Bachelor Thesis

Simulation and Control of a 2DOF Robot Arm

Sarah Gerster

Bradley Kratochvil

Adviser

Prof. Dr. Bradley J. Nelson

Institute of Robotics and Intelligent Systems

Swiss Federal Institute of Technology Zurich (ETH)

2005-07

IRIS Institute of Robotics and Intelligent Systems

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Preface

In this project I had the opportunity to learn a lot about programming. I got to know linux a lot better, and I learnt how to work with open source programs more efficiently. I thank the following people to having assisted me in my work and made it possible to do this project:

Prof. Dr. Bradley J. Nelson, head of the institute

Bradley Kratochvil, Adviser

Sarah Gerster
Zurich, June 2005

Contents

List of Figures

Abstract

A two DOF robot-arm with a gripper as end-effector should be developed on the base of an already existing two linked robot arm for the class "Introduction to Robotics & Mechatronics". There are several semester projects involved in this work. My tasks are the following:

- In the first part of the project:
  - write a driver to move the real arm
  - test the driver with a client program that uses the opensource software Player to communicate with the arm
- In the second part of the project:
  - create a simulation of the arm
  - find a solution to be able to run the exact same client on the real arm and on the simulation by using Player

Zusammenfassung

Fuer das Fach "Introduction to Robotics & Mechatronics" soll ein bereits existierender Roboterarm mit zwei Freiheitsgraden zu einem Roboterarm mit Greifer weiterentwickelt werden. Es sind mehrere Semesterprojekte an dieser Arbeit beteiligt. Meine Aufgaben sind die folgenden:

- In einem ersten Teil:
  - einen Driver fuer den reellen Arm schreiben
  - den Treiber mit einen Client testen, der das opensource Programm Player verwendet um mit dem Arm zu kommunizieren
- In einem zweiten Teil:
  - eine Simulation fuer den Arm schreiben
  - eine Loesung finden, so dass derselbe Client via Player sowohl auf dem reellen Modell als auch auf der Simulation luft

1 Introduction

This bachelor project is based on the work with the two-linked robot arm from the class "Introduction to Robotics and Mechatronics" [1]. The arm has been modified to have a gripper as an end-effector, such that it can pick up chess pieces. You can see a picture of the arm in Figure 1. The goal of the project is on one hand to write a part of the software to be able to communicate with the arm by using the open source software Player and on the other hand to create a simulation of the arm that acts the same way as the real arm.

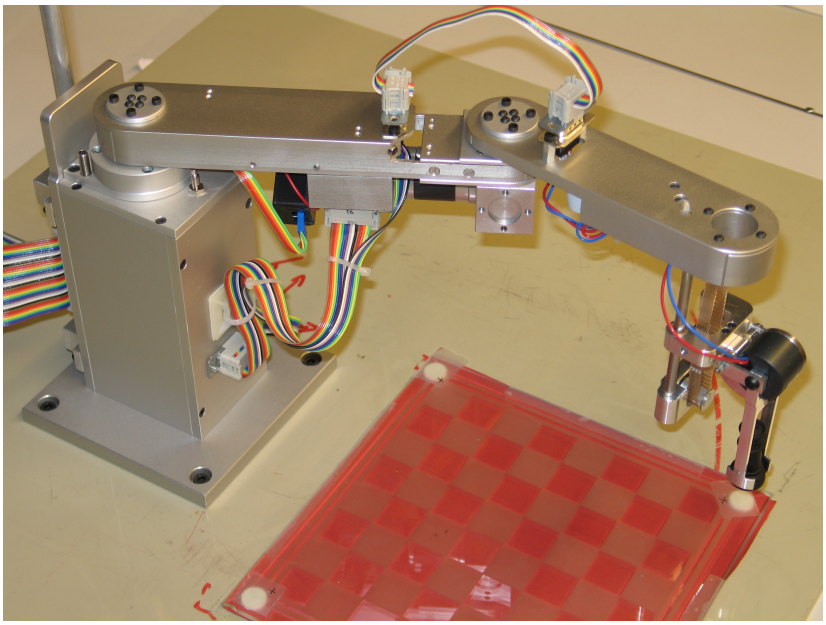

Figure 1: Two-linked robot arm with gripper

1.1 Real Arm

The arm has two motors that need to be controlled to get the tip to the desired position in the xy-plane. It has a third motor that needs to be controlled to set the z-position of the gripper and a magnet which can be used to open and close the gripper.

[1] Class at the Institute of Robotics and Intelligent Systems (IRIS) at ETH Zurich.

1.2 Simulation

The model that has to be created should show the same behaviour as the real arm. It should be robust and look as nice as possible.

1.3 Used Software

The software that is used is opensource. Player is used for the communication between the client and the arm or the model. Player is a network server to control robots. There are more informations about it in section 2. Gazebo is used for building the arm model. Gazebo is a simulator for mobile robots. You can find more informations about it in section 3.

1.4 Documentation

This report is part of the documentation of the created model. It can be used to determine how the simulation is meant to be used. It can also be used to develop new simulations with Gazebo. The problems that occured and the way they were solved are documented in the code. There won't be much details about these problems and their solutions in this report. It is easier to understand an issue, if you actually see the piece of code. Therefore, there are a lot of comments in the code. With this report, you should have gotten a CD-ROM with my code. When describing a problem, there will be a note to tell you in which file you can find more informations about it. In Appendix B you can find a list of all the stuff that is on the cd-rom.

2 Player

2.1 Introduction

Player is free software, released under the GNU Public License. It is a network server for robot control. Player provides an interface to the robot's sensors and actuators. This means, that a client can send its commands to the Player server, which then passes the commands on to the right driver. For this project, the use of Player makes it possible for a user to write one single client for both, the real arm and the simulation. Depending on how Player is started, the server will send the commands to the right driver. Therefore, one can use the same client with different drivers simply by changing the configuration file which is used to start Player. There are two sample configuration files on the CD-ROM.

2.2 Functionaltity

The client connects to the Player server which then connects to different drivers. A principle sketch of the way communication through the Player server works is shown in Figure 2. This sketch shows the drivers used in this project. `GzArmDriver` and `gzPosition3d` are used when the commands should be sent to the arm model. `ArmDriver` and `s626` are used to control the real arm. The `men` driver would be used if the arm was fixed on MARVIN [2].

`GzArmDriver` is needed to transform position into velocity commands, because `gzPosition3d` only accepts velocity commands. `ArmDriver` is used to transform commands given in taskspace coordinates to jointspeeds or joint positions which can be handed on to `s626`. There is no direct communication between `GzArmDriver` and `gzPosition3d` or between `ArmDriver` and `s626`. All the communication is done by passing through the Player server.

Another important thing to know is that Player is multithreaded. This means that several clients can be started at the same time and that they can talk to different physical devices.

[2] Mobile robot developed at the Institute of Robotics and Intelligent Systems (IRIS) at ETH Zurich.

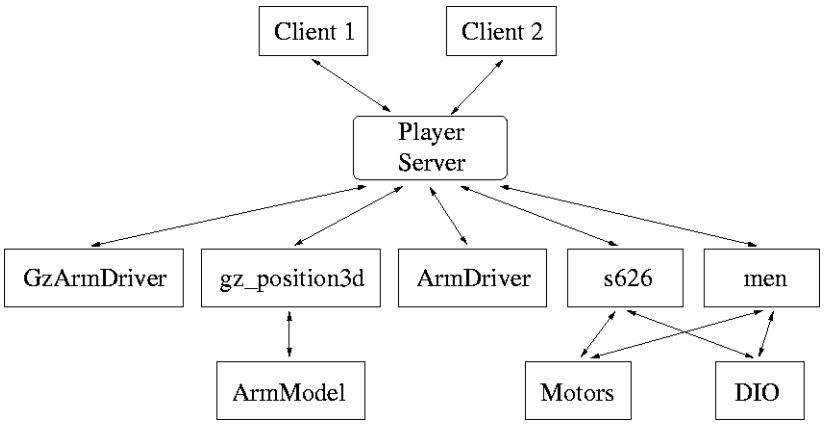

Figure 2: Principle sketch of communication through the Player server

2.3 How to Use Player

When starting Player, a configuration file needs to be specified. There can be a number of things specified in these files, for example some geometric informations about the model. The important part, is the declaration of the drivers, since those will tell the server where the commands from the client should be sent. There made two sample configuration files, one for the real arm and the other one for the model, which you can find on the CD-ROM. These files are documentated to explain what is going on.

To start Player, you have to type the following command in the directory where your configuration file is:

```
player configfile.cfg
```

Once Player runs, you can start your client program.

3 Gazebo

3.1 Introduction

Gazebo is an opensource program to create 3D models. It is made to be used with Player. With Gazebo one can simulate a population of robots, objects and sensors. Gazebo generates physically plausible interactions between objects. It was created to simulate mobile robots, or at least movable objects. This is a bit of a problem when trying to build a fixed model, like the robot arm. More information about this can be found in section 5.

Gazebo is based on the Open Dynamics Engine (ODE) library. This is an open source library for simulating rigid body dynamics. It uses OpenGL for the drawing.

3.2 Physically Based Simulations

Gazebo tries to respect the rigid-body physics. For example:

- Two solids can't stand on the same spot.
- When two solids collide, the reaction forces are computed. A body can't penetrate another one.
- Different joint types are available.

This allows to build 3D environments where the robot can move and act as if it was in the real world.

3.3 How to Use Gazebo

To create a new Gazebo model, the best thing to do is to look at already existing models to see how they were built. You can find some examples in /gazebo/server/models/. More details about creating a new model can be found in section 5.

To use Gazeo you need a world file describing your environment. In there, you

can define the cameras and their position, the models you want in your environment and a lot more. There are some examples of world files on the CD-ROM. To start Gazebo, you have to type the following commans in the directory where your world file is:

```
wxgazebo worldfile.world
```

Instead of calling `wxgazebo`, you can also call `gazebo`. This computes the model without displaying the model. It is faster than `wxgazebo`, and can be used for debugging. But once the simulation seems to work, it is better to use `wxgazebo` to really see what is going on.

Depending on what interfaces your model supports, you can already control your model now with the "Gazebo control box" (this works for example for a position3d interface). If you want to start a client to control the model, you have to start Player:

```
player -g default configfile.cfg
```

And now, once Player runs, you can start your client program.

4 Real Arm

4.1 Introduction

The first part of the bachelor project was to write a driver for the real arm. This needed to be done at the beginning, such that another student could use it when starting to work on the real arm to get the chess-playing part to work. It was good to start with this, because it made me look at the Player documentation thoroughly. It was a big help for the rest of the project to know where to look for what in the documentation.

4.2 Driver

4.2.1 Structure of the Driver

The client sends a command with which it tells the Player server how the arm should move. The server sends the commands to the right driver, in this case `ArmDriver` (see Figure 2). The job of the driver is to get those commands, to interpret them right and to send out commands in joint space coordinates which can be sent, through the Player server, to the `s626` driver. To be able to do this, the `ArmDriver` needs to know the configuration of the arm and it needs to keep track of how the arm moved. Therefore, the driver first has to be initailized. Then it periodically executes the following steps:

- check if the configuration changed
- check the commands from client, modify them and send the new commands on to the the server
- update the data

Since the `ArmDriver` does not directly talk to the motors, there was no need to implement a new controller. This is already done in the `s626` driver. The documented code for the `ArmDriver` can be found in the file `arm_position3d.cpp`.

4.2.2 Kinematics

The arm can be run in different modes. We can send either position commands or velocity commands. We can also choose if we want to send the commands in jointspace or in taskspace.

The kinematics equations are deduced from the theory in the class "Introduction to Robotics & Mechatronics". There are some new boundary conditions to make sure that the arm can't reach a point from where it can't be moved away anymore (kinematic singularity). This is an issue when working in taskspace coordinates, since the Jacobian is not allowed to be zero for the conversion of velocities form taskspace to jointspace.

Since the kinematics are not only needed for the real arm, but also for the simulation, they are implemented in separate files which can be included wherever they are needed.
The documented code for the kinematics can be found in the file `kinematics.cpp`.

5 Gazebo Model

5.1 Introduction

To add a new model to Gazebo, it is the best to create a plugin model. There is not much documentation on the internet on how to do this, but there is a simple example of a plugin model in the Gazebo folder [3]. The best thing is to copy this model and to change it to get the model we want. Another good thing to do, is to look at the already existing static models. They can be found in gazebo/server/models. They give a good insight into what can be done with Gazebo.

5.2 Concept of the Model

The model should behave just like the real arm when the client is tested on it. The model should use the same kinematics as the real arm. So there are two problems to be solved:

- Create a model that has the same degrees of freedom as the real arm.
- Find out how the commands have to be passed to the model.

5.3 Model

5.3.1 Geometry

Since drawing the model takes a lot of computation time, the model was simplified as much as possible with the idea that one could still put a skin on it to make it look nicer. To have a model that behaves the same way as the real arm, there have to be six independant parts in the simulation:

- base
- first and second link
- gripper base
- left and right gripper finger

[3] gazebo/examples/plugins/ExampleModel

There are predefined geometries that can be used to create a new model. In this porject, boxes are used to represent the objects. In the file `ArmModel.cpp` there is a function that draws the base, another one to draw the links and yet another one to draw the gripper.

Once the drawing is done, the pieces have to be set to the right position. Since Gazebo makes a physically based simulations of rigid bodies, there is also some collision detection. If two bodies lay on the same spot they start jumping around (due to the gravity). So setting the relative positions of the objects right is important. But even with setting those positions, the links will fall on the floor and start bouncing around because of gravity and the base starts shaking until it falls over.

Another idea was to try to somehow fix the base to the ground. This didn't work. Gazebo is intended to be used for simulations of mobile robots, not for bodies that should not move. There are tow different cases in which the arm could be used:

- arm fixed on a mobile robot
- arm standing alone on the ground: The base is stuck to the ground. So the base is actually part of the environment, not of the arm.

With this distinction and with additionally setting gravity to zero, the base stays upright without shaking and the links don't fall down anymore.

There seems to be a bug with the arm standing on the robot. The arm seems to stay on the right spot, if there is no skin. When a skin is added, the arm doesn't stay fixed to the same spot on the robot when this one is moving. Some more comments about this problem are in the file `ArmModel.cpp` in the function `DrawBase`.

The positioning of the links and the gripper is done with respect of the position of the base. There are two functions for positioning: `SetLinkPos` and `SetGripperPos` in `ArmModel.cpp`. The most important things to keep in mind

when looking at those functions are:

- The position of a body is always given by the position of its center of mass. When a body is made of several parts, you have to give the position of the center of mass of the first part of the body.

- The origin is chosen on the ground, in the surface center of the base.

5.3.2 Joints

In the next step the joints are added. There are four hinge joints and one slider joint that need to be implemented. You can see them in Figure 3. There are a few things one should know to set the joints right in Gazebo:

- Even if you set the anchor of a hinge joint at the end of an object, the object still rotates around its center of mass. Probably nobody tried to do something like this up to now. If you just need hinge joints to fix wheels, the axis will go through the center of mass of the object. To be able to turn around another place than the center of mass of an object, help pieces need to be added.

- A maximum force which can be exerted on a joint has to be set. If it is not set, the joint can't be move. Probably the default value is set to zero somewhere. If it is set too high, it starts behaving strangely when you want to set small angles to move the gripper fingers. The force seems to be too big for the small movement, and the object starts "flying" around. This explanation doesn't satisfie me completly, since, for example, the joint doesn't behave the same way when the gripper is opening or closing. So why would the force be too large to close the gripper, but not to open it?

- With a small maximum force, the gripper can be opened and closed without the joint being ripped appart. But it is still not perfect. The opening is good. But when the gripper is closed, the joints "oscillate" around their target positions.

- When you set the joint maximal force too low, you don't have enough grip to pick up the pieces anymore.

There are functions for this task too. On one hand `SetLinkJoints` sets the hinge joints for the two links. On the other hand `SetGripperJoints` sets two hinge joints and a slider joint for the gripper.

Now the geometry of the arm model is done. You can see it in figure 4.

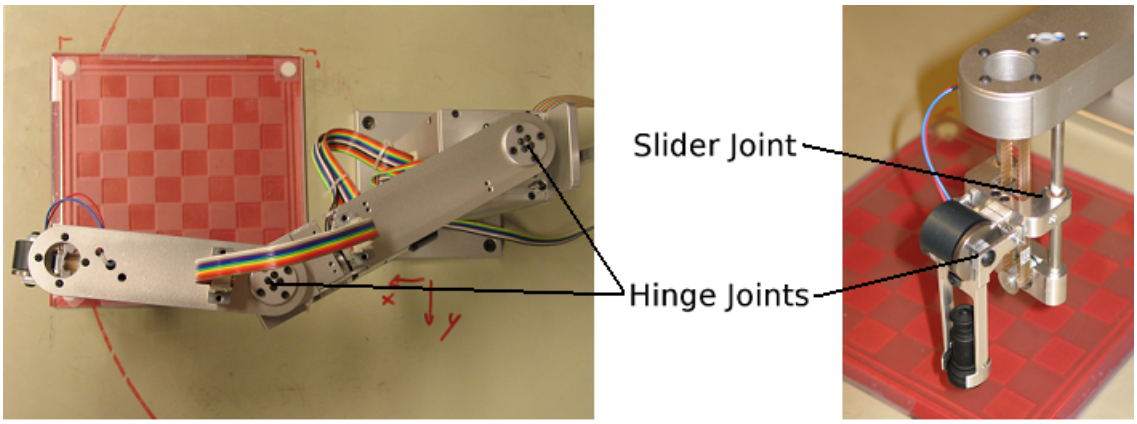

Figure 3: Joints in the real model that have to be implemented in the model

5.3.3 Skin

Once the model is stable, one can try to put a skin on it. There are several examples on how this works in existing models (/gazebo/server/models). The function `SetSkin` in `ArmModel.cpp` puts the skins on the bodies. It is important to know the following things:

- The size of the skin has to be scaled to match the model. There is no easy way to do this. So you have to use trial and error.
- The same holds for positioning the skin on the model. You save a lot of time if your skinfile is centered over the origin.
- If you use different cameras and see the skin in only one of them, you have to get the new version of gazebo via CVS to fix the problem.
- The collision detection is still made with respect to the model. Therefore, if the skin is a lot bigger than the object, it can overlap with another object without bouncing back.

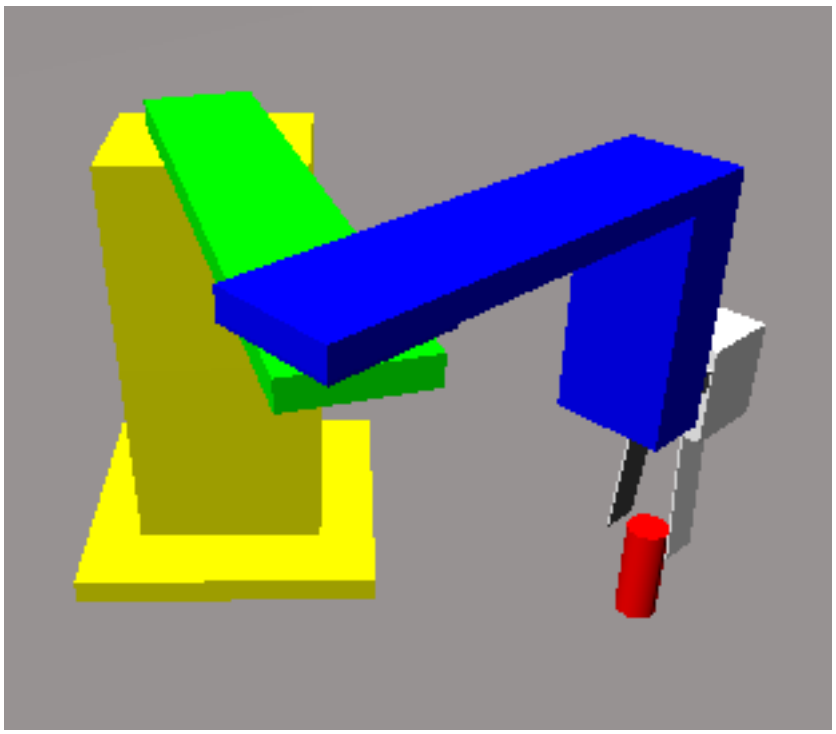

Figure 4: Arm model without skin

You can see the model with a skin in figure 5.

5.4 Commands to the Model

5.4.1 Send Commands to the Model

In the file where the geometry of the model is described, there also needs to be some code to get commands, refresh the data and send the commands to the joints. The details can be found in the file `ArmModel.cpp`. Here there is only a brief overview of what is needed.

To be able to get commands, there must be at least one interface. In this case, there are two of them: a position and a gripper interface. Similarly to what is described for the driver for the real arm, the model has to periodically check for new commands and update the odometry. It does it in this order:

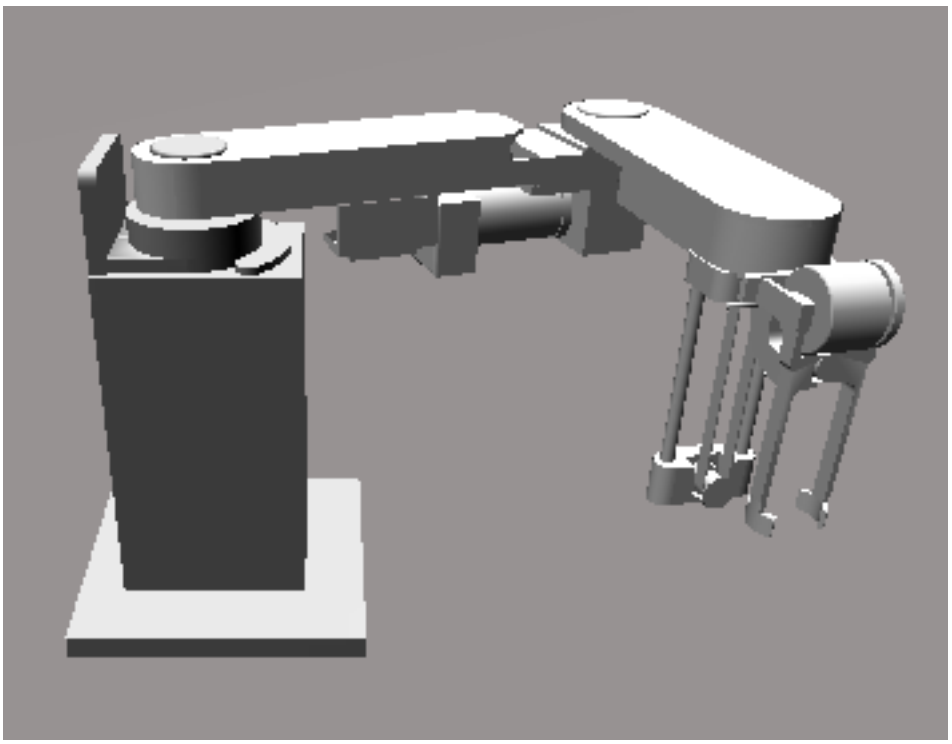

Figure 5: Arm model with skin

- update the odometry
- update the simulation time
- get the new commands
- interpret the commands and send them to the joints
- update position data

This works fine, but one can only get velocity commands from the position interface and one can also only send velocity commands to the joints.

5.4.2 Position Commands

Since the client should be able to use the Position3DProxy, the arm model has to be able to execute position commands. Therefore, the position commands have to somehow be transformed into velocity commands. To be able to do this, a new driver had to be written. The client sends the commands via the Player server to the driver (GzArmDriver in Figure 2). The driver then transforms the commands

to send out only velocity commands via the Player server to the gz_position3d driver. The gz_position3d driver then hands those over to the model. The advantage of adding a driver between the client and the model is that the code for the model can be kept short. There is no need to distinguish between jointspace and taskspace in there anymore, since all the commands it will get will be jointspace velocities.

The driver for the simulation is based on the same principle as the driver for the real arm. The only difference is that it does not send the commands to motors, but to the gz_position3d interface. To transform the position commands to velocity commands, a PID controller is used. More details about the implementation can be found in the code in the file `gz_arm_position3d.cpp`.

The 3D model takes a lot of computation to update. If too many cameras are used to look at the model, if there are too many objects, or too many complicated skins, then the model doesn't move smoothly anymore. On one hand this does not look nice at all. On the other hand, it is also a problem for the positioning of the end effector. When the arm moves step by step, the chance is pretty high that it misses the target position several times, and we have to wait a long time until the end-effector reaches the desired positin with the PID controller. Of course, the PID parameters can be adapted, but if they have to be tuned each time a camera or an object is added to the simulation, it gets pretty annoying.

5.4.3 Gripper Commands

Since the gz_gripper interface supports all the functions from the GripperProxy which the client uses, the commands for the gripper can be sent along directly from the client to the model without having to pass through the driver.

6 Summary and Contributions

The arm which was used in "Introduction to Robotics and Mechatronics" now has a gripper as end-effector which can be used to pick up small objects. A new driver was written which allows to talk to the arm through Player. This has two major advantages:

- The arm is now plugged in the s626 box. But since Player also interfaces the "men" driver, the arm could be stuck on MARVIN [4] and be controlled there without having to write a new driver. The only thing that needs to be changed is the configuration file to start Player.
- Since Player can be used with Gazebo, the same client can be run on the 3D model as well as on the real arm.

A 3D model of the arm has been created with Gazebo. Even though it does not work perfeclty yet (see section 6.1), it can be used to test some client code:

- The positioning in the xy-plane works fine. There might be small position errors between the simulation and the model which will have to be adjusted when working on the real model.
- The moving of the gripper works too, except of the picking up of objects. Because of the effects of the physically based simulation, the piece flies away when the arm is moving. Probably the centripetal force gets bigger than the gripping force. When changing the forces, you have to watch when the shaking of the closed fingers starts again.

6.1 Future Work

For the moment, the PID controller parameters are tuned to be able to move the model without skin. It is hard to find good parameters for the skinned model. So a great thing to change, would be to find a way to reduce the computation time for the model.

There are still some problems with the gripping of objects. Maybe this problem cannot be solved with the current version of Gazebo, since there never was a

[4] Mobile robot developed at IRIS.

response to my question about this on the forum. The two following things could be tried to improve the gripping:

- Change the joints of the gripper finger from hinge joint to slider joints. This might work, since moving the gripper up and down is implemented with a slider joint, and works pretty well.
- Add a piece of "soft" material in the gripper, which would adapt to the shape of the piece. Then the contact surface would be bigger, and maybe the pieces would not fall down anymore.

Another nice thing to do would be to stick the arm on Marvin, to have a robot with a gripper-arm. It would also be nice to create a Gazebo model of Marvin to have a 3D simulation of Marvin that could be used for "Advanced Robotics & Mechatronic Systems".

Figure 6: Arm model with chessboard and pawn

References

[1] Institute of Robotics and Intelligent Systems ETH Zurich.
http://www.iris.ethz.ch.

[2] Open Dynamics Engine.
http://www.ode.org.

[3] The Player/Stage project.
http://playerstage.sourceforge.net.

[4] T. Demmig.
jetzt lerne ich LaTeX 2ε.
Markt + Technik Verlag, Germany, 2004.

A Abbreviations

DOF Degrees of Freedom

ETH Swiss Federal Institute of Technology

IRIS Institute of Robotics and Intelligent Systems, ETH Zurich

ODE Open Dynamics Engine

PID Proportional-Integral-Derivative (controller)

B CD-ROM

On the CD-ROM you can find the following code files:

- Real Arm
  - arm_position3d.h & arm_position3d.cpp
    These two files implement the driver for the real arm. They are located in /src/marvin/drivers/arm-mod/src
- Arm Model
  - gz_arm_position3d.h & gz_arm_position3d.cpp
    These two files implement the driver for the simulation. They are located in /src/marvin/drivers/arm-mod/src
  - ArmModel.h & ArmModel.cpp
    These files implement the Gazebo model of the robot arm. They are located in /src/marvin/gazebo/models/arm/src
- Kinematics
  - kinematics.h & kinematics.cpp
    These files implement the kinematics which are used for the real arm as well as for the model. They are located in /src/marvin/drivers/arm-mod/src

Besides the code files, you can also find some other interesting stuff:

- configuration files for the simulation and the real model in /src/marvin/drivers/arm-mod/src
- world files for the model in /src/marvin/gazebo/worlds
- client program to test drivers and model in /src/marvin/util/src
- skins for the model in /src/marvin/gazebo/models/arm/skins
- movies of the simulation and of the real arm in /src/marvin/gazebo/models/arm/movies
- pictures, powerpoint presentation...