Dawn of Kate's Gazebo Simulation

A Senior Project

presented to

the Faculty of the Computer Science Department

College of Engineering

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science, Software Engineering

by

Tracy Davies

June, 2011

Project Advisor: Dr. Franz Kurfess

Table of Contents

Table of Figures

1 Introduction

1.1 Overview

1.1.1 Problem/Motivation

ZAFH Servicerobotik does not have any 3D Simulation of their service robot, Kate. They only have a 2D simulation, that can detect the location of the robot on a plane. However, the real world is in 3D. Therefore 2D simulation is insufficient and too simplistic. It does not include the actual structure of the robot, sensors, or camera nor can it simulate how it interacts with the world around it.

1.1.2 Solution

Create a 3D model of Kate in a simulated world environment. The 3D simulation will allow testing of the robot and its algorithms without actually using the physical robot. It will also be used for visualization purposes in demonstrations.

1.2 Scope

To research using Gazebo for 3D simulation of Kate. This includes how to model and control Kate in the simulated environment. The research will be done over four months in Ulm, Germany at the University of Applied Science Ulm.

1.3 Previous Work

There are currently already robots that are controlled and simulated in Gazebo. These include the Care-O-Bot, the Pr2 and the Pioneer3dx. The Care-O-Bot and the Pr2 robots have controllable arms like Kate's Katana. While the Pioneer3dx is more of a base for robots to build on. Kate uses a Pioneer3dx for its base.

1.4 Background

Robot Operating System (ROS) is packaged libraries and tools to help create robot applications. It is based on Switchyard, created at Stanford and written by Morgan Quigley. Since then it has had numerous contributions from around the world. Willow Garage is now the main over seer of ROS. Willow Garage designed the Pr2 and its simulated version in Gazebo. They are major players in implementing Player into ROS. Gazebo is the 3D simulation part of the Player Project. They have since added many more plugins into Gazebo to help its interface between ROS and Gazebo [7,11].

1.5 Approach

Become familiar with Gazebo to be able to simulate Kate. There are tutorials that shows how to model objects in Gazebo on the ROS Wiki. After becoming familiar, analyze previous robots that are simulated in Gazebo. Robots like the Pr2 and Care-O-Bot are ideal because they have robotic arms with multiple degrees of freedom. They resemble Kate's build up with wheels at the base and controllable arms. Since both robots use similar arms and controller it would seem feasible that they could work for Kate. So controllers could be adjusted from the Pr2 or Care-O-Bot controllers to be used for Kate's Katana.

2 Requirements

Create a 3D simulation of Kate in a real world environment. It needs be controlled through either an interface, a terminal commands, or a joystick. This includes controlling the wheels and the Katana. The Katana's joints need to be controlled with angles. The main focus is to get a simulated the Katana as it is the most complex part of Kate.

3 Tools/Technology

The following are need for this project.

- Ubuntu 10.04
- Graphics Card and Driver[12]
- Gazebo
- ROS
- Wiki for Documentation and svn

4 Implementation

4.1 Installation

4.1.1 Ubuntu Installation

The computer needs to have a compatible operating system before installing Gazebo and ROS. ROS is mainly targeted for Ubuntu. Though it can be installed on other operating systems, it is not recommended [1]. Installed Ubuntu 10.04 on computer. Gazebo and ROS are not included in Ubuntu.

4.1.2 ROS Installation

Gazebo is packaged with ROS, therefore it is needed to installed Gazebo. Administrated permission is needed for installation and configuration. Installed ROS C Turtle [3]. C Turtle is not the most recent distribution, however it was the most stable and recommended by Willow Garage [11]. The following commands install ROS C Turtle [1].

sudo sh -c 'echo "deb http://code.ros.org/packages/ros/ubuntu lucid main" > /etc/apt/sources.list.d/ros-latest.list'

Is the command to setup sources.list to accept packages from the ROS server.

```
wget http://code.ros.org/packages/ros.key -O - | sudo apt-key add -
```

Sets up your keys.

```
sudo apt-get update

sudo apt-get install ros-cturtle-pr2all
```

Re-indexes the ROD.org and installs the cturtle packages.

```
echo "source /opt/ros/cturtle/setup.bash" >> ~/.bashrc

. ~/.bashrc
```

Adds the ROS environment variables automatically to your bash. Without the ROS environment variable added to .bashrc Gazebo will still not run.

4.1.3 Gazebo Installation

Gazebo is not automatically installed with ROS though it is included in the ROS packages. It needs to be independently installed with its dependencies before being able to run. This could take awhile if the dependencies have not been previously built [14]. Before installing Gazebo though make sure that the computer has a Graphics card with OpenGL [12]. The following are the terminal commands to install and make Gazebo:

```
rosdep install gazebo_worlds

rosmake gazebo_worlds
```

There should be no errors in the installation and compiling of Gazebo. To test Gazebo was installed correctly open a new terminal and run the following command:

```
roslaunch gazebo_worlds empty_world.launch
```

It should start Gazebo with an empty world that just contains a gray tile plane and a blue cloudy sky. This is the base for adding models into Gazebo.

