define it. In Unreal Engine, only one position describes the bone. The connection seen between to bones, which looks very similar to the Blender bone body, is simply a visualization of where the parent of the current bone is. The mismatch and weird movement of the bones occurred, because Unreal Engine places its bone on the blenders tail of a bone. The tail was first defined as the position of the next child, and the head being the important origin of a bone, but for Unreal this all needed to be shifted. This results in the skeleton within Blender to look a bit disconnected, but works fine within Unreal Engine.

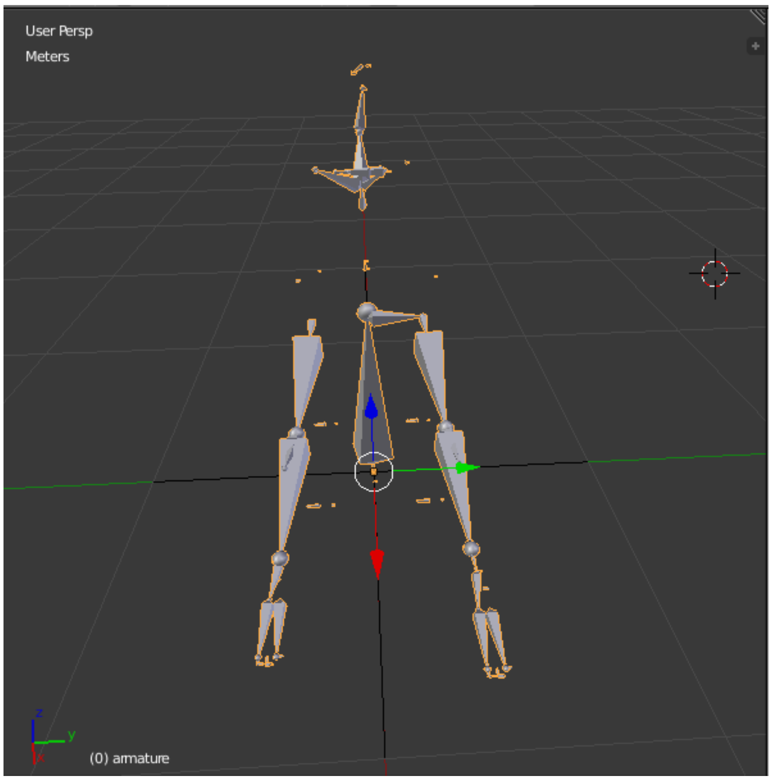


Figure 3: *PR2 skeleton after having all the necessary settings and modifications applied for the Unreal Engine export.*

Another important setting is the *path mode: copy* setting. This copies all textures of the meshes into a folder which will be exported with the FBX. If this is not done, the textures will end up missing in the resulting object within the Unreal Engine.

As far as bone limits and constraints go: they can be applied to the bones fairly easily within Blender. Even IK chains can be setup. This can all be done with the script also, based on the URDF. However, either Blender does not export these constraints properly or they are generally not supported by FBX or Unreal Engine simply ignores them. This is a bit unclear but during my research I had found several forum posts suggesting that this is unfortunately, impossible. It might be related to Unreal Engine not having a definition of general skeletal mesh constraints, outside of the physics asset or animation blueprint. So unfortunately, the code to generate these constraints within Blender exists, but has been commented out again since it has no effect on the model within Unreal Engine.

4.2.2 Unreal Engine: Preparing the Robots model for VR

The following section will describe in more detail what exactly was done to get the robot to move within the virtual reality. How the robots body was mapped to the humans, which difficulties arose and how they were solved. The section is split into different aspects of this process, from the import of the FBX model to inverse kinematics used to move the robot and other features implemented. In the beginning of the development and for many of the features, Blueprints were used. This is due to the fact that they are intuitive to use and allow for very fast prototyping of functionality. For getting to know how the Unreal Engine works in the beginning suing them was totally fine. For the more complicated things later on in the development, C++ was used since the limits of what the Blueprints can do were hit. It also seems like the support for C++ code for general in-game logic is a lot more present than for animation. It seems like Unreal almost forces the use of Blueprints for Animation purposes.

4.2.2.1 Import of the FBX Model

For the import of the FBX, no setting changes need to be done, except setting the tick at *skeletal mesh* in the Unreal Engine import window, and using the import all button. Even though the FBX skeleton might have looked a bit off in Blender, within Unreal Engine it produced the expected result:

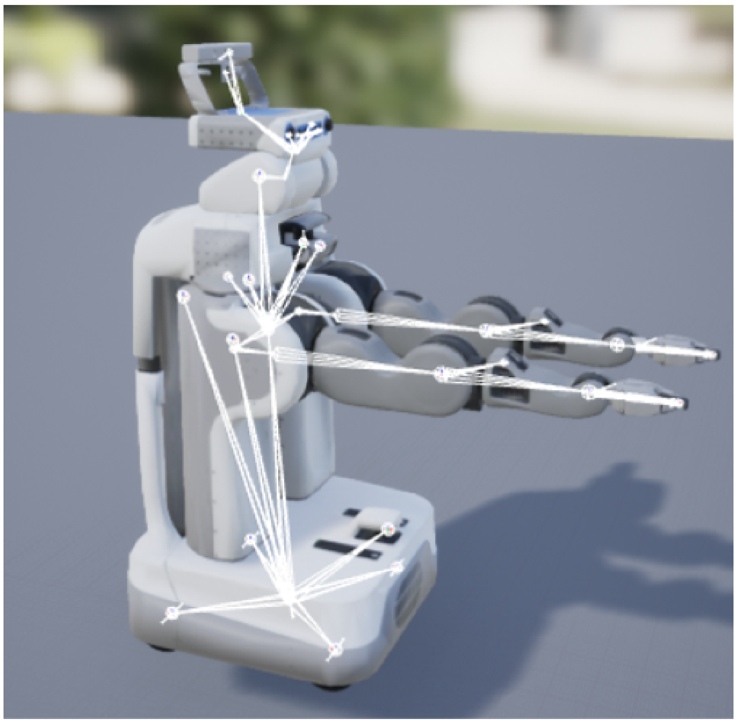


Figure 4: *The imported PR2 skeletal mesh within the Unreal Engine. The tiny spheres are essentially the bones, the long connections between them simply visualize their parent relation between one another.*

It might take a few minutes for the textures to render and be fully displayed, but the above depiction should be the result. Auto-generated at import as individual files are the skeletal mesh, the skeleton and a physics asset which contains some collision capsules, but not for all the bones. The skeleton really only contains the bones and can be used to inspect how moving a particular bone would influence the skeleton, while the skeletal mesh component is rather focused on the 3D-mesh aspect of the skeleton. So while having similar names, these components do rather different things.

4.2.2.2 Base movement with and without VR

To be able to move the skeleton within the game, a `BP_PR2_Character` was created, which is a character blueprint representing the PR2 within the name. It is linked to the imported mesh of the PR2 and is pretty much responsible for how the character interacts with the Virtual Reality environment. It takes care of essentially everything except the skeletal animation, which is being computed within the *PR2_Anim_BP*, which is an animation Blueprint associated with the character.

One of the very first things that got implemented was the characters movement based on button-press on the keyboard. Even though later on the movement got taken over by VR entirely, for the beginning phase of the project it was very useful to be able to move the PR2 with the keyboard. Within the projects input settings, the `w,a,s,d,q,e`-keys were mapped to input axis of the Engine. E.g. whenever the button `w` would get pressed, a value of $0,5$ gets generated and sent to the according input axis event. Whenever this happens, the forward vector is being obtained based on the position of the PR2 mesh, or later on, the `VRCamera` object. This vector as well as the value of the axis input is then being passed on to the `Add Movement Input` node, to generate the according movement. For general testing with the standard view port of the engine this is completely fine. While when using VR and the VR Previw view port, using this movement can cause motion sickness.

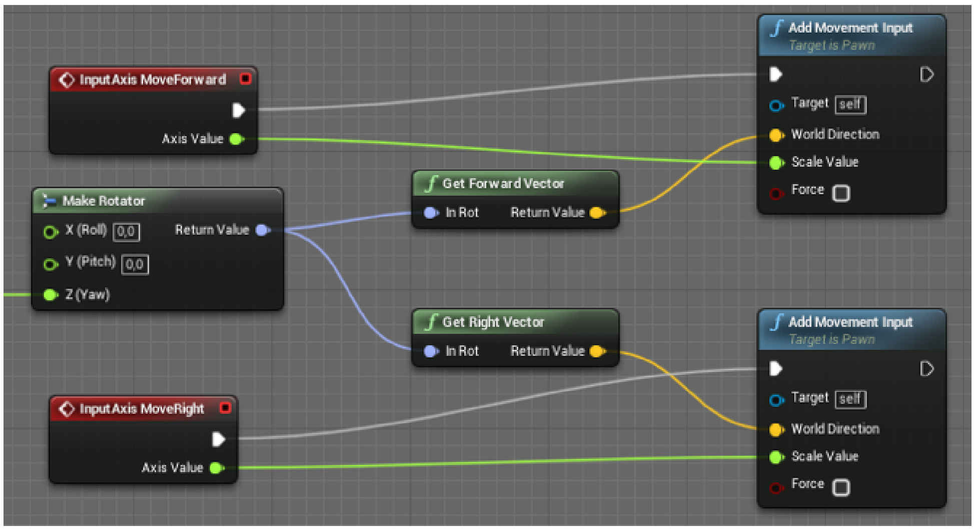


Figure 5: *part of the BP_PR2_Character Blueprint, showing a part of the implementation of key-press based navigation/movement.*

In order to include the HTC Vive headset and motion controllers into the movement capabilities, a few more objects needed to be created and attached to the `BP_PR2_Character`. First, a scene component was added. Scene components are similar to the empty objects in Blender. They are essentially objects which do not have any mesh or other visual component, they are just a point or a coordinate system, that can be placed in space in order to reference a transform in the virtual space. They can also be used to set up parent-child dependencies between the objects. One of these components is called `VR_Origin` and represents the origin point of the VR setup, as the name suggests. It has a position of $x = 0, y = 0, z = 0$ relative to the PR2 mesh, which is its parent. It has three children, two motion controllers, `MC_Right` and `MC_Left` and the `VRCamera`, which is the camera component that gets possessed by the player, or rather the HTC Vive headset and generates the field of view that the human user perceives.

The movement mapping of the headset to the PR2 character might be set up a bit unconventional. It just developed into what it currently is based on testing and based on how some solutions were found and developed over time. If set up completely from scratch, it might result in a completely different solution. But this is currently what is used:

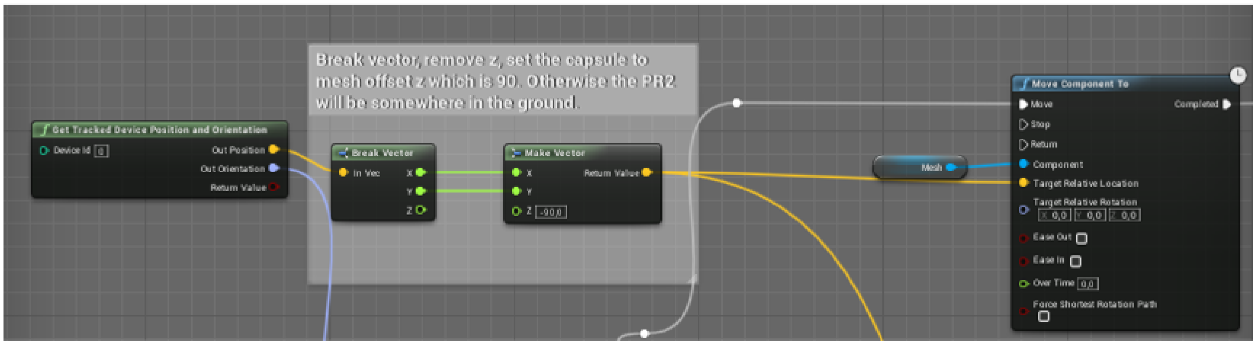


Figure 6: *Navigation based on the position of the Head Mounted Display.*

The position of the Head Mounted Display (HMD) is being obtained in device space. The X, Y components are being extracted, since the Z component which represents the height of the headset does not matter for navigation, it is being ignored and replaced with -90. This value is important, because the `BP_PR2_Character` has a capsule as its root component, and the skeletal mesh if not given an offset, would be located in the middle of that capsule. So in order for the PR2 to not float in the middle of the capsule, he has this $Zoffset$, which of course needs to be maintained throughout navigational position changes.

To correctly rotate the PR2, was a whole other issue. The rotational Z component can be easily obtained, but if it is applied directly to the mesh or the root component, it would cause interesting side effects. Either the robot would rotate but not around the Z axis of itself, but some other point in space, which results in the robot driving around in a circle like a car instead of rotating in place. This would also cause the `VRCamera` not to follow but to just stand in place, leaving the PR2s body and watch him rotate, until after 360° he would be at the original position again. Or, the mesh would rotate correctly around its own axis but the motion controllers would fly away on rotation, since they would rotate around the same origin point the PR2 initially wanted to rotate around

also. So the solution to this problem was to save the Z rotational value in a variable, pass it to the `PR2_Anim_BP` where it is used within the Animation Graph to rotate the PR2s *pr2_empty* bone, which is the origin of the mesh.
As previously mentioned, this solution probably came to be due to an error in the initial setup and how objects are inheriting from one another, but it works.

4.2.3 Skeletal Mesh Animation using Inverse Kinematics

In order for the robots body to be able move its arms, the original idea was to use the inverse kinematics nodes Unreal Engine provides. It would be possible to define an inverse kinematics chain, describing the arm of the robot, e.g. from the *shoulder_pan_link* to the *gripper_palm_link*, setting the gripper_palm_link as the end-effector, and mapping its pose to a VR-motion-controller. The position of the base of the robot would be mapped to the VR-headset, ignoring the z component of the pose since the robot should be positioned on the floor and not mid-air, as is also described in the previous section.

All the Unreal Engine skeletal control nodes can be found within the animation blueprint, which is associated to a skeletal mesh, which in this case is the skeletal mesh of the PR2 robot. This means that the inverse kinematics is animation based only.

Unfortunately, this did prove to be a lot more labor intensive and harder than it sounds. In games development, the focus of the animation is to look good. In our case, the focus was not only to make the animation look a certain way, but it had to behave a certain way. It needs to mimic the robots capabilities as closely as possible, and this involves moving the arms in a way the real robot would be able to do too. This means that each joint has to be restricted in the same way as it would be on the real robot, or at least as close to the real robot as possible. For the PR2 robot this means that the joints have to be limited on a per-axis basis, since each joint has a certain range of motion on one axis only, since most of the joints are revolute, and others are continuous. Without these restrictions, one would be able to move the arms of the robot within VR in ways which would look broken, dislocated and which would be impossible for the real robot to achieve. Since the goal of this thesis is to try and collect better data for the robot to use as a baseline to perform a task, and to limit the human user to the capabilities the robot provides, these limits are crucial. A little difference between the real robot and the animated version within VR can be expected, but the closer the Unreal-Robot is to the real one, the better and more useful the collected data will be.

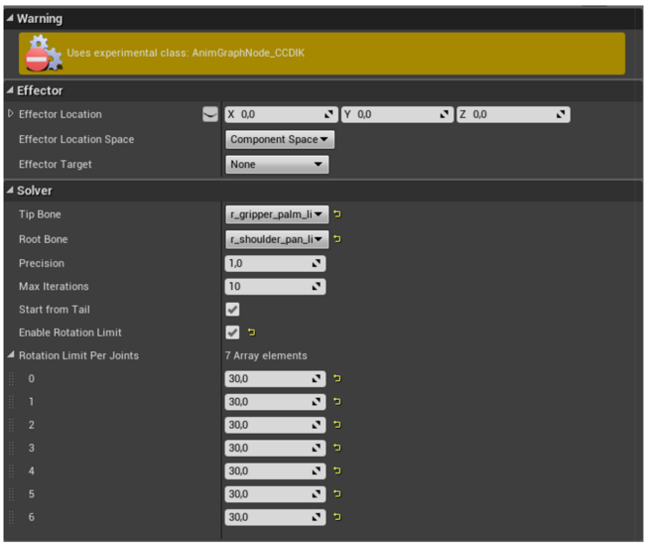
Before we dive into detail of all the different approaches which were tried, here is a tabular overview of them, which nodes they involved and what their main result or issue was, hence why they failed to become the end solution. It mainly boils down to being able to set bone or joint constraints on a per axis basis, to replicate the PR2s movement. It will also mention a few IK nodes which will not be explained further into detail in the upcoming section, where the sentence within the table is enough to explain why they could not be used.

Used node combination	**Result**
CCDIK	IK with only symmetric rotational limits per joint. Cannot apply limit per-axis.
CCDIK + ApplyLimits	Limits are applied after the IK calculation, shifting the end-effector away from the goal.
FABRIK	No option to set limits at all.
FABRIK + ApplyLimits	Limits are applied after the IK calculation, shifting the end-effector away from the goal.
Two Bone IK	PR2's arm has more than two bones in a chain.
Leg IK	Always takes the floor into consideration, leading to very weird configurations. Allows limits, but it's not enough.
CCDIK + ApplyLimits + CCDIK + ApplyLimits	Recalculate IK after moving it into limits. This can be repeated several times. Improves the result but does not reach goal.
FABRIK + ApplyLimits + CCDIK + ApplyLimits	Similar to previous, although slightly better, closer to the goal.
CCDIK + ApplyLimits + elbow goal	Split the IK chain in half, at the elbow. Calculate upper arm and lower arm IK separately. Generates nice elbow behavior, but the limits do not act correctly.
CCDIK + Physics Asset	Generate a physics asset for the PR2, including setting up all joint constraints within physics. Looks good in physics asset simulation, behaves strangely/breaks in the game world on play. Also CCDIK ignores physics limits.
FABRIK + Physics Asset	Similar to the above. Also ignores physics constraints.
CCDIK or FABRIK + Physics Asset + Animation/Physics Blending	Does not reach goal. Similar problem to ApplyLimits that CCDIK and FABRIK bot ignore the physics asset constraints
Virtual Bones + any IK	Summarize upper arm and lower arm bones into one virtual bone each, have only the elbow joint defined. IK ignores virtual bones and does not see them as a chain, even if they are chained.
KDL based PR2 IK Node	Takes limits into account, does not always find a valid configuration but works.

Table 2: *Overview of all the IK approaches tried within this thesis.*

4.2.3.1 CCDIK

The core idea of using an inverse kinematics node is also largely based on the CCDIK (Cyclic Coordinate Descent Inverse Kinematics) node, which is a newly added experimental feature in Unreal Engine, and which on the first glance provides a way to set limits.



Warning

Uses experimental class: AnimGraphNode_CCDIK

Effector

Effector Location	X 0,0 · Y 0,0 · Z 0,0
Effector Location Space	Component Space
Effector Target	None

Solver

Tip Bone	r_gripper_palm_li
Root Bone	r_shoulder_pan_li
Precision	1,0
Max Iterations	10
Start from Tail	✓
Enable Rotation Limit	✓
Rotation Limit Per Joints	7 Array elements
0	30,0
1	30,0
2	30,0
3	30,0
4	30,0
5	30,0
6	30,0

Figure 7: *Settings options for the CCDIK Animation Graph Node*

However, after some research and trial and error, the limits set within the node are all meant to be symmetrical, and not on a per-axis basis. Meaning that this limit just limits the motion across all axes, and not a single one, like would be needed. This leads to impossible configurations, which look broken and wrong, but the goal position can be reached.

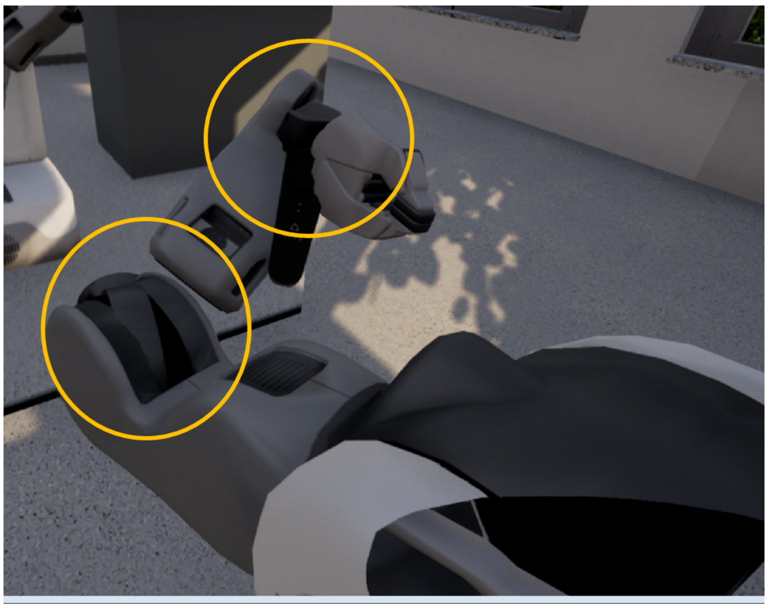


Figure 8: *Some of the disjointed states of the PR2 arm, while using the CCDIK node*

4.2.3.2 CCDIK + Apply Limits

In order to fix this and try to apply some realistic PR2 limits, the idea was to use the ApplyLimits node, which allows to set max and min limits between -180 and +180 degrees, on an per-axis basis. Placing this node after the CCDIK node however, results in the application of these limits after the CCDIK calculations, which moved the arm away from it's goal. Placing this node before the kinematics node has no effect, since the kinematics calculations will not take the limits into account at all. Another closely related idea to this approach was to chain multiple **CCDIK** and **ApplyLimits** nodes together, in the hope that for the second ik node, the input position will be one which was moved closer to the goal already by the previous ik node and had the limits applied to it, so maybe it could move it closer to the goal again and the limits can be applied again to, in a way, try and force it to get closer and closer to the goal. While this slightly improved the result, it didn't solve the issue that with these limits, the goal position could not be reached.

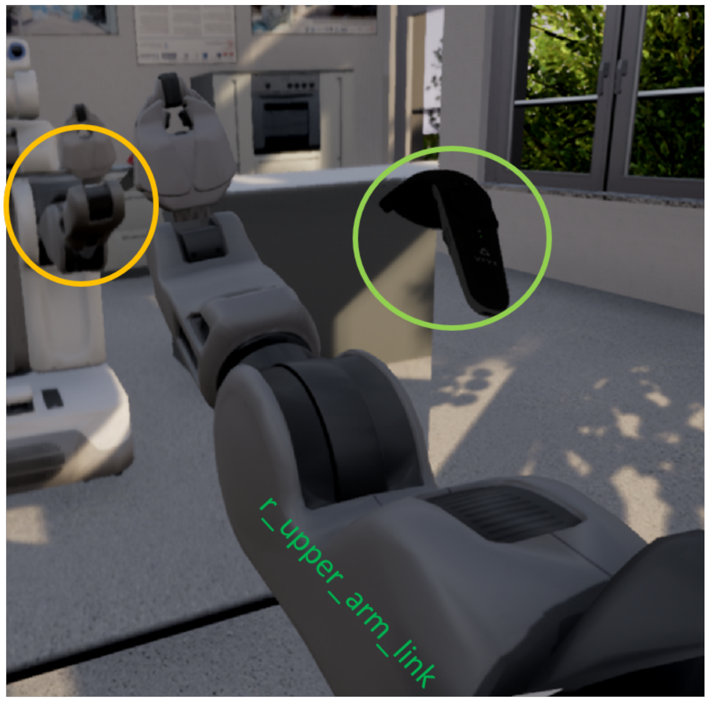


Figure 9: *By rotating the shoulder_pan_link away from the goal (green circle) and allowing the upper_arm_link to rotate at all and bend the elbow, the goal position could be reached. However, CCDIK and the ApplyLimits node do not work well together and cannot reach the target goal.*

Also some of the limit application yielded weird results. For example, the *upper_arm_link* is able to rotate beyond 180 degrees in one direction. This is impossible to be set with the settings. Adding an offset did not seem to shift it and it seemed to actually block the movement if it reached 180 degrees and not allow it to rotate further, even if it should be possible with the given offset or setting the limits to -180 and +180 degrees. There also seems to be a known bug with this node, that the min and max values for X and Z axis seem to be swapped.[34] This meant that for the *shoul-*

[34]ApplyLimits-Node bug report: `https://issues.unrealengine.com/issue/UE-66293` (last accessed: 30.11.2020)

der_pan_link, which movement should be only around the Z axis, it needed to be set in the settings as around the X axis. This just led to overall confusion when setting the limits.

4.2.3.3 FABRIK

Another inverse kinematics node that could be used is the FABRIK node, which is an implementation of the FABRIK algorithm[35] which in the original version of the algorithm does not support joint limits, but in an extension[36] developed a few years later, does. However the Unreal Engine node of this algorithm does not have the ability to set any limits at all, but the idea was to see if it maybe interacts better with the ApplyLimits node. Unfortunately, it did not. It reached the goal in a more straight-forward fashion than CCDIK did, but it distorted the arm even more away from the goal once the limits were applied. Applying limits seemed to work the same way with FABRIK as it did with CCDIK, meaning that they got ignored completely. After some research it was found that generally, the FABRIK algorithm itself supports axis-based limits, but that this feature of the algorithm just simply haven't been implemented yet.[37]

4.2.3.4 Other IK Nodes

Unreal Engine has a few more inverse kinematics nodes, for example Two-Bone-IK, Leg-IK and Spline-IK, but most of them just consider very small IK-chains of just two links and a joint (hence the name Two-Bone-IK), or have very specific uses. For example, the Leg-IK node allows to specify limits, but always considers and aims towards the floor, which defeats the purpose of using it for an arm. Trying to use the Two-Bone-IK while creating Virtual Bones which essentially allow to divide the arm into two parts, the upper and lower part with the elbow as a joint, didn't work that well either.

4.2.3.5 Split CCDIK and ApplyLimits to Parts of the Arm

So if one very long chain is a problem and the joint-limits cannot be set properly, it was tried to split up the chain into two parts for the IK-solvers. This means, that instead of defining the arm in one node, it will be defined with two nodes. The first going from *shoulder_pan_link* to *elbow_flex_link*, and the other would go from *elbow_flex_link* to the *gripper_palm_link*. This approach also means that two goal positions would be needed: one for the end effector, *gripper_palm_link* and one for the elbow *elbow_flex_link*. In order to know where to place the *elbow_flex_link*, a pose was calculated based on the motion controllers current position. Basically, the forward vector of the motion controller is obtained so that it is known where the controller is pointing to, and based on this the

[35] [2]: Andreas Aristidou and Joan Lasenby. "FABRIK: A fast, iterative solver for the Inverse Kinematics problem". In: *Graph. Models* 73.5 (Sept. 2011), pp. 243–260. ISSN: 1524-0703. DOI: `10.1016/j.gmod.2011.05.003`. URL: `http://dx.doi.org/10.1016/j.gmod.2011.05.003`

[36] [1]: Andreas Aristidou, Yiorgos Chrysanthou, and Joan Lasenby. "Extending FABRIK with Model Constraints". In: *Comput. Animat. Virtual Worlds* 27.1 (Jan. 2016), pp. 35–57. ISSN: 1546-4261. DOI: `10.1002/cav.1630`. URL: `https://doi.org/10.1002/cav.1630`

[37] Angle Contraints for FABRIK Node:
`https://forums.unrealengine.com/development-discussion/blueprint-visual-scripting/40344-fabrik-node-doesnt-respect-angle-constraints` (last accessed: 30.11.2020)

motion controller pose is offset backwards by an amount which was estimated with trial and error.

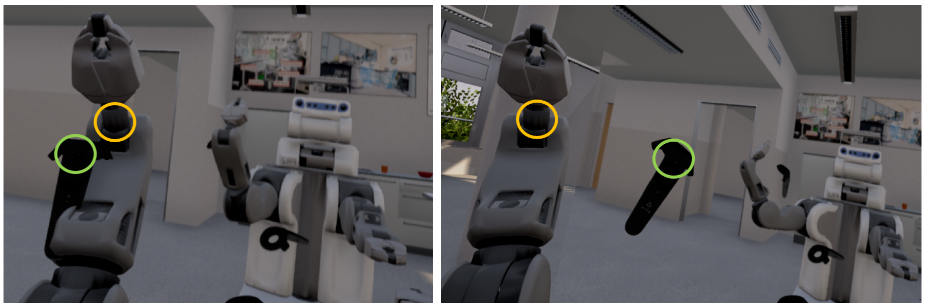


Figure 10: *It can be seen in the left image, that this approach provides a better elbow movement, and reaches the goal in some configurations. However as can be seen in the right image, just a little rotation can shift the entire result by a very large margin.*

This would allow for a nice behavior of the elbow, since now it was actually trying to bend outwards away from the body while the palm link would try to go inwards. The ApplyLimits node could me monitored here a lot easier also, since it was responsible for just a few bones instead of the entire chain, which allowed for more easier debugging. Unfortunately, this does not solve the limit problems entirely either but allowed for a more natural looking animation, at least in regards to the elbows behavior. For some other bones, as for example the *upper_arm_link*, which had issues with the ApplyLimits node as described above, a manual offset can be added using the Transform-Modify-Bone node, which allows to completely freely modify bones.

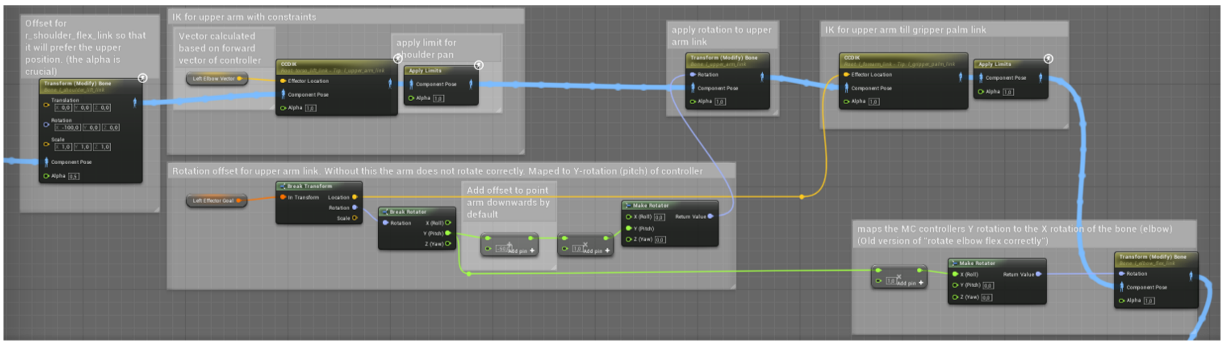

Offset for r_shoulder_flex_link so that it will prefer the upper position. (the alpha is crucial)
Transform (Modify) Bone
Translation
Rotation
Scale
Component Pose
Alpha
IK for upper arm with constraints
Vector calculated based on forward vector of controller
Left Elbow Vector
CCDIK
Effector Location
apply limit for shoulder pan
Apply Limits
apply rotation to upper arm link
IK for upper arm till gripper palm link
Rotation offset for upper arm link. Without this the arm does not rotate correctly. Maped to Y-rotation (pitch) of controller
Left Effector Goal
Break Transform
In Transform
Location
Break Rotator
X (Roll)
Y (Pitch)
Z (Yaw)
Add offset to point arm downwards by default
Add pin
Make Rotator
Return Value
maps the MC controllers Y rotation to the X rotation of the bone (elbow) (Old version of "rotate elbow flex correctly")

Figure 11: *The nodes setup necessary to achieve this motion. It does not need to be viewed in detail it should just visualize how many different nodes are needed to get this sort of interaction, and that even then it is not yet perfect.*

This approach, while the most promising, was very setup-time intensive, contained a lot of calculations to try and calculate parts of it manually, was rather not reliable and really hard to maintain. If no other approach would have been found, as it will be described below, this might have been the end solution. It would have not been perfect, but it would be acceptable, even if some movements would cause the arms to look crazy and disjointed.

4.2.3.6 CCDIK with Physics

While looking for a way to constrain the Unreal Engine IK somehow, it was discovered that joint limits for a skeleton can be set within the physics asset of the skeletal mesh. Each skeleton imported into Unreal Engine, by default, will generate a physics asset which is used to compute collisions, forces and other physics-based events between objects. In order to make collision detection more efficient, primitive shapes like spheres, cubes or capsules are used. Usually a capsule would be created around a bone and the mesh. Then constraints can be set between the shapes. There are no limits as to how many constraints can be set for an individual shape or bone. The auto-generated asset only contained three capsules, one for the torso and one for each arm. This of course, would not be enough. So a capsule for each bone in the arm was created with the according constrains.

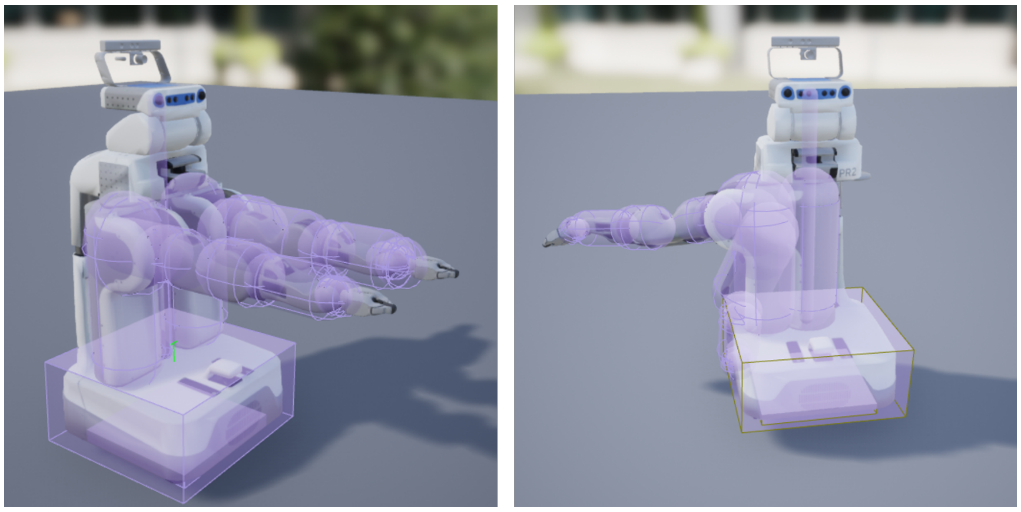


Figure 12: *On the left: the inactive physics asset with all the currently setup capsules defining the constraints. On the right: the physics asset during simulation. The right arm is being positioned by a mouse click.*

Each constraint can limit a joint for translation or rotation alongside an axis. The rotational constraints however are not referred to as XYZ, but rather *Swing 1 Motion, Swing 2 Motion* and *Twist Motion*. *Swing 1* and *Swing 2* each allow to constrain the angle of motion around the XY and XZ pane, which are supposed to be the symmetric angles of a cone, and *Twist* is limiting the symmetric angle of roll along the X-axis. While these limits are still not quite what is needed, since they are symmetric and the PR2's range of motion of the individual joints is not, they were looking quite promising within the physics-asset simulation. The PR2 could essentially be used as a rag-doll there. However, unfortunately the physics did not interact well with the rest of the environment and with the animation blending. Animation blending in this case means that it is possible to set a value alpha, between 0 and 1, which would describe the strength of the two inputs against each other. Meaning, that if the physics blending is set to 1, everything within the Animation Blueprint will be ignored and only physics will be used to move the robot and vice versa. Setting it to 0,5 allowed for a good mix between the two, but often times it seemed like the IK within the Animation Blueprint, was not strong enough to move the arms of the Robot to where they were supposed to be. Motors could be added within the Physics Asset to drive the animation, but the individual values to drive these motors cannot be mapped to the IK result, since the IK result is an Animation Pose and does not provide access to the individually computed values for the individual joints. This also caused occasionally very weird effects at run-time. With some settings, the robot would float in the middle of the room or fly away out of it, hectically move the joints and fall apart entirely, fall through the floor or have one arm fly off. This also didn't tackle the essential problem of not being able to apply the limits to the IK solver, so that they could be respected during computation of the new pose. It was only another way to apply limits after the computation, and as seen before with the ApplyLimits node, this will just shift the end effector out of its goal position. Maybe there is a way to get it to run with the help of the physics asset, and I just lack the experience to do so, but it seemed to simply introduce more problems than providing a good solution.

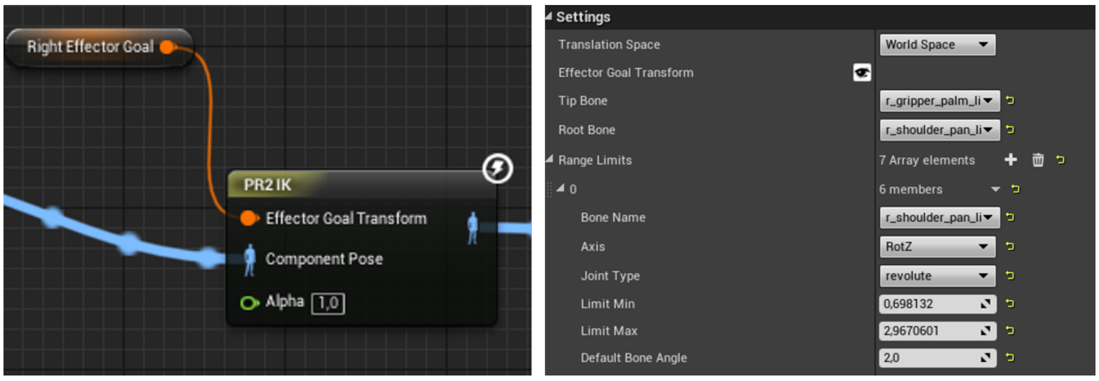
4.2.3.7 KDL

After all these attempts which yielded rather unsatisfying results, it was clear that Unreal Engine at the time of writing this, does not have a good and free inverse kinematics solver which would respect joint limits on an axis basis. There were mentions of other IK plugins found, but they were not open source and not available for free, unfortunately. The best way to get the virtual PR2 robot to move as similar as possible to the real world would be to use the same inverse kinematics solver within the Animation Blueprint as is used on the real robot. There are several options for the PR2 out there, all with their own benefits and downsides, but the best solution seemed to be to use the Orocos-KDL(Kinematics and Dynamics Library)[38] and the Newton-Raphson[39] position IK-Chain solver with joint limits. This solution had four developmental stages:

1. Create a custom Node for the Animation Graph.
2. Import the KDL library into this node.
3. Create a kinematic chain based on the given inputs, including joint limits.
4. Calculate the correct transforms.

Creating a new custom Node for the Animation Graph
In order to be able to use the KDL library for the animation in Unreal Engine, an animation graph node needs to be created, which will be the interface between Unreal Engine and the KDL third party library. After some research, another custom animation graph node was found[40], which was used as a basis. This template also wraps the node already into a plugin structure, which is very useful, since this will not be needed to be implemented later on, in case this should become a public plugin.
Generally, an animation graph node contains two components. The code for the node itself, in this case `AnimNode_PR2IK.h` and `AnimNode_PR2IK.cpp`, and the code which describes how this node appears and behaves like within the Unreal Engine Editor and the Animation Graph: `AnimGraphNode_PR2IK.h` and `AnimGraphNode_PR2IK.h`. The later two don't really do that much so they won't be explained further, but the `AnimNode_PR2IK.cpp` will be explained in more detail below. At this point it simply needs to implement certain functions it is inheriting from and that is that.

[38]KDL: `https://orocos.org/kdl.html`
[39][6]: A. Goldenberg, B. Benhabib, and R. Fenton. "A complete generalized solution to the inverse kinematics of robots". In: *IEEE Journal on Robotics and Automation* 1.1 (1985), pp. 14–20. DOI: `10.1109/JRA.1985.1086995`
[40]Custom Animation Graph Node: `https://github.com/dawnarc/ue4_custom_anim_graph_node`



Settings

Translation Space	World Space
Effector Goal Transform	
Tip Bone	r_gripper_palm_li
Root Bone	r_shoulder_pan_li
Range Limits	7 Array elements
0	6 members
Bone Name	r_shoulder_pan_li
Axis	RotZ
Joint Type	revolute
Limit Min	0,698132
Limit Max	2,9670601
Default Bone Angle	2,0

Figure 13: *On the left: the PR2 IK node within the animation graph. On the right: exemplary settings for the PR IK node*

Importing the KDL library into the Custom Animation Node

This has proven to be more tricky. Not only is KDL needed, but the Eigen[41] library, since KDL has a dependency on it. The first approach was to follow a tutorial[42]. However that did not work initially, since the libraries imported there, were already build while Eigen and KDL were not. The idea that since Unreal Engine uses Microsoft Visual Studio[43] to build everything, and since Visual Studio with a plugin can build CMake files (as explained in this tutorial[44]) it might also be able to build Eigen and KDL, failed unfortunately. Another approach was manually build these libraries by hand and import them into the Plugin. This is not the most elegant solution, and it remains to be seen if there is a more elegant one, but it worked. Some other tweaks were however also needed. For example, the Eigen library contains exclusively header files. However, they do not have a `.h` extension, so Unreal could not find them. The solution was to add the extension there manually. This unfortunately also needed to be done with all the includes where KDL references Eigen or Eigen references itself. They also needed to be specified with the correct extension. After this was done, it was finally possible to use KDL within the animation graph node.

Creating a Kinematic Chain based on the Input received

The `PR2_IK` node within the animation graph needs to get the two end-points of an IK chain to be defined. These can be selected in the settings of the node, in the `Tip Bone` and `Root Bone`. For the right arm the `Tip Bone` is set to the `r_gripper_palm_link` and the `Root Bone` to textttr_shoulder_pan_link. In order to set the limits, one can expand the list under these settings, select any bone and set which axis should be limited, if the joint is fixed or revolute and the min and max limits can be set in radians. Another feature is the `default bone angle` field. It allows to define the starting point for the joint. This is used to create PR2s initial position, where both arms are bend at the elbow and facing down slightly. If the arms remain outstretched, maintaining the default position given by the URDF, the IK will have a harder time finding a valid configuration for nearby

[41]Eigen library: `http://eigen.tuxfamily.org/` (last accessed: 06.12.2020)
[42]Importing a third party library into Unreal Engine tutorial: `http://www.valentinkraft.de/including-the-point-cloud-library-into-unreal-tutorial/` (last accessed: 06.12.2020)
[43]Microsoft Visual Studio: `https://visualstudio.microsoft.com/de/`
[44]CMake with Microsoft Visual Studio: `https://docs.microsoft.com/en-us/cpp/build/cmake-projects-in-visual-studio?view=msvc-160`

poses. A default pose can also be defined via animation blending, but this might lead to some bad side effects, in which the poses are permanently offset for the IK solver. This is what seemed to happen during debugging, but could be investigated further in future work.

The main development is within the `EvaluateSkeletalControl_AnyThread(...)` function. First, all members of the IK chain are found by iterating from the `Tip Bone` to the `Root Bone` and adding every found parent which does not match the `Root Bone` to an array of pairs, where each pair contains the bone name and the Unreal Engines internal bone index. Every bone within a skeleton has a unique autogenerated index received upon import. Since the obtained array contains the bone references in tip to root order, and the IK solver requires them in the root to tip order, the array is being flipped to correct this. Then another array gets initialized, which contains all the default values for the bones. If the IK has already moved the bones previously, then this step is skipped since the array will contain the previously calculated poses of the joints, aka. the previous state, which is used as a seed state for the solver.
Before any poses can be fed into the IK solver however, the difference in units needs to be accounted for. In Unreal Engine, poses in world and component space are calculated in centimeters, and local space transforms are represented in meters. KDL requires poses to be in defined meters also. Therefore some conversions need to be made to match the different units to one another. Since there were some issues with scaling, the scale parameter of all transforms received from the Unreal Engine is by default set to $1.0, 1.0, 1.0$, to ensure that it stays correct and does not accidentally get affected by transform multiplications.

Before the IK chain can be created, the corresponding `KDL::Joint` and `KDL::Frame` objects need to be defined. Once again, the list of all bones is iterated upon and for each bone a `KDL::Joint` object is created, containing the axis of rotation specified earlier in the settings of the node. The `KDL::Frame` contains the location vector of the bone in local space. Once both of these objects are generated, they are both added to the `IK-chain`, which combines these two elements into a `KDL::Segment`. At the same time, two lists of joint limits are generated, one for the maximal and the other for the minimal joint angles specified in the node. The input to the KDL solver is not only the `IK-chain` but also an array of joint angles of the previous KDL computation or the default angle state given in the settings of the node.

Calculate the necessary Transforms
The parent bone of the root bone is obtained, which in this case is the `torso_lift_link` and the goal transform, which originally is provided to the node in world frame, is being recalculated to the `torso_lift_link` frame. It is important to note, that within ROS transforms in a product are usually applied from right to left and in Unreal Engine, they are applied from left to right. This initially caused a lot of errors.
In order to apply this transform, the transform of `torso_lift_link` is obtained in component space. The values of the location component are divided by 100 so that they are converted from centimeters into meters. The scale is set to $1.0, 1.0, 1.0$ for reasons previously mentioned, and an offset of about 90 degrees is added around the Z axis to the rotation component. This offset is there because the stretched-out arms of the PR2 when pointing forward, are not in their 0 position rotation wise. Instead, they have an offset of $+90$ degrees around the Z axis, which needs to be compensated by an additional transform

multiplication of just this offset. The code snippet, performing these calculations is the following:

```
FTransform RootTransformComp = Output.Pose.
    GetComponentSpaceTransform(Output.Pose.GetPose().
    GetParentBoneIndex(RootIndex));
RootTransformComp = FlipTransform(RootTransformComp);
RootTransformComp.SetLocation(RootTransformComp.
    GetLocation() / 100);
RootTransformComp.SetScale3D(FVector(1.0, 1.0, 1.0));
FTransform adjustYaw;
adjustYaw.SetRotation(FQuat(FVector(0, 0, 1), 3.14159 /
    2)); //arm offset rotation
FTransform GoalInBoneSpace = adjustYaw*
    ScaledEffectorGoalTransform * ComponentTransform.
    Inverse() * RootTransformComp.Inverse();
```

The types are `FTransform` since this is the transform-type Unreal Engine works with. After this calculation, the scale is being fixed again, and the same operations are being performed for the calculation of the tip bone transform in `torso_lift_link` space.

Since KDL uses its own types for vectors and quaternions (e.g. `KDL::Vector`), the goal pose needs to be converted into these types. After this, the solver is finally initialized and computes the joint angles needed for the `IK-chain` to reach its goal. If the goal cannot be reached and maximal amount of iterations is exceeded, an error message is printed and the arms simply remain in their last known position. This is how the arms essentially can get *stuck*. One can accidentally move them in a way that KDL cannot compute a solution to move them out of the configuration. After this, the solution computed by KDL needs to be converted into Unreal Engine's units, meaning the location component back into centimeters as well as the types need to be cast into Unreal Engine vector and transform types again. The result is then applied to the skeletal mesh.

Most of the code needed to use KDL is essentially just parsing values from Unreal Engine data types and classes into Eigen or KDL data types, and then back again. For future work this could be generalized so that it wouldn't need to cloud up the code needed to run KDL and the necessary transform multiplications.

A feature is also the potential detachment of the grippers. They are always located at the same pose as the goal which is sent to KDL, which is based on the motion controller location within the VR world. When the goal is reachable and KDL finds a solution for it, the arm will be attached to the gripper at the wrist and move with the movement of the gripper. If KDL cannot find a solution, the arm will detach from the gripper instead. This allows the human user to still see where the goal is, and also take notice that this potential configuration does not seem to have a solution. The user can then try to reattach the gripper by moving it close to the wrist again and try to move the arm out of the locked configuration.

Since the node currently includes specific mentions of the PR2 links, namely for the feature mentioned above, it is being checked if a name matches to e.g. `r_gripper_palm_link`. Therefore the node is currently still called the `PR2 IK` node, since that call makes it PR2 specific. Once these references are removed or exposed to the node settings available within the blueprints, it can be renamed into a general KDL node.

4.2.4 Unreal Engine: Animation of the PR2 robot

This section will describe in more detail which features were implemented within the animation blueprint in order to create a realistically moving PR2 model.

4.2.4.1 Opening and Closing Grippers

The opening and closing of the gripper is mapped to the trigger of the motion controller. Since the values of the trigger input are between 0 and 1, they are scaled up to be in the range between 0 and 30. This range is based on the visible range of motion of the gripper. These calculations look like this:

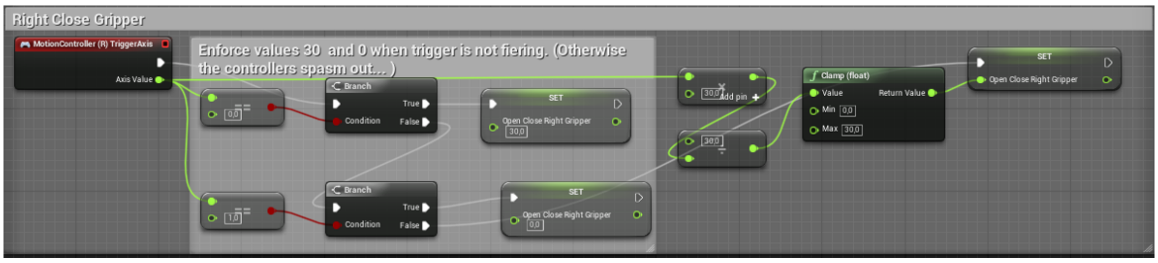


Figure 14: *Scaling of the motion controller trigger value in order to be in the range of 0 to 30*

Based on this value, within the animation graph one finger of the gripper, the `r_gripper_r_finger_link` gets moved via the use of the `Transform (Modify) Bone` node, which takes the value calculated based on the trigger and transformed into a Z component of a rotator and adds it to the current rotation of the bone in bone space. The `r_gripper_r_finger_tip_link` copies the resulting rotational movement from the `r_gripper_r_finger_link` using the `Apply a Percentage of Rotation` node, but since it has to rotate into another direction around its Z axis, the multiplied of -1 is applied. This is repeated accordingly for the fingers on the left side of the gripper. The resulting chain of movement can be seen in the following figure:

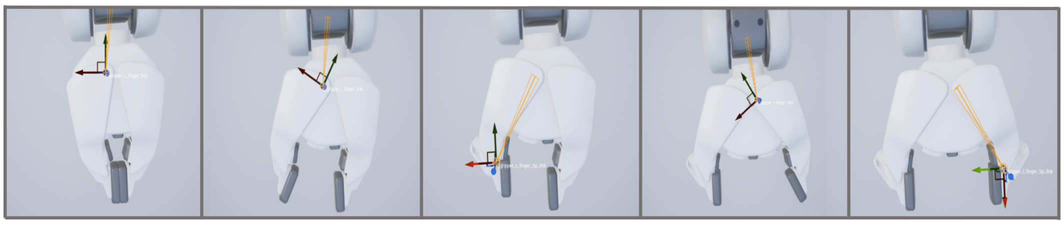


Figure 15: *Opening and closing of the gripper based on a sequence of copying the rotation of one bone.*

4.2.4.2 Grasping Objects

Grasping objects is implemented based on ray tracing between the fingers of the gripper. A socket within the skeletal mesh is created, representing the origin point from which the ray tracing should take place. The socket is in the same position as the `mesh_r_gripper_r_finger_tip_link`, since it is defined as its child. The reason for the use of a sockets is, that it is easier to access via blueprints than a bone would be. The socket will however follow the pose of its parent bone. Within the `BP_PR2_Character` another scene component, `RightHand_Grasping_Point` is added, which follows the position of the socket and to which a *SemLog* object is parented. This object is important for generating grasping events, but this will be discussed later on in this chapter. Based on the location of the `RightHand_Grasping_Point` and if a pickup event occurs, which occurs once the trigger of the motion controller is pressed, the ray tracing event is triggered. A short ray is being projected between the fingers of the robot and if another object is in between them, and this object is a bowl, cup or spoon, it is attached to the gripper and its physics simulation is being disabled. If it would not get disabled, the object would very likely slip out of the gripper or move weirdly during the transporting event, since it is essentially attached to only one point it tends to wobble and rotate. Once the motion control trigger is released and if an object is currently held, the object is released and its physics simulation is enabled again.
A physics handle in order to perform physics-based grasping was also implemented and tested, but for the current setup which is very animation heavy, the attachment implemented in the way above is just more robust and easier to work with from a user perspective.

4.2.4.3 PR2 Head motion

In order for the PR2 model to feel more responsive to the user, the robots head movement was mapped to the HTC VIVE Headset. The rotation of the headset is being obtained and the Z and X components are extracted. The Z component is being put back into an empty rotator and passed on a `Transform (Modify) Bone` which applied this rotation to the `head_pan_link`. The same is being done with the X component, just that this time it is applied to the `head_tilit_link`.
Depending on if the PR2s body-rotation is mapped directly to the headset or to buttons on the motion controllers, this implementation might be less visible. In the first scenario, only the up and down movement of the head will be seen by the user, since the rotation from side to side is also followed by the body. With the second navigation scenario, the head would be fully movable.

4.2.5 Robot to Human Body Adaptation

This section will cover the measures taken to try and make the usability of the virtual robot as natural and intuitive for the human user as possible. This will also cover how some differences between the two bodies were handled and addressed, and which solutions were found to try and eliminate these differences.

4.2.5.1 Gripper and Arm Range Extension while Compensating for Torso Bending

As also already mentioned in her thesis by Zihe Xu[20], the PR2's arms are longer than the average human ones. However, a human is able to compensate for this by bending the torso forward, while the robot cannot. In order to mimic the robots limitations as closely as possible, bending of the torso should be avoided since in a grasping scenario, it would also move the position of the VR camera closer to the surface from which the object is being picked up. Since the base position of the robot is being calculated based on the VR Headsets position, this can lead to the robots base colliding with the environment. In order to avoid that, bending the torso forward should either be prohibited, or not necessary. It was already attempted in a previous approach[20] to prohibit the bending of the human torso by attaching another tracker onto the human users chest and providing visual ques accordingly. However, this only had limited success. While investigating this same issue and trying to solve it based on the headsets position alone, it was also found that just by looking down while wearing the headset, the Z component of the transform which describes the height position, varies so much, that it would be very likely nearly impossible to differentiate between the bending of the torso or just simply looking down.

Therefore the attempt in this thesis is to render it unnecessary for the human to bend the torso in the first place. To achieve this, the human user is able to move the detached grippers forwards or backwards, basically reaching forward beyond the motion controllers position. This allows to deal with two issues at once: It allows to compensate for the difference in arms length between the human and the robot, since the human can now virtually extend the grippers beyond the length of the users arms, and it allows to avoid the bending forward issue, since it is now less effort to just press a button instead of having to bend forward to reach for something. Of course, this effect applies more over time, the longer the system is being used by the human. This feature also helps with the initial adjustment of the human user to the robots shape, since a comfortable position to maneuver the robots arms can be found within a few seconds after launching the project. From a personal perspective, it almost feels like stepping into a robot-suit.

The position of the Grippers is computed based on the current position of the motion controller. The forward-vector is obtained, and scaled incrementally, depending on how long the up (face button 1) or down (face button 3) buttons on the track pad of the controller are pressed. The grippers are then offset into the forward or backward direction accordingly.

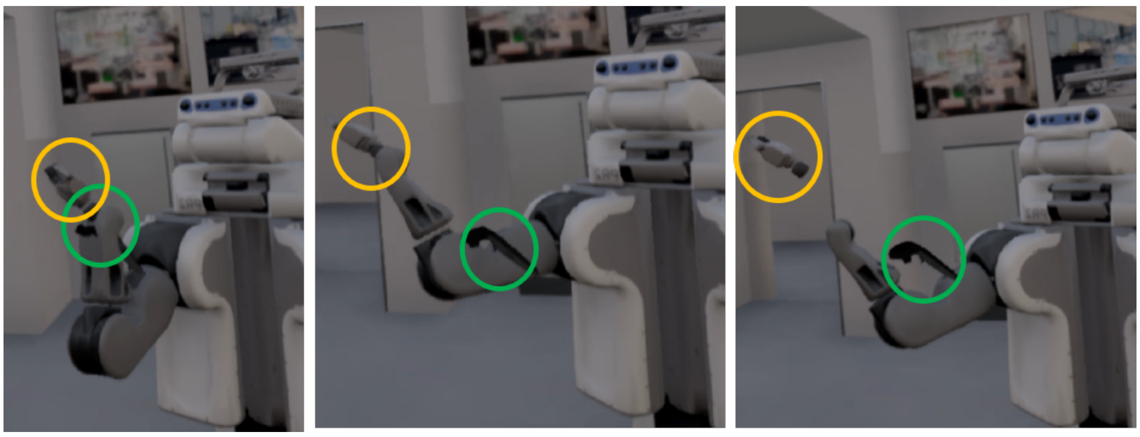


Figure 16: *Gripper detachment feature. the green circle shows the position of the motion controller, while the orange circle shows the position of the gripper and how it can be extended.*

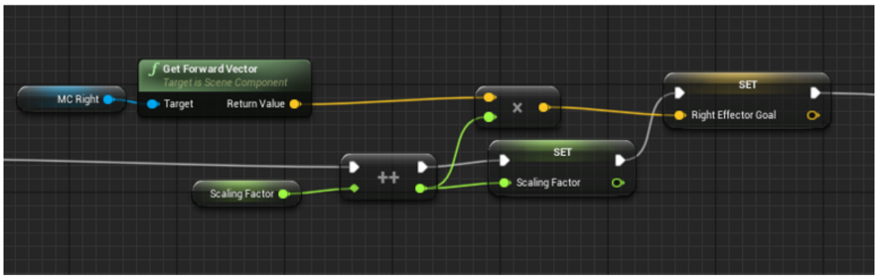


Figure 17: *A part of the blueprint which is calculating and scaling the forward vector, in order to adjust the robot to humans arms length.*

4.2.5.2 Mirror

In order for the human user to get a better understanding of how the robots body behaves based on the provided input, a very large mirror was setup within the VR environment. The mirror was simply created by creating a rectangle mirror body, which essentially is just a flat cube object, which got a very reflective material applied to it, and which is encapsulated within a planar reflection component. The mirror is overall very useful and can be also used for when an arm gets IK-stuck out of the field of view. It is a very simple but also effective solution.

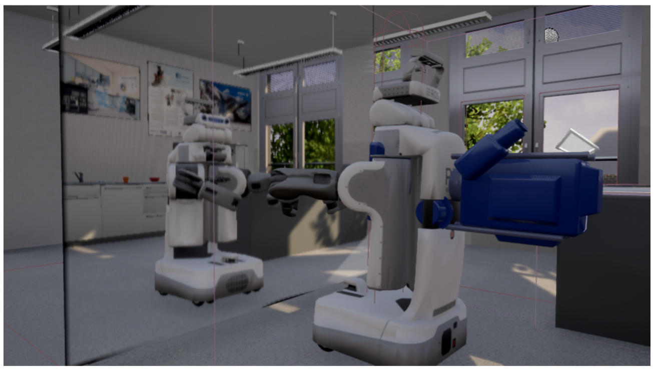


Figure 18: *A mirror object to help the human user to settle into the robots body.*

4.2.5.3 Body height adjustment

Not every human user is the same height as the PR2, and since the VR Headsets height is based on the human users height, the PR2 needs to adapt accordingly. If the robot does not adapt, manipulating the VR world through the robot gets an unpleasant uncanny feeling, since the human user is either stuck in the middle of the torso and the shoulders and arms height of the robot do not match up with the humans at all, or the user could be floating above the robot, experiencing similar issues. In order to avoid that, depending on the height of the human, the torso of the PR2 is moved up or down.

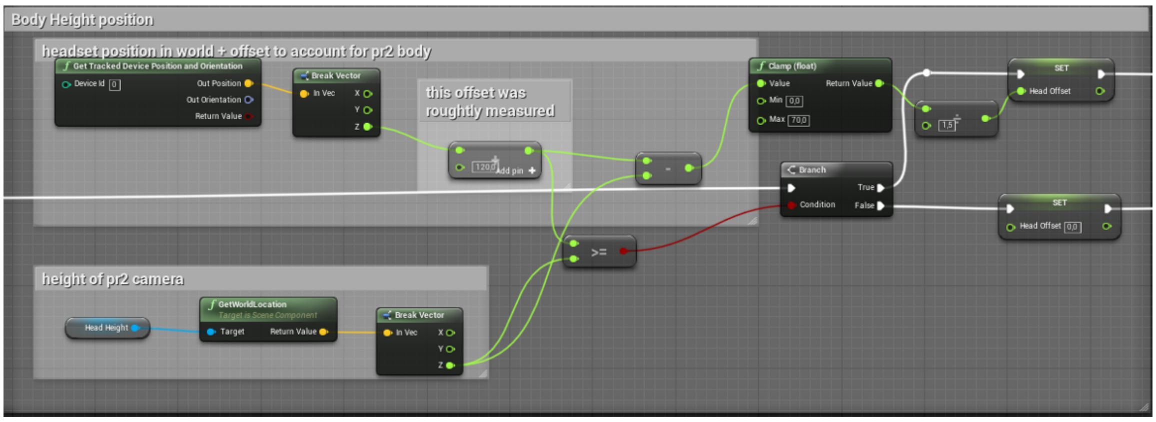


Figure 19: *Body height calculations.*

Some of these values might seem rather large on the first glance, for example the `Clamp (float)` node is set to 70, which is a rather large value to add towards the height of the robot. The value is divided by 1, 5 in order to scale it down again. This is done because if the mapping is setup in a one to one way, the body ends up shaking, since the headsets height varies constantly to some degree also. After some trial and error, this seemed to be a good solution to the otherwise very jumpy torso problem.

4.2.6 Adapting USemLog to collect VR data

In order to be able to log everything the human does within the Virtual Reality environment, a logger is needed. USemLog is a plugin for Unreal Engine providing just that. However, since this thesis uses the pipeline developed prior[10][11], and since the new version of KnowRob, at the time of writing this is not yet complete, it was decided to stick with the old KnowRob and therefore also the old USemLog 0.2[45] version.
In order to setup the logging, different objects needed to be added to the VR environment, depending on what kind of event needs to be tracked. Also, each object need to be defined within its *Actor - Tag* property, as a dynamic or static object. Other properties can be of course given also. E.g. what class an object belongs to or which mesh should be loaded for its representation. For example, the Tag of the PR2 robot looks like this:
`SemLog;Runtime,Dynamic;Class,PR2;PathToSkeletalMesh,package://robcog/Content/Models/Pl`
The ID is always autogenerated, whenever the `Generate New Ids` button is pressed within the SemLog plugin. All dynamic objects can be moved, while static objects stay where they are. This differentiation is important. When this tag is provided, raw data, meaning the position of the object within the world is being recorded to a json file. Since the PR2

[45]USemLog 0.2: `https://github.com/robcog-iai/USemLog/releases/tag/v0.2`

is a skeletal mesh object, the position of every bone is being recorded.
In order to be able to generate contact events between objects, an `SLContactManager` object needs to be added to the surface, in a way encapsulating it. These objects were added for the kitchen island and the sink area, since these are the surfaces which hold the objects for the performed pick and place tasks.
I Generating grasping events was a bit more tricky. This event generator is essentially bound to an implementation of the motion controllers within the UMCInteraction[46]. Since motion controllers were already setup and defined for this thesis, the code of the UMCInteraction plugin was altered and decoupled from the motion controllers. Fundamentally, a sphere is attached to the gripper of the PR2. Whenever something overlaps with this sphere, a grasping something event is generated. This solution was very hacky and is not necessarily stable, but it works and generates some data.
In order to be able to generally log data, two more objects need to be added to the world. These objects are: a `SLFurnitureStateManager` which, as the name already suggests, logs the state of all the furniture items within the world, and a `SLRuntimeManager`, which generally contains settings as to how often raw data should be recorded and similar.
Since there was not much documentation about this very old version of this plugin, it could be that the setup is wrong, since it was essentially reverse-engineered based on an old RobCoG project version. It does generate data, but does not seem to be as accurate about it as former data sets suggest.

[46]UMCInteraction plugin: `https://github.com/robcog-iai/UMCInteraction`

5 Experimental evaluation

5.1 Evaluation of the VR-PR2-model and behavior

This section will discuss how the generated PR2 model behaves within the VR environment, based on the overall look and feel from a user perspective. It will be described how it feels to pilot the PR2 and to be *inside the robots body*, how responsive the system is, which features it contains and which flaws. It will be also discussed how performing a pick and place task as the PR2 robot compares to performing the same task with the previous method of just piloting human hands within the VR environment. This comparison will unfortunately be only subjective, since it is currently impossible to conduct a proper user survey because of the current pandemic. However due to my personal previous experience with the RobCoG, hands-only-VR system within my Bachelors thesis and the evaluation performed for a paper I will try my best to make a fair comparison between the two systems.

5.1.1 Differences between the systems based on general differences between the human and robots bodies

The following sections will discuss the general differences between the human and robots bodies, as well as how the two different systems, meaning the RobCoG approach with only human hands in VR and the new approach in this thesis of having the robots body in VR, compare to one another, based on look and feel for the human user, as well as how these differences of the bodies might play into the data acquisition process for teaching the real robot to perform everyday pick and place tasks. Some aspects which are described in the *general look and feel* section are results of the mapping between the human to robot body and the reasoning behind them will be explained in their respective sections.

5.1.1.1 General look and feel

The **RobCoG** system allows for a lot of freedom for the human user. Since one only has to essentially focus on the virtual human hands, and not have to take into a count any sort of body, one can really focus on grasping objects, from which angle and direction to grasp, and to try and make the grasps as realistic as possible, if one wishes to. Since the objects attach to the palm of the hand however, some grasps are harder to perform then others. For example, grasping the spoon is probably the hardest one. In order to grasp it, one has to press essentially the hand completely onto the surface on which the spoon is located, so that it can get in reach of the palm, in order to be grasped. This is not necessarily a realistic grasp, neither for the human nor the robot, but this is a rather extreme example. Navigating the virtual environment is rather easy, since, again, one does not have to worry about a body potentially colliding with things. It also allows to pass through furniture, as long as the human user holds the hands in a way that they won't collide with the environment. For example, in order to place an object on the opposite side of the kitchen island, one can walk through it, even though that feels weird for the human user since the human brain obviously sees an object there that one usually cannot just walk through, instead of having to walk around the kitchen island. This might sound crazy at first but it can be beneficial, since the cable of the VR headset has a limited length and this allows the user to reach positions which otherwise would be unreachable, limited by the size of the available real world environment and the mentioned cable length of the headset.

Since the hands have physics objects defined on them, one can open drawers by hooking the hand between the handle and the drawer and just pulling to open the drawer that way. The collision occurring between the hand and the drawer handle will cause it to open as expected. One can also push the drawers closed by just pushing on them with the hand, without needing to grasp anything for it.

Overall the system is fairly intuitive to use, it is a lot of fun. The grasping takes a while to get used to and some limitations of the system, like missing a physical body, can be abused to gain some benefits where Virtual Reality might have been limited by the real world.

The **PR2 VR** system is comparatively quite different. The human user now has a new body that is very different than the real human body and which takes a while to get used to. In order to ease in this transition, a full body mirror was created in which the human user can observe how the robots body behaves based on the human movement. The robots body follows the headset completely, including the rotation. This makes the navigation of the robot within VR intuitive, but since one cannot rotate the head independently of the body, it can in some specific cases be a hindrance. For example, if one wants to look at the robots arm which might be just out the field of view, one cannot just rotate the head and look at it. With the current implementation the human user would have to walk back to the mirror, to see how exactly the arm is positioned. Since the grippers are attached to the position of the motion controllers rather than being attached to the rest of the arm, the intuitive first reaction is to attach them to the arms again, which can be done by simply moving both grippers to the wrist. Since, as will be explained further down in this chapter, the arms of the robot are a lot longer than the human arms, the human user can move the gripper further forward or backwards by pressing buttons on the motion controller. While this sounds like it might not be intuitive, it does feel intuitive. From my personal point of view, it feels like one is stepping into an exo-skeleton, which happens to be the PR2 robot. This feature also allows for the human user to stretch out the PR2s arm beyond the humans arms length but within the robots arm length, allowing for better robot base positions since now the user does not have to move that close to the surface the objects to manipulate are located on. After a while, once one gets used to this feature, it is noticeable that instead of stretching out the real humans arm in the real world, it is preferred to just stretch out the virtual arm instead, while keeping the real humans elbows rather close to the torso. After all, pressing a button requires less effort than moving an entire arm.

Grasping objects is currently not physics based at all. This means that in order to grasp an object, it has to be located between the two fingers of the gripper. This also means that the grippers currently will always close completely in order to grasp an item, going visually through it in the process. While this is not very realistic, for now it is a decent solution that can be improved upon in the future.

5.1.1.2 Real world navigation mapping to VR

In RobCoG the position of the VR headset has no visual representation. It is just the position where the human user is within the VR world. The position of the motion controllers follow the headset and are always positioned relative to it.

The VR PR2 also follows the position of the VR headset and also rotates with it. However there are essentially two ways of setting up the rotation. In this case the rotation of the headset was mapped directly to the body. Another option is to map it to two buttons instead, so that the head can be moved completely independently of the body. In earlier testing, the second approach was the original idea, to just rotate the body when needed on button press, so that it matches the rotation of the headset again. However for collecting data, it seemed rather tedious to always have to set the rotation manually, so the first approach was rather implemented. However this is open to preference of the user. Rotating the body of the PR2 including the camera on button press would however cause motion sickness, so this approach was not followed further.

5.1.1.3 Robot to human height adjustment

Within the RobCoG approach, the height of the human user compared to the robot was never an issue, since there was no body being visualized which might have been impacted by it. The motion controllers were directly mapped to the position of the human hands in the real world and that was completely sufficient.

However, when accounting for a robots body which needs to be mapped to the human user one, height becomes an important concern. The robots torso need to move up and down depending on the height of the human, within a limit, so that the head-position as well as the arms and shoulders heights feel natural to the user. If the robots body is too low, the viewpoint of the human is above the robot, the shoulders are way too low and then the overall interacting experience suffers. Same applies for the robots torso being too high up, which would lead to the shoulders being above the head and the viewpoint being inside the torso of the robot. If any of these two cases occurs, it gives an uncanny feeling to the user since one can see that something is not quite right. Therefore, the height of the robots head is essentially mapped to the height og the VR headset, by moving the torso of the robot accordingly to match the height.

This solution however has its own limits. One can argue that very small or very tall people might hit the limit of the capabilities of the real PR2, since the torso only has a limited movement space. In order to compensate for it, one could go above the PR2s limit. It might not look too off visually within the VR environment but would have to be accounted for later, when processing the data. For rather small people there might not be such an intuitive solution. Moving the robots torso down has a fixed limit. It can be forced down further but then it would also look ad feel wrong, since the shoulder-meshes would collide with the base and essentially need to pass through it. There the solution might lay in scaling the entire world down, or the PR2's body. But that would be a rather exceptional and rare case.

5.1.1.4 Human torso bending

The human user can bend the torso forward, in order to reach for an object and keep the feet in place. This prolongs in a way, the arms reachability capabilities. While the user can do so within the RobCoG game, which might lead to potential positions for the robot which collide with the environment, since the head and therefore the headset on which the base position is later based moves forward also, this is not possible to do with the PR2 VR model. Of course the user can perform this movement in the real world, there is no stopping that, but the result in VR would just be not the expected one. Since when bending forward, the head also moves forward, it would mean that the entire robot moves forward in VR also, potentially colliding with the environment. Also the grippers might detach from the wrist in this process and the human user learns quite quickly that this is an overall undesired effect, not gaining the expected benefit to the situation. Instead it is way more comfortable to stand upright at the same spot and just press a button to stretch out the arm further. After just a few minutes, this adaptation is intuitively made and the torso-bending is no longer a concern.

5.1.1.5 Human feet and the robot-base position

Humans have very small feet, compared to the PR2 robots base. Also our feet are located directly under our head, while the base of the PR2 robot extends forward by quite a margin. This generally means that the human can stand a lot closer to surfaces for pick and place activities as the robot would be able to.

Within the RobCoG approach, there was nothing to account for this difference during data acquisition. The human is not able to extend the VR hands to be able to manipulate objects from more of a distance, and there are no visual ques at all for the position of the feet or the size of the robots base. For the so collected data, this would mean that the feet position is calculated based on the position of the VR headset, removing the z component of the pose in order to project the position onto the floor. For the robot to be able to use these poses to learn e.g. where the human was standing when he grasped an object, an offset of 0.2 meters was added to the cameras position, basically projecting the position away from the object, to make space for the robots base.

With this new approach of the PR2 robots body being in VR, the human user can directly see the robots base and be therefore made aware of the limitations and potential collisions it entails. Also, not only is the camera position tracked, but since the PR2s body is a skeletal mesh, the position of each bone is recorded. Meaning that the position of the base can be used directly, and no offsets need to be applied to it.

5.1.1.6 Human arms and robot arms

As previously mentioned throughout the thesis, the arms of the PR2 robot are a lot longer than the human arms. One rather major difference is that the grippers of the PR2 are initially detached from the rest of the arm. The gripper locations are mapped to the motion controllers, while the movement of the arm is computed by KDL. In order to compensate for the length difference, the location of the grippers can be offset to the the motion controller on button press. The grippers can move further forward or backwards, however the human user prefers or needs. Some might prefer to have the motion controllers as close to the grippers as possible and to stretch out the human arms as much as they can, in order to achieve that, while other users might prefer to keep their arms rather close to their body, and use this feature more instead. Generally, this feature allows the human user to keep standing further away from a surface, keeping the distance far enough to not collide the base with the potential surface base, and use the length of the PR2s arms to an advantage. The detachment of the grippers also compensates for the realistic scenario of KDL not being able to find a solution for the current configuration. Instead of completely stopping the arms movement and potentially confusing the user, the gripper only detaches. The user is than made aware by this that the current position or grasp attempt, might not be suitable for the robot. The user can then reattach the gripper by moving it closer to the arm again and moving the arm to an overall better configuration for the robot. Sometimes the reattachment does not work completely, and as un-intuitive as it sounds, shaking the controller a bit can help. Sometimes however the arm can get stuck entirely, and the project has to be restarted entirely to unstuck it. In future work however, there could be a button implemented to just reset the arms position to a default position to help unstuck them.

5.2 Result of the Evaluation of the VR-PR2-model and behavior

Based on the previous section and my subjective overall experience, the differences in both approaches are many. With the PR2 model as a body inside of VR, the human user has to keep a few more things in mind while performing pick and place tasks. One has to keep an eye on the base, although after a while as a user one gets an intuitive automatic estimation of where it is without having to look to check. Because the movement of the arms is calculated by KDL, which occasionally might not find a solution, one has to keep an eye on the gripper to arm connection, quite literally. While within RobCoG the focus lies completely on the hands, here the focus is also on the overall arm position and how it behaves. The resulting movement might be slower and more controlled since instead of just watching what object one is grasping, the entire arm is being watched. It is also pretty hard to keep both grippers attached to both arms at the same time. The resulting solution was one of two things: either one performs the pick and place setup one arm at a time, transporting all objects from the sink to the kitchen island one by one with first only the right arm, and then with the other arm to essentially generate a full data set. Or to keep the gripper to arm connection while performing one grasp, then basically ignoring the detachment while the other arm is grasping and ignoring it during the navigation portion between the kitchen island and sink area, and then attach the grippers again to the arms while placing down the objects. The later solution is an acceptable one, because for the real robot one can define stable carrying positions which can be used no matter which object has been grasped and needs to be transported. Usually these carrying poses would also want to move the arm out of the field of view, in order to not obstruct future perception tasks, which is something the human user might want to avoid in VR, since if the gripper gets detached while outside the field of view, one would need to go back to the mirror to see and reattach it.

Collecting pick and place data with the PR2 body is overall very similar to doing so with RobCoG. The differences are mainly that while collecting data with RobCoG, one could either move completely freely and behave entirely like a human would, bending forward if needed be, or choose to try to generate data which might be the most useful to the robot later on. The later would require to always keep in mind not to bend the torso forward, try and imagine how the PR2s gripper would be able to grasp an object, try and stand in front of an area in a way that would seem like a position that would be plausible for the PR2 given a specific offset. All these things come with experience and observing either the real PR2, or collecting the data and running a lot of simulation on it in order to learn how to generate good and usable data. A lot of these things are now no longer concerns since one can directly already see if the PR2 would be able to stand in a specific location, if the arm can reach the object, if it fits into the gripper in order to grasp an object from the front, if the arm or the base would collide with something while performing a certain motion. Of course, there is room for improvement and a lot of feedback can be provided to the human user via vibration of the controllers or more visual ques, but this is another step into generating better data to teach robots with. This solution also gives the human user a better understanding of the PR2s movement capabilities. One could say, the human user learns about the robot by becoming the robot.

Grasping objects as the PR2 is a lot more different then what it was expected to be like when collecting the data previously within the RobCoG setup. With RobCoG more top grasps were performed, because it seemed to be the easier way to grasp something for the robot. While inside the PR2s body, getting a good arm configuration to perform a top grasp was not as easy as it once seemed, so that in the resulting data many front grasps were performed, or things were grasped from the top but with an angle, instead of the straight down approach. This might be due to the arm tending to get stuck in a configuration while performing that grasp, or because it feels like an unusual movement to perform as a human. In RobCoG, since one had no visual representation of the robot, it might have appeared to be the only good way to grasp an object. Now, that one could directly see another solution working at least virtually, as a human one might intuitively resort to a more comfortable movement for grasping.

Overall, this solution could serve in generating more reliable data for the robot in order to perform pick and place tasks and further help in understanding, as a human, how a robot interacts with the environment since one can experience it, in a way, first hand.

5.3 PR2 KDL IK Evaluation

In order to be able to move the arms of the robots body as realistically as possible, an inverse kinematics solver node was implemented, which uses the KDL library with the Newton-Raphson solver, which is also used for the real PR2 robot. This node takes joint limits into account and is used to calculate the joint angles between the `gripper_palm_link` and the `shoulder_pan_link` of each arm. The limits have to be set by hand in radians, and in order to help avoid getting the arms stuck in the outstretched forward position, default values can be passed to the node in order to generate an initial pose. To further make the piloting of these arms a bit easier, and in order to avoid having the upper arm hit the table while performing a front grasp, the limits set for some links were stronger limited than the PR2 description and URDF file would specify. This mostly affected the `shoulder_lift_links`. Both continuous joints of the arms were fixed or strongly limited, since it was observed that KDL struggles very hard to find feasible configurations with continuous joints. These changes were more of quality of life improvements and can easily be adapted to the original PR2s limits without having to touch the nodes code. As already discussed previously in this chapter, the feature of detaching the grippers is also partially implemented within the KDL node, since if it cannot find a solution within the set amount of iterations, it will detach the `gripper_palm_link` from the rest of the arm. Currently this is hard-coded within the nodes code and would need to be adjusted for the future. Overall the PR2 IK node works well. It takes a bit of practice to get used to the robots arm positions, but it is very important for future work, that now while collecting data for the robot, the human user gets direct visual feedback of what would be reachable for the robot and what not. The user can also be sure that if e.g. the grasping action was performed without getting the gripper detached from the arm, the real robot will be able to replicate that movement.

5.4 PR2-body-based VR data evaluation

A goal of this thesis was to collect data from the Virtual Reality while performing pick and place tasks within the PR2s virtual body. A dataset was created for this purpose, containing 15 episodes, of which each contains a full set of bringing the bowl, cup and spoon from the kitchen sink area to the kitchen island and back once with the right and then with the left hand, keeping the IK chain intact as much as possible while performing the task with the corresponding hand. This data has been imported to an old KnowRob and MongoDB version, using scripts[47]. Unfortunately, not many evaluation runs could be performed, since for an unknown reason, the chain of CRAM->KnowRob->CRAM->BulletWorld kept on crashing, approximately every 5 minutes. Restarting it every time was just taking too much time, so that I had to abort this evaluation.
From the 23 performed runs within the bullet world simulator, unfortunately only one was completely successful, meaning that all three objects were successfully picked and placed. 12 runs reported only one transporting failure, 8 reported 2 failures and 3 runs failed entirely. There were no search fails, meaning that the robot could find a pose to perceive the object from every time. There occurred 24 fetch and 13 delivery fails. This kind of evaluation is based on this paper[11], since it is using the same tools.
There are a few theories to what could have caused this. Either something with the recorded data is not quite right, which is probably very likely since the logging solution is very improvised. Maybe an offset between the Virtual Reality and the CRAM Bullet World changed, since the real world environment in which the data got recorded, also changed and was very limited. An update could have broken something on accident or maybe something else entirely caused this error.
In order to solve this the dataset could be analyzed in more detail, or recreated within the same physical environment, as the previous one, just to avoid this being a real-world-to-vr-scaling issue. However, this should be investigated, so that this new approach of having the PR2 body within VR can be evaluated properly.

[47]scripts to import data to MongoDB and KnowRob automatically: `https://github.com/hawkina/vr_neems_to_knowrob`

6 Conclusion

6.1 Summary

In this thesis a skeletal mesh model of the PR2 robot was largely auto generated via a few scripts for Blender and imported into the Unreal Engine and the RobCoG Virtual Reality environment. In order to be able to replicate the robots movement as close to the real robot as possible, an inverse kinematics node was implemented for the Unreal Engine's animation graph. It uses the Newton-Raphson IK solver provided by the orocos-KDL library, to compute joint angles for the arms of the robot, taking min and max limits on a per axis basis into account. This solution can guarantee to the human user that if the robot was able to perform a certain motion in the Virtual Reality environment, the real robot could perform this motion too, since now both robots use the same inverse kinematics solver. The opening and closing of the grippers was implemented essentially based on the trigger of the motion controller and the movement of one of the finger bones. Grasping of objects was implemented via ray-tracing between the fingers of the PR2 gripper. There were a few other features implemented, in order to map the robots body to the human and provide as an immersive experience as possible. For example, the robots head moves the same way the head mounted display of the VR user does. The height of the robot is being adjusted to the height of the human. A large mirror is provided so that the human user can get accustomed to piloting the robot. A way of logging the performance of manipulation activities within the Virtual Reality using an old version of the USemLog and UMCInteraction plugins was set up. A dataset of 15 episodes was created and attempted to be evaluated.

6.2 Discussion

This thesis proves to have been a great journey. In the beginning, it was not expected that an own IK node would be implemented as a result of this. The idea was originally to use one of the IK nodes Unreal Engine already provides and the IK setup was expected to be much easier and straight forward than what it turned out to be. However, this node could prove to be useful for other researches as well and could be made into a publicly available plugin.
Same goes for the adaptation done within the phobos plugin for Blender. It could become a sister-project of phobos. Overall, while some features got implemented, there seems to be always room for more. The way the PR2 gripper interacts with the objects it grasps at the moment could be improved with physics, so that the robot does not grasp through them. Where problems were not expected at all, problems occurred. One such case was the export of the generated skeletal mesh from Blender to Unreal Engine. It has proven to be very fragile and required many attempts in order to figure out the correct settings for the export and import, as well as how an object has to look like within Blender to be exportable within Unreal Engine. It was surprising how many differences the two programs had with one another. How different skeletal meshes could be handled and that there are several ways of defining bones was also rather surprising.
Overall being inside the PR2s body and performing pick and place tasks as a robot seems in a way to be more intuitive than the previous approach since now instead of wondering if one is standing far enough away from the furniture and if the current position would generate a viable navigation pose, one can directly see it. It is also directly visible if

the robot would be able to reach a certain pose or not with the arm, since if the goal is impossible KDL will not be able to find a solution for it.

6.3 Future Work

There are many ways in which this work can be continued. For one the entire process of generating the skeletal mesh of the robot based on an URDF file could be finessed, in the sense that the file might not need to be filtered by hand to be able to be imported into blender to begin with. This could become a stand alone plugin for Blender and be decoupled from Phobos or to become a stand-alone extension or sister project of Phobos, called Deimos. The KDL-IK Node for the animation graph within Unreal Engine, which was developed for this thesis, could also become a publicly available plugin, which hopefully would help other users and researchers. Even when Unreal Engine is planning to introduce their own full-body IK system[48] KDL might still be useful to other researchers who would like to be able to use the Newton-Raphson iterations algorithm for their own research.

Overall using a robots model within VR could help new students who are starting to study robotics to understand the robots movement better and maybe even help the general public to gain interest in robots and reduce their fear of robots. The task of performing every day activities as a robot within Virtual Reality could even become a game, to help and collect more data for research.

Another plugin for the Unreal Engine could be developed, which reads in a URDF file and configures the IK automatically, based on the limits provided in the URDF. If the robot will be equipped with a full physics asset, the plugin could automatically configure the constraints within the physics asset and generate the respective bodies. If a full body physics asset would be beneficial for the robots performance in VR should be discussed, but maybe physics could be generated and applied to the base and the grippers of the robot, so that collisions can be detected and the human user could obtain feedback based on these collisions.

The entire approach could be tested with other robots, to see how it would be to pilot less human-like robots, e.g. robots with only one arm, or which are a lot larger or smaller than humans are, and which solutions might these robots need in order to be mapped to the human body, if at all.

The data collection methods can be also improved upon. The currently implemented solution uses very old plugin versions, which by now have newer versions available, with new features and a different way to set them up. Both USemLog[49] and KnowRob[50] have been hugely reworked and updated in the last few months. They should also be updated within this project.

Overall the evaluation of the here presented model should be evaluated in as much detail as its predecessor, which is described in the paper was, so that both approaches could be directly compared.

[48]Announcement of the introduction of full body IK with limits for a new upcoming Unreal Engine version: `https://docs.unrealengine.com/en-US/WhatsNew/Builds/ReleaseNotes/4_26/index.html` (last access: 05.12.2020)
[49]USemLog GitHub repository `https://github.com/robcog-iai/USemLog` (last access: 05.12.2020)
[50]KnowRob homepage: `http://www.knowrob.org/` (last access: 05.12.2020)

A navigation capability based on button-presses could be developed also. It would be important to develop it in such a way that reduces or avoids motion sickness, which is a general VR problem. This would the user to navigate a larger Virtual Reality environment and be less limited by the available real-world space for VR.

7 Appendix

7.1 Bibliography

References

[1] Andreas Aristidou, Yiorgos Chrysanthou, and Joan Lasenby. “Extending FABRIK with Model Constraints”. In: *Comput. Animat. Virtual Worlds* 27.1 (Jan. 2016), pp. 35–57. ISSN: 1546-4261. DOI: 10.1002/cav.1630. URL: https://doi.org/10.1002/cav.1630.

[2] Andreas Aristidou and Joan Lasenby. “FABRIK: A fast, iterative solver for the Inverse Kinematics problem”. In: *Graph. Models* 73.5 (Sept. 2011), pp. 243–260. ISSN: 1524-0703. DOI: 10.1016/j.gmod.2011.05.003. URL: http://dx.doi.org/10.1016/j.gmod.2011.05.003.

[3] Michael Beetz et al. “Know Rob 2.0 — A 2nd Generation Knowledge Processing Framework for Cognition-Enabled Robotic Agents”. In: May 2018, pp. 512–519. DOI: 10.1109/ICRA.2018.8460964.

[4] F. Brizzi et al. “Effects of Augmented Reality on the Performance of Teleoperated Industrial Assembly Tasks in a Robotic Embodiment”. In: *IEEE Transactions on Human-Machine Systems* 48.2 (2018), pp. 197–206. DOI: 10.1109/THMS.2017.2782490.

[5] Open Source Robotics Foundation. *Bullet world demonstration.* URL: http://cram-system.org/tutorials/intermediate/bullet_world (visited on 04/09/2018).

[6] A. Goldenberg, B. Benhabib, and R. Fenton. “A complete generalized solution to the inverse kinematics of robots”. In: *IEEE Journal on Robotics and Automation* 1.1 (1985), pp. 14–20. DOI: 10.1109/JRA.1985.1086995.

[7] A. Haidu and M. Beetz. “Action recognition and interpretation from virtual demonstrations”. In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* 2016, pp. 2833–2838. DOI: 10.1109/IROS.2016.7759439.

[8] Andrei Haidu. *Robot Commonsense Games.* URL: http://www.robcog.org/games.html (visited on 04/08/2018).

[9] Andrei Haidu and Michael Beetz. *Automated Models of Human Everyday Activity based on Game and Virtual Reality Technology.* (Visited on 04/01/2018).

[10] Alina Hawkin. “Towards robots executing observed manipulation activities of humans”. Bachelor Thesis. Institute of Artificial Intelligence, University of Bremen. (Visited on 04/23/2018).

[11] Gayane Kazhoyan et al. “Learning Motion Parameterizations of Mobile Pick and Place Actions from Observing Humans in Virtual Environments”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* 2020.

[12] Yuan-Hong Liao et al. “Synthesizing Environment-Aware Activities via Activity Sketches”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).* 2019.

[13] L. Mösenlechner and M. Beetz. “Parameterizing actions to have the appropriate effects”. In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems.* 2011, pp. 4141–4147. DOI: 10.1109/IROS.2011.6094883.

[14] Lorenz Mösenlechner. “The Cognitive Robot Abstract Machine”. Dissertation. München: Technische Universität München, 2016.

[15] Xavier Puig et al. *VirtualHome: Simulating Household Activities via Programs.* 2018. arXiv: `1806.07011 [cs.CV]`.

[16] Eric Rosen et al. “Testing Robot Teleoperation using a Virtual Reality Interface with ROS Reality”. In: Mar. 2018.

[17] Kai von Szadkowski and Simon Reichel. “Phobos: A tool for creating complex robot models”. In: *Journal of Open Source Software* 5.45 (2020), p. 1326. DOI: `10.21105/joss.01326`. URL: `https://doi.org/10.21105/joss.01326`.

[18] Moritz Tenorth and Daniel Beßler. *KnowRob.* URL: `http://knowrob.org` (visited on 04/15/2018).

[19] D. Whitney et al. “ROS Reality: A Virtual Reality Framework Using Consumer-Grade Hardware for ROS-Enabled Robots”. In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* 2018, pp. 1–9. DOI: `10.1109/IROS.2018.8593513`.

[20] Zihe Xu. “Designing Human-controlled Robots in VR for Learning Everyday Manipulation Tasks”. Master Thesis. Institute of Artificial Intelligence, University of Bremen. (Visited on 12/10/2019).

List of Figures

List of Tables

List of used software

Name	**Version or Branch**	**Repository or Download link**
Blender	2.79	`https://www.blender.org/download/releases/2-79/`
phobos	rigging	`https://github.com/hawkina/phobos/tree/rigging`
Unreal Engine	4.22.3	`https://www.unrealengine.com`
RobCoG	master	`https://github.com/hawkina/RobCoG`
CustomAnimNode	main	`https://github.com/hawkina/CustomAnimNode`
VR Neems to KnowRob	master	`https://github.com/hawkina/vr_neems_to_knowrob`
CRAM	boxy	`https://github.com/cram2/cram/tree/boxy`
Eigen	Eigen3	`http://eigen.tuxfamily.org/index.php?title=Main_Page`
KDL		`https://orocos.org/kdl.html`
KnowRob	kinetic	`https://github.com/knowrob/knowrob/tree/kinetic`
UMCInteraction	master	`https://github.com/hawkina/UMCInteraction`
USemLog	master	`https://github.com/hawkina/USemLog`

7.2 Acronyms

HMD	Head Mounted Display
CRAM	Cognitive Robot Abstract Machine
IK	Inverse Kinematics
owl	Web Ontology Language
RobCoG	Robot Commonsense Games
ROS	Robot Operating System
VR	Virtual Reality