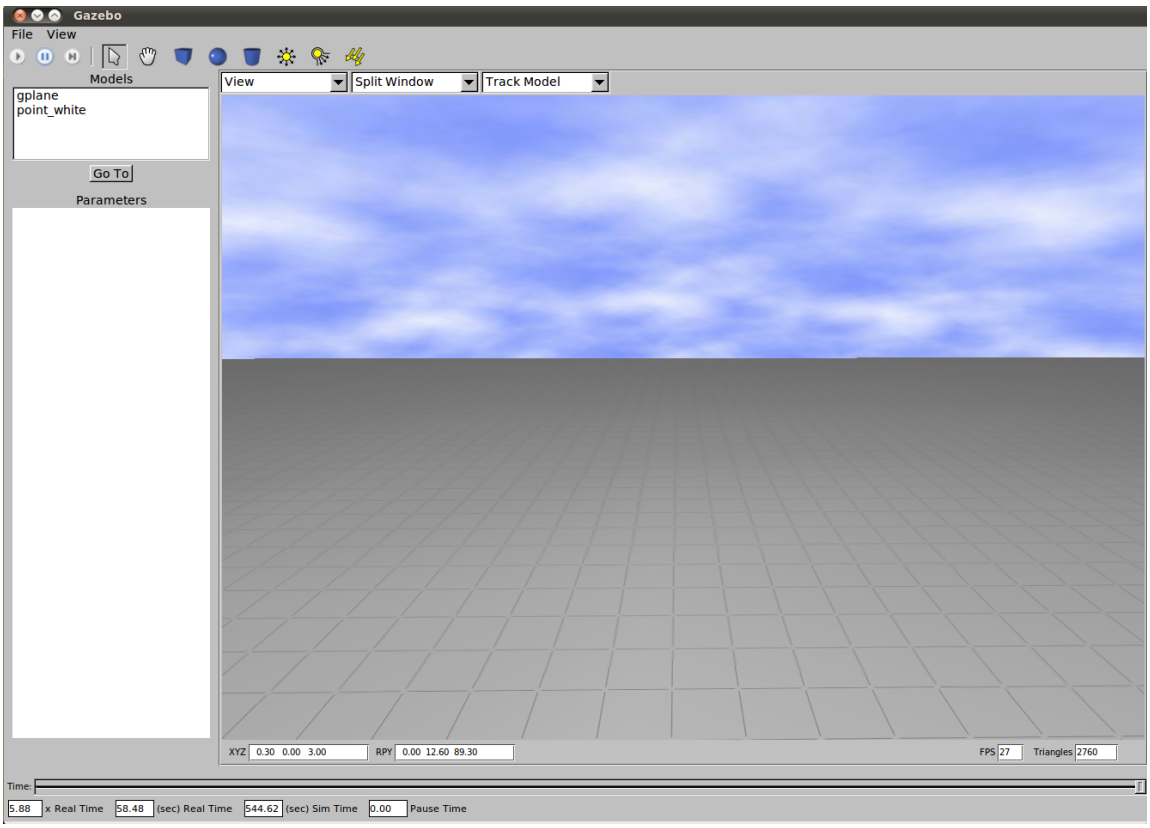

Figure 1: Example of a Gazebo empty_world

If there is no file in gazebo_worlds, try rosmake gazebo_worlds. If Gazebo does open but there is no sky and only a plane the mouse will move the view of the world around. Gazebo is running but has not models in it. It is an empty world.

4.2 Creating Simple URDF

Gazebo opens with an empty world, there are no objects simulated in the world. Models need to be create so that they can be loaded and used in Gazebo. Models are in URDF. URDFs contain information on the object that it models. It is in an XML format containing a tree of links and joints. It includes the inertial, visual and collision knowledge of the object being model.

Open gedit to start create a URDF file. URDFs can be created in any text editor. Copy and paste the following code into gedit to create a simple box URDF model.

```
<?xml version="1.0" ?>


  <link name="my_box">
    <inertial>
      <origin xyz="2 0 0" rpy=”0 0 0”/>
      <mass value="1.0" />
      <inertia  ixx="1.0" ixy="0.0"  ixz="0.0"  iyy="100.0"  iyz="0.0"
izz="1.0" />
    </inertial>

    <visual>
      <origin xyz="2 0 0" rpy=”0 0 0”/>
      <geometry>
        <box size="1 1 2" />
      </geometry>
    </visual>

    <collision>
      <origin xyz="2 0 0" rpy=”0 0 0”/>
      <geometry>
        <box size="1 1 2" />
      </geometry>
    </collision>

  </link>

  <gazebo reference="my_box">
    <material>Gazebo/Blue</material>
  </gazebo>


</robot>
```

Figure 2: Code for a cube in URDF format.

Now to understand the URDF file and what each part of the code does.

```
<?XML version="1.0" ?>
```

The first line of the file helps with XML formatting. It will color code the URDF file.

```
<robot name="simple_box">
```

The “robot name” is the name of the model. It will be the name Gazebo uses for model indication.

Next in the code is the link. It holds all of the information that is needed to model the URDF. There are three parts of the link: inertial, visual, and collision.

```
  <inertial>
      <origin xyz="2 0 0" rpy=”0 0 0” />
      <mass value="1.0" />
      <inertia  ixx="1.0" ixy="0.0"  ixz="0.0"  iyy="100.0"  iyz="0.0"
izz="1.0"/>
```

There are three parts of Inertial: origin, mass, and inertia. The origin of inertial is where the center of mass is located. It does not need to be the same of the geometric origin. “xyz” represent the xyz offset and “rpy” represent the axis roll, pitch and yaw angles in radians. If the origin is not set, it defaults to the identity. The mass value is the mass of the object, it is measured in kg. The inertia is given by a 3x3 matrix. Only ixx, ixy, ixz, iyy, iyz, and izz are used though. Inertial is optional however if it is included the mass and inertia are needed.

```
    <visual>
      <origin xyz="2 0 0" rpy=”0 0 0”/>
      <geometry>
        <box size="1 1 2" />
      </geometry>
    </visual>
```

There are three parts of Visual: origin, geometry, and material. The origin is the origin of the geometric shape. The geometry includes the visual object. The geometry has to be either a box, cylinder, sphere or a mesh. A mesh is used to import an object from a filename. The material is the material of the visual object. It has a color of rgba (red, green, blue, alpha) and a texture defined by

```
  <material name="LightGrey">
    <color rgba="0.6 0.6 0.6 1.0"/>
  </material>
  <material name="RollLinks">
    <texture
filename="package://pr2_description/materials/textures/pr2_wheel_left.png"/>
  </material>
```

a file name. Gazebo has default materials that can be called with “Gazebo/Blue”. Materials can also be predefined to be reused through out the file or after the link in a reference like the example above. Visual is optional but the geometry is required.

```
    <collision>
      <origin xyz="2 0 0" rpy=”0 0 0”/>
      <geometry>
        <box size="1 1 2" />
      </geometry>
    </collision>
```

There are two parts of collision: origin and geometry. The collision can have different properties than the visual properties. The geometry in collision is often simpler than the visual geometry as it is not shown and can reduce the computation time [16].

To check the code is without errors before trying to model it run it through the URDF parser.

```
$ rosrun urdf check_urdf simple_box.urdf
```

The parser will say if the URDF file was successfully parser or if there are errors. Now have a

simple model to load into Gazebo but not how to spawn it into Gazebo.

4.3 Spawning URDF

Gazebo opens with just a world, no objects loaded. Gazebo does not need objects in the world to run though. To load a URDF into Gazebo open a new terminal, go to the folder where the file is located and use the following command:

```
rosrun gazebo spawn_model -file simple_box.urdf -urdf -z 1 -model my_object
```

ROS runs Gazebo for spawning a model. It loads the file simple_box.urdf[1] into Gazebo with the model name "my_object". The object is spawned at a height of 1 meter, "-z 1". Make sure that

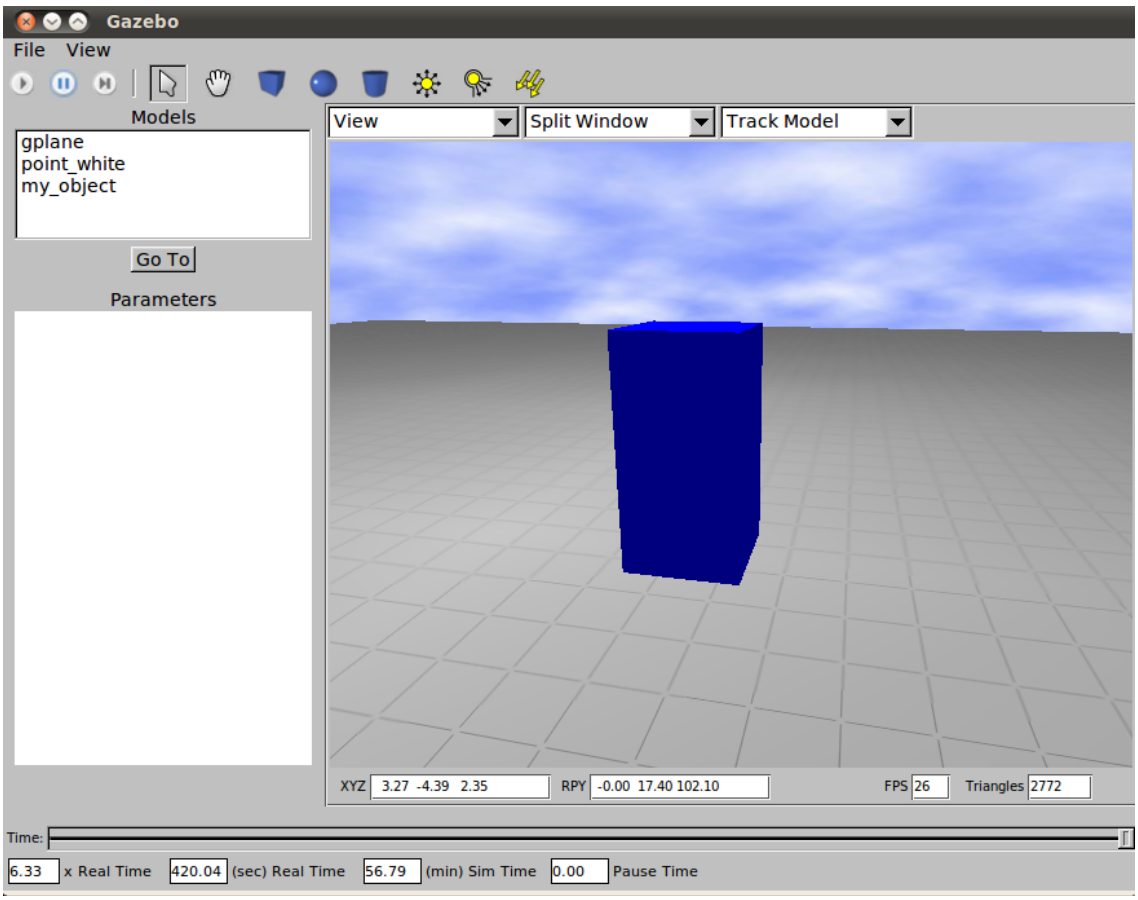

Figure 3: Gazebo with simple_box.urdf

Gazebo is already running with an open world. Loading models will not open Gazebo, only a world can. Loading a model will not give an error though if there is no Gazebo world. It will just run without opening Gazebo. If this happens cancel the model and load a Gazebo world and then the model. Once the model is loaded in the Gazebo the following command will show the model status [13].

```
rosservice call gazebo/get_model_state '{model_name: my_object}'
```

This will show the position and orientation of the model as properties.

To spawn a URDF file in a terminal is complex to run command and remember even when loading a simple model. There might be a script that can be run or created. Most object simulated

1 Can find file in /Examples/my_examples/

are not made up of one geometric shape but are more complex models that have joints.

4.4 Scripting Spawning URDF

There are long complex terminal commands to spawn a Gazebo world and spawn multiple objects in to the world. There has to be a simpler or scripting language that can be used to make it easier to start Gazebo and models.

There are launch files that are used to spawn redefined objects into Gazebo, like a table or coffee cup. A list of predefined models can found in gazebo_worlds. To look at the gazebo_worlds package use:

```
$ roscd gazebo_worlds
```

Roscd is like cd but it looks through all of the ROS packages and goes to that path.

To explain launch files, the following example launch code is taken from table.launch[2].

```
<launch>
  <!-- send table urdf to param server -->
  <param name="table_description" command="$(find xacro)/xacro.py $(find
gazebo_worlds)/objects/table.urdf.xacro" />

  <!-- push table_description to factory and spawn robot in gazebo -->
  <node name="spawn_table" pkg="gazebo" type="spawn_model" args="-urdf -param
table_description -z 0.01 -model table_model" respawn="false"
output="screen" />
</launch>
```

Figure 4: Code from table.launch.

The launch files first parses the XACRO file into a URDF in the “table_description”. Then pushes that parameter and spawns the model into Gazebo. The second command is the same command given to the command line in the previous section.

To run the launch file Gazebo needs be running first. In another terminal the following command will run the the launch file and spawn the model into the Gazebo world.

```
$ roslaunch gazebo_worlds table.launch
```

The table will spawn in Gazebo. This launch file though uses a XACRO file instead of a URDF file and calling a Gazebo Worlds still needs to be done in another terminal.

2 Found in cturtle/stacks/simulator_gazebo/gazebo_worlds/launch.

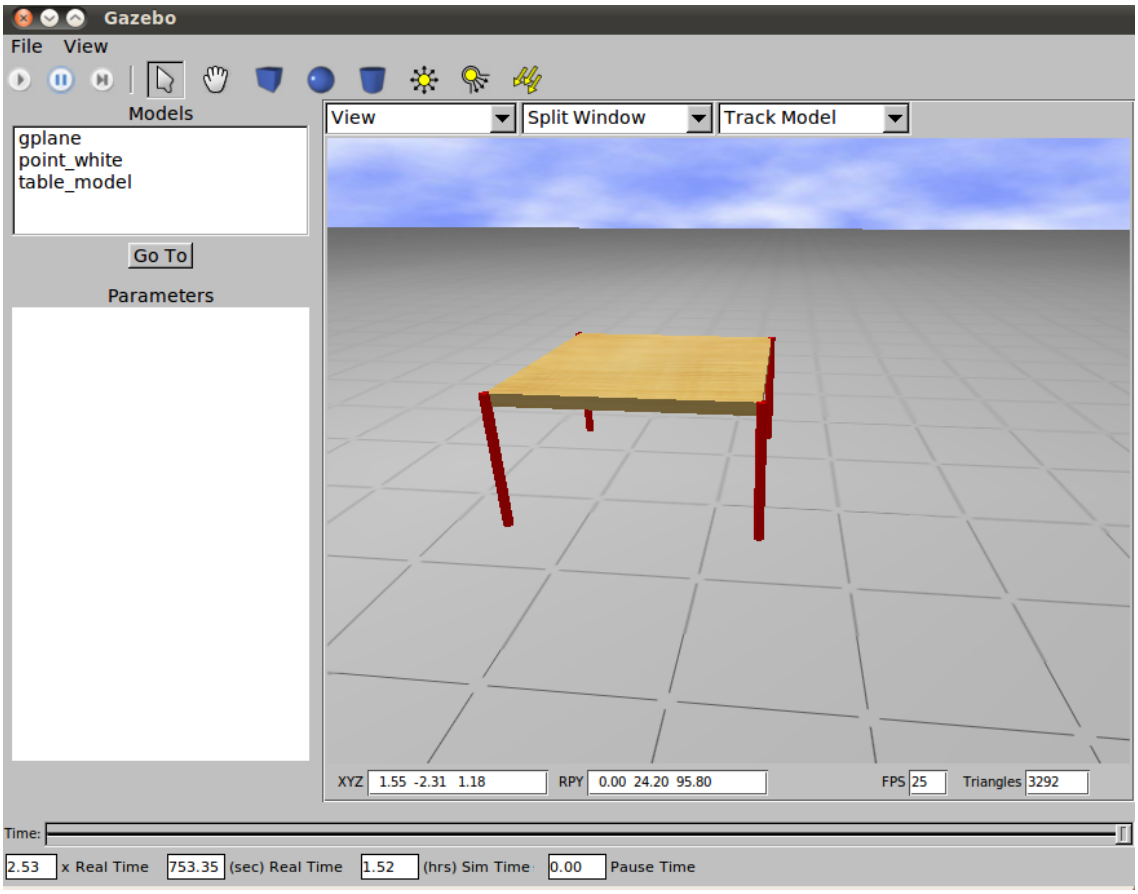

Figure 5: Table spawned in Gazebo with table.launch.

4.5 Launching URDF

The launch files that are used to launch predefined models in Gazebo use models in XACRO, not URDF. Have to find how to use launch files but with the parameter being a URDF file.

The parameter name in the launch files is a command to convert the XACRO to urdf format before using it in the following command to spawn it to Gazebo. Should be able to make the command be a URDF file instead. This does not work because it is not a command. So the parameter

```
<?XML version="1.0"?>
<launch>

  <param name="robot_description" textfile="simple_box.urdf" />

  <!-- push robot_description to factory and spawn robot in gazebo -->
  <node name="spawn_pr2_model" pkg="gazebo" type="spawn_model" args="-urdf
-param robot_description -z 1 -model my_object" />

</launch>
```

Figure 6: A launch file that launches simple_box.urdf.

needs to be changed to a textfile, look above. Then the launch file will launch the URDF correctly. Can now launch URDF files and XACRO files so what format is better to use for modeling and still

have to open Gazebo world first.

4.6 XACRO for Modeling

Gazebo predefined models are in XARCO not in URDF. They both model the object and have the same information included in the file. XACRO is an XML macro language. It can be used to create shorter and more manageable XML files than URDFs[17].

The following is an example of a XACRO file. It can uses variables and also equations instead of the developer.

```
<xacro:property name="width" value=".2" />
<xacro:property name="bodylen" value=".6" />
<link name="base_link">
        <visual>
                <geometry>
                        <cylinder radius="${width}" length="${bodylen}"/>
                </geometry>
                <material name="blue">
                        <color rgba="0 0 .8 1"/>
                </material>
        </visual>
        <collision>
                <geometry>
                        <cylinder radius="${width}" length="${bodylen}"/>
                </geometry>
        </collision>
</link>
```

Figure 7: Example of a XACRO file with one link [17].

XACRO is a type of XML macros that is useful with large XML files or projects. The Pr2 uses XACRO and not URDF to model itself.

4.6.1 URDF vs XACRO

Both URDF and XACRO files can be used to model objects in Gazebo. URDF files are simpler to understand. But XACRO have macros. These can be used to defined multiple variables and equations. This is helpful when more than one links have the same properties and dimensions, like table leg. In a URDF file every table leg would have to have its own defined properties and dimensions, like a cylinder radius and height. But in a XACRO they could all equal a variable.

Figure 8: A flow chart of how the XACRO, URDF and launch files all interact.

Though it seems that XACRO are more useful and would be more common, in the end the XACRO is converted to a URDF before being modeled in Gazebo. This conversion adds computation time. When a launch file is used to load a XACRO it converts the XACRO into a URDF file before uploading it in Gazebo, this is seen in the above chart. However, the conversion can also be done in terminal commands so the URDF is separate and can be looked at later. The following converts the Pr2 XACRO file into a URDF.

```
rosrun xacro xacro.py `rospack find pr2_description`/robots/pr2.urdf.xacro
-o /tmp/pr2.urdf
```

After converting a XACRO file into a URDF file it can be check with the URDF parser, from **Creating Simple URDF.**

Since either URDF or XACRO files can be used to model objects in Gazebo. It is more of a developers choice to which they prefer. URDF is better for understanding the link and joints relations. But, it can have repeated code within the links. While XACRO files are better for larger XML files and models with the macros available. All example except in this section are URDF files, not XACRO files. This is due to URDF file's structure being more standard than XACRO files and its clearly defined links and joints.

4.7 Spawning the Model and the World

Gazebo examples of launching models still need the world to be started in another terminal before running the launch file. However launch files can include other launch files. The following is an example of a launch file that first launches an empty world and then a box.

```
<?XML version="1.0"?>
<launch>

  <!-- This is in a default in gazebo for an empty world-->
  <include file="$(find gazebo_worlds)/launch/empty_world.launch"/>

  <!-- This will launch just an arm in Gazebo. It will not be attached to anything -->
  <include file="$(find kate_description)/launch/box.launch" />

</launch>
```

Figure 9: A launch file that calls empty_world.launch and box.launch.

Can not use a launch file that includes a world when Gazebo is already running. It will close the other Gazebo and not model your object. Launching the Gazebo world and the model at the same are not always a good idea. For testing reasons it is better to start Gazebo world and then spawn the model. The errors are easier to see when the world is already launched.

4.8 Creating Complex URDF with Joints

Most models are not just a simple geometric shape but are made up of many. Gazebo does this in URDF files by having links and joints. The links are the geometric part and the joints content them. The URDFs links and joints are in a tree that connects all of them to a base link. All origins are based off of the base and the parent links.

Create a simple two links and a joint URDF to see how the joint interacts with the links. To create a URDF and links look at **Creating Simple URDF**. The following joint was taken from simple_joint_orig.urdf[3]. It includes all of the information need to create a joint. It does not include

3 File can be found in /Examples/my_examples/

```
<joint name="my_joint" type="continuous" >
  <!-- axis is in the parent link frame coordinates -->
  <parent link="base_link" />
  <child link="my_cylinder" />
</joint>
```

Figure 10: Basic joint with only required information.

any optional information.

To make sure that the links and joints make a correct tree, first run the URDF file through the parser, look at **Creating Simple URDF.** When the code compiles correctly used:

```
rosrun urdf urdf_to_graphiz my_urdf.xml
evince test_robot.pdf
```

to get a visual of the joint and link tree. It will create a pdf of the URDF's tree. This will help with more complexer models that have multiple joints that branch off. The tree also includes the origin of

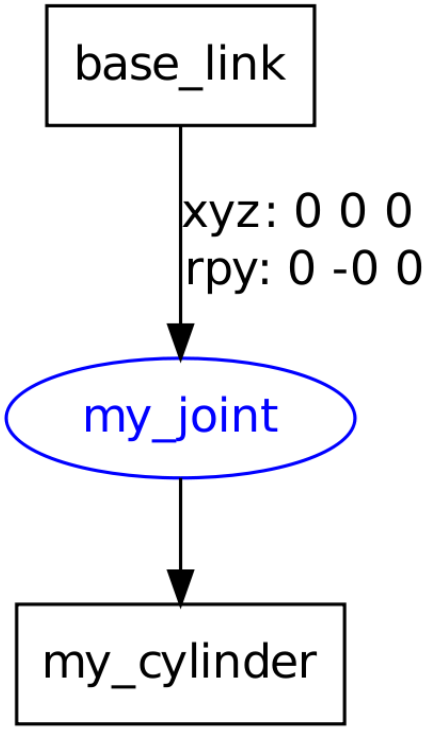

Figure 11: Link Tree for simple_joint_orig.urdf.

joint with its xyz and rpy. Since simple_joint_orig.urdf has no origin defined for the joint, it is set to the default.

Though the links and joints match up in the URDF file it does not mean they will when the model in loaded in Gazebo. In fact since the joint was not given a origin the child's origin was used. This makes the model depend on the child's link. But the child's origin changes with the joints

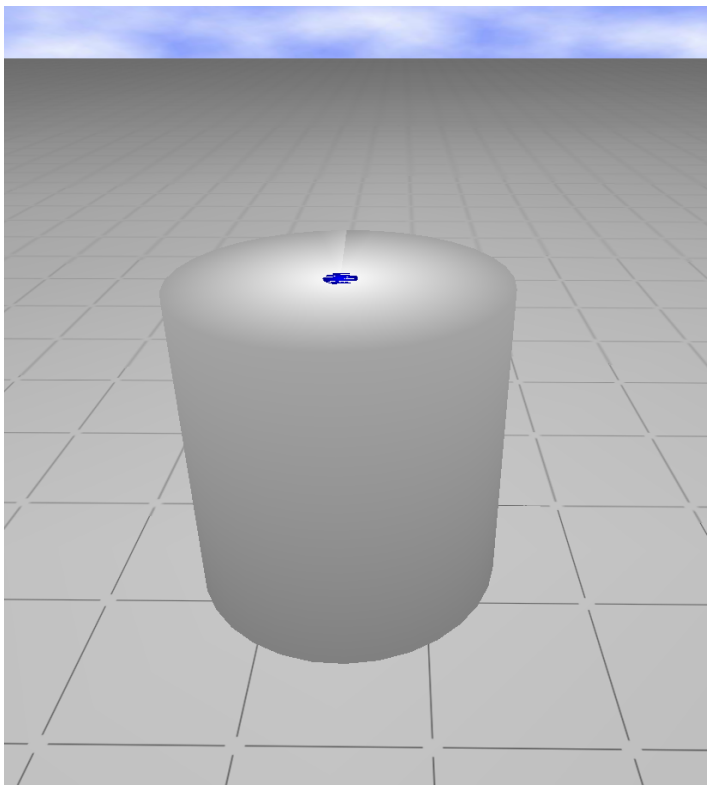

Figure 12: Gazebo model of simple_joint_orig.urdf

origin too. The previous Figure show the model with two links and a joint where the child and the joint's origin are set to default. A simple joint with the requirements is not helpful without its other properties. However what are the properties that can be included in a joint.

4.8.1 Joints

All that is required for a joint is the name, type, and parent and child links. However, more information can be included in the joint. The following is a complex joint that includes optional attributes [15].

```
<joint name="my_joint" type="floating">
   <origin xyz="0 0 1" rpy="0 0 3.1416"/>
   <parent link="link1"/>
   <child link="link2"/>
   <axis xyz=”0 1 0”/>

   <calibration rising="0.0"/>
   <dynamics damping="0.0" friction="0.0"/>
   <limit effort="30" velocity="1.0" lower="-2.2" upper="0.7" />
   <safety_controller k_velocity="10"/>
</joint>
```

Figure 13: Example code of a complex joint.

A joint is made up of attributes and elements. The attributes include the name and type of the joint. There are six different types of joints: revolute, continues, prismatic, fixed, floating and planar [15]. Revolute joints are hinge joints, they have limits to their rotation and rotate around a certain axis. Continuous joints are hinged joints that continue rotates around the axis. Prismatic joints slide along an axis in a limited range. Fixed joints do not move but connect two links. Floating joints

have all six degrees of freedom free. Planar joints move along a plane perpendicular.

```
<origin xyz="0 0 1" rpy="0 0 3.1416"/>
<parent link="link1"/>
<child link="link2"/>
<axis xyz=”0 1 0”/>
```

The origin of the joint is defined from the parent link. The parent and child links need to be links names. Make sure not to switch the parent and child links. A parent can have many child but a child can only have one parent. The axis is either the rotation axis, the axis of translation, or the surface normal depending on the type of joint. It is not always needed, i.e. floating joints. The axis default is set to “1 0 0”.

```
<calibration rising="0.0"/>
<dynamics damping="0.0" friction="0.0"/>
<limit effort="30" velocity="1.0" lower="-2.2" upper="0.7" />
<safety_controller k_velocity="10"/>
```

The calibration is used to calibrate the position of the joint. It has a rising and a failing. They are triggered when the joint moves in a position direction. The dynamics if the joint is for simulation modeling. It has damping and friction. They are the physical values that are used in physics. The joint has limits: lower, upper, effort, and velocity. The upper and lower are radian values of what the joint is allowed to move. The effort and velocity are the maximum applied effort or velocity allowed. If the joint includes a safety_controller the k_velocity is required but the soft_lower_limit, the soft_upper_limit, and the k_position are optional. They default to 0 if not given a value.

To see if the joint works, rosservice can be used to apply a joint effort. This is useful to see if the joint moves correctly without having a controller running at the time. The following commands are for applying a joint effort of 0.01 N/m to a joint called link_joint and then to stop and clear the effort [13].

```
rosservice call gazebo/apply_joint_effort '{joint_name: link_joint,
effort: 0.01, start_time: 10000000000, duration: 1000000000}'


rosservice call gazebo/clear_joint_forces '{joint_name: link_joint}'
```

When a joint includes more information, the links are not modeled inside of each other. This can be seen in simple_joint.urdf[4]. The origin of the joint is at “0 0 2” so that the joint does not connect the two links in the model. Yet it seems that though the joint is continuous there is a limit. This limits is the physical pendulum swing limit. However, the joint and links still do not match up the way that is expected. The joint location is from the parents origin not the child's.

4 File can be found in /Examples/my_examples/

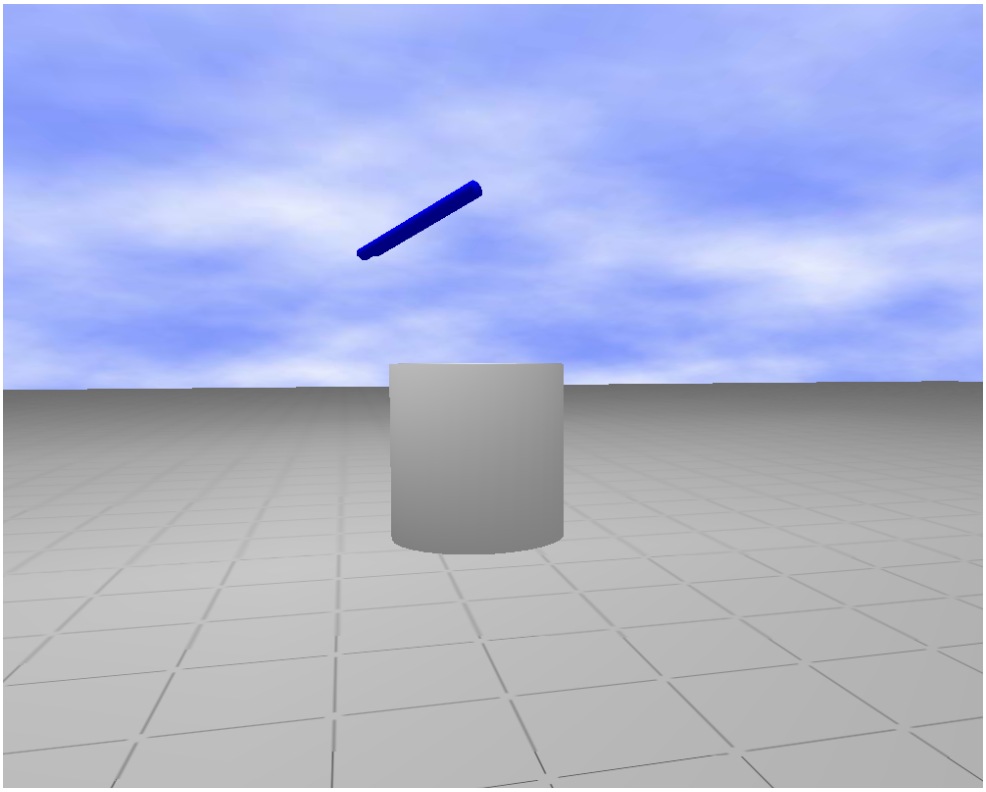

Figure 14: Gazebo model of simple_joint.urdf.

4.9 Modeling Simulated Robot

Gazebo can simulate geometrics and joints but can it simulate robots in a simulated really world. Examples of different robots from independent groups are given. Though they are given as example they may not run. There are two main types that are of interest: bases, like Erratic and Pioneer3dx and robots with arms, like Care-o-bot and the Pr2.

4.9.1 Modeling Base Robots

Gazebo has two examples of base robots: Erratic and Pioneer3dx. The Erratic is a basic model of a base. It has two main wheels, and a castor wheel. It is an unembellished Pionner3dx. Erratic has no meshes, just the Gazebo geometric and materials.

```
$ roslaunch erratic_gazebo erratic_wg.launch
```

Loads Erratic into Gazebo. If the package is not available make sure that ros-cturtle-erratic-robot is installed on the computer.

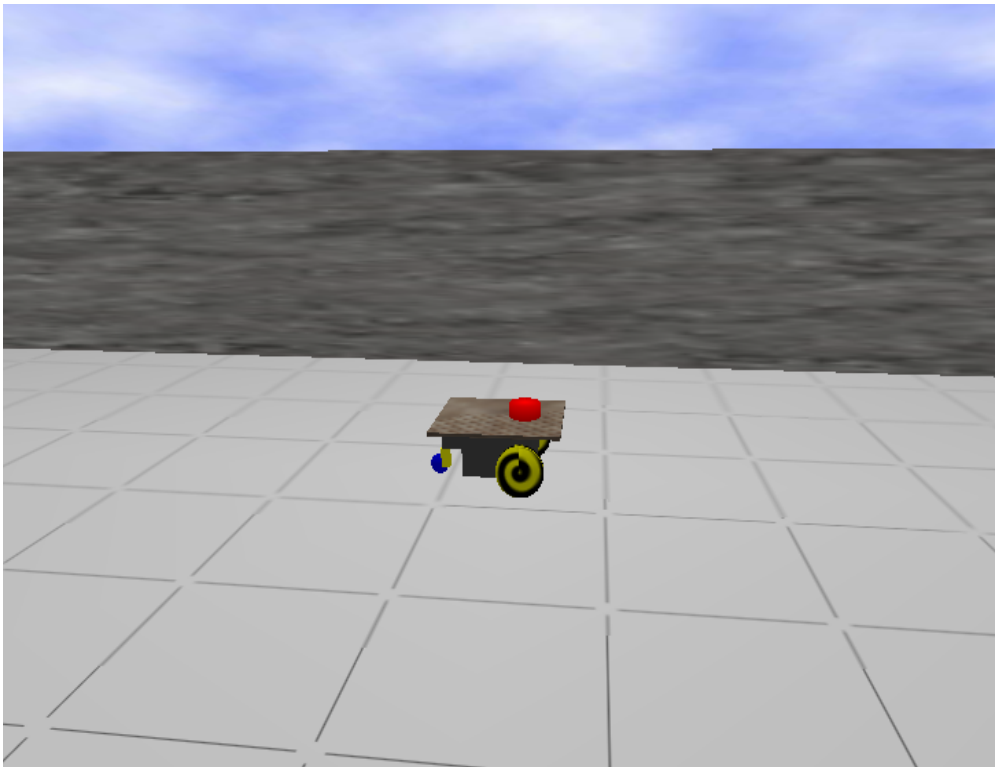

Figure 15: Gazebo model of Erratic.

The other robot base that is already model in Gazebo in a Pioneer. The base of Kate is a Pioneer3dx. If the predefined Pioneer in Gazebo will model, then the base does not need to be redesigned or simulated. Their Pioneer model can be added eventually be added on to simulate Kate.

```
$ roslaunch p2os_urdf pioneer3dx.gazebo.launch
```

Can be used to load a Pioneer model into Gazebo.

Gazebo can model Pioneer and Erratic models in Gazebo. However, they are stationary and only move when dropped from high heights. There has to be a way to control the wheels of the robots, so that they can move around the world.

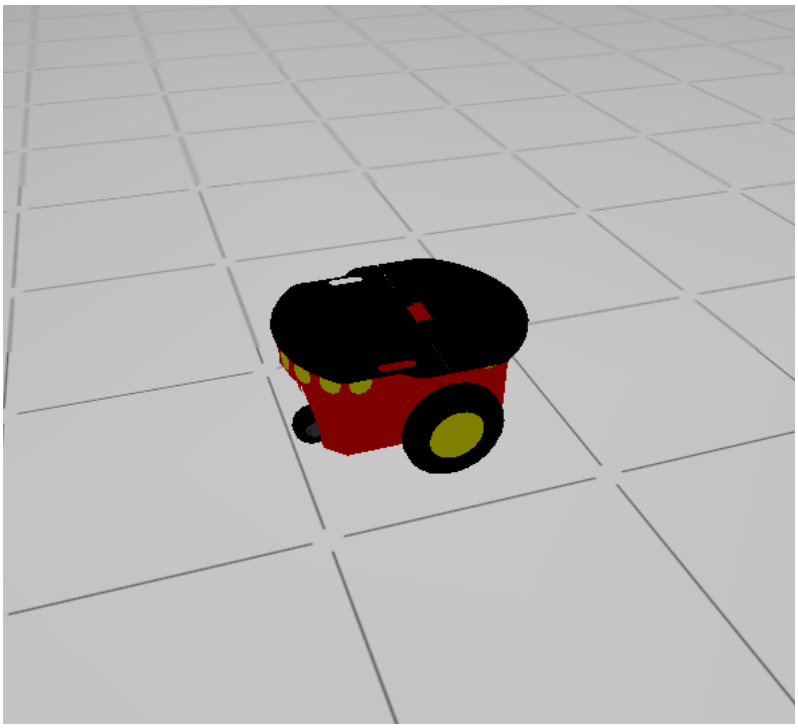

Figure 16: Gazebo model of Pioneer.

4.9.2 Modeling Robot with Arms

In Gazebo there are already models of two robots that have arms similar to the Katana, the Pr2 and the Care-O-bot. They each have an arm or arms that have between 5 and 8 degrees of freedom and a gripper at the end.

Before trying to simulate the Care-O-bot the package, ros-cturtle-care-o-bot, needs to be installed and compiled. The following command will compile the Care-O-Bot.

```
$ rosdep install cob_bringup
$ rosmake cob_bringup
```

To start the Care-O-bot in Gazebo use the following commands.

```
$ export ROBOT=cob3-1
$ export ROBOT_ENV=empty_world
$ roslaunch cob_bringup cob3-sim.launch
```

The first environment variable is what version of the robot to use, there is cob3-1 or cob3-2. The second environment variable is what the environment or world that will be used. There are many robot environments for the Care-O-Bot[5]. The last line opens Gazebo and models the Care-O-Bot.

5 Other worlds can be found ros/cturtle/stacks/cob_simulation/cob_gazebo_worlds

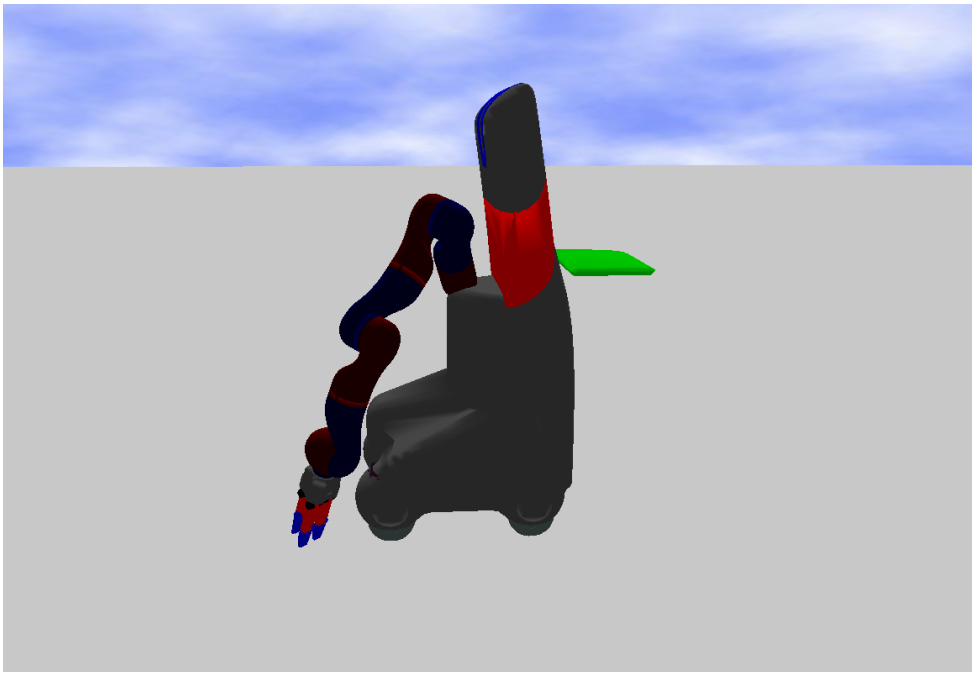

Figure 17: Gazebo model of the Care-O-Bot.

The other robot that has a similar arm to the Katana is the Pr2 [9]. The Pr2 is package with ROS that was installed since it is part of Willow Garage. To load the Pr2 into Gazebo use the following command:

```
$ roslaunch pr2_gazebo pr2_empty_world.launch
```

Between the Care-O-Bot and the Pr2, the Pr2 has the closest arm to the Katana. It has a two finger gripper and the arm without the shoulder have the five degrees of freedom. The Pr2 comes with controllers that can move the simulation with rostopics. Once the controllers are working in Gazebo it may be possible to just copy and paste the controllers and use it for the Katana.

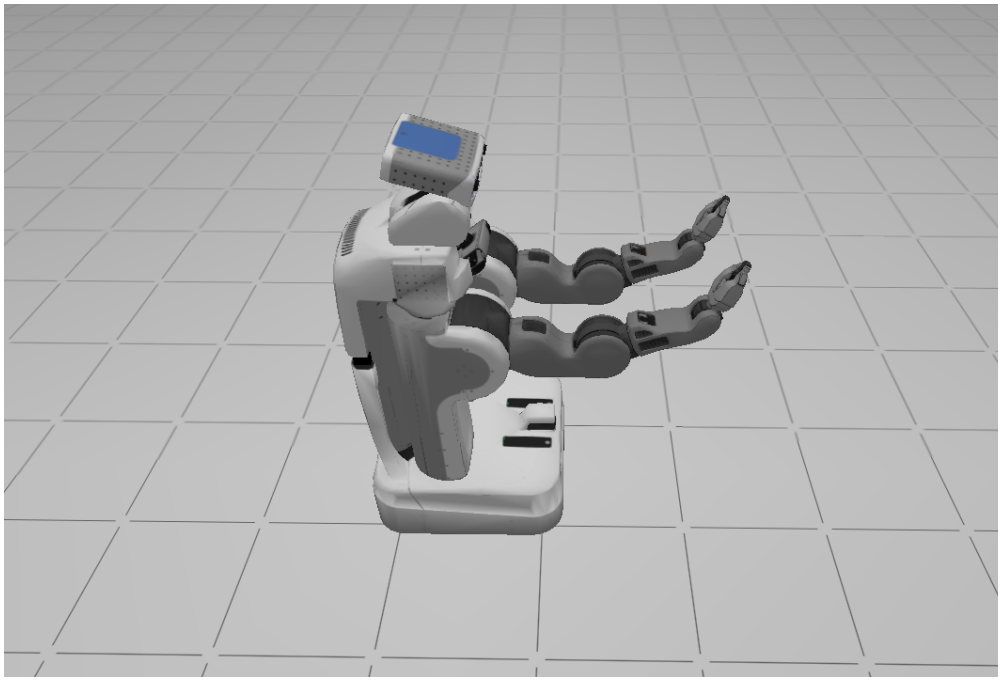

Figure 18: Gazebo model of the Pr2.

4.10 Controlling Pr2

The Pr2 comes with controllers implemented into the simulation. The controllers are rostopics and controlled with messages. When the Pr2 is loaded into Gazebo the controllers are already running. The controllers are a rostopics that runs automatically with the simulation. To see all of the rostopics that are running use:

```
rostopic list
```

It lists all of the running rostopics that are connected to the Pr2. The list includes controllers, cameras, stereos and more. There is r_gripper and l_gripper controls as well as r_arm and l_arm joint_trajectory controllers.

The controller can be controlled through messages from the command line. To command the right gripper to open find all listed rostopics for the r_gripper.

```
rostopic list | grep r_gripper
```

Before using r_gripper_controller/command to control the gripper check the type of messages that it accepts.

```
rostopic info /r_gripper_controller/command
```

There are three types of ROS messages: JointControllerState, JointTrajectoryControllerState and Pr2GripperCommand [5]. The following uses "pr2_controllers_msgs/Pr2GripperCommand". To see what the message includes use the following command.

```
rosmsg show pr2_controllers_msgs/Pr2GripperCommand
```

The message contains a floating point position and max_effort value, both of which are needed to control the gripper [6].

```
rostopic pub r_gripper_controller/command pr2_controllers_msgs/Pr2GripperCommand
"{position: 0.06, max_effort: 100.0}"
```

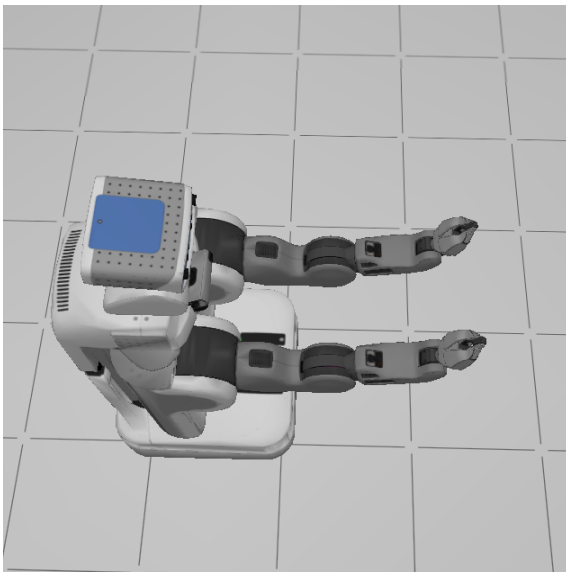

Figure 19: Pr2 before command.

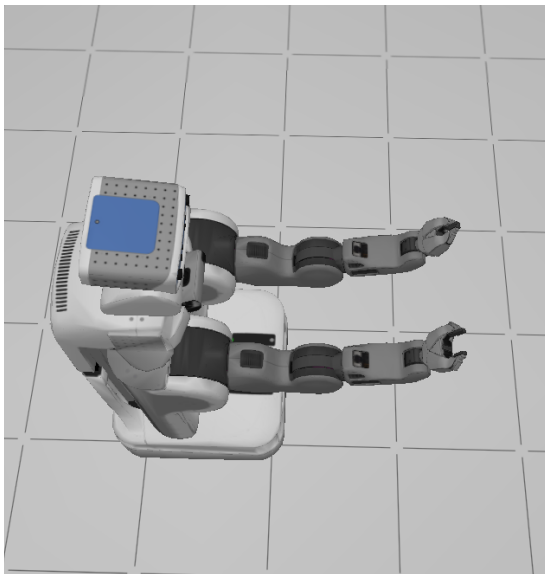

Figure 20: Pr2 after r_gripper command.

When trying to control the /r_arm_controller/command, it is a trajectory_msg/JointTrajectory. It is not the same type of message as the gripper controllers. So the

message may take different information. When showing what the message excepts it seems to be much more complicated. Not only that but it asks for arrays of variables. Neither the array sizes are listed nor what the controller thinks the arrays sizes are. There is no way to tell what the arrays need to include at this time. Tried guessing what the arrays would be but the Pr2's arm does not move.

```
Header header
  uint32 seq
  time stamp
  string frame_id
string[] joint_names
trajectory_msgs/JointTrajectoryPoint[] points
  float64[] positions
  float64[] velocities
  float64[] accelerations
  duration time_from_start
```

Figure 21: The information needed for a trajectory_msg.

It seems that using the Pr2's arm controller is not as easy as copy and pasting. The Pr2 has a controller for each of their grippers and arms. It may be possible to take one of the gripper controllers and simplify it to be able to use it for a controller for the Katana gripper. Since the gripper controllers was working, is easier to understand, and controls one joint at a time for each controller it may be able to just get that controller running.

4.10.1 Just Pr2 Gripper Controller

The gripper controllers for Pr2 works but they are not the only controllers running when the Pr2 is started. There are many other rostopics and controllers. However most of the rostopics are not needed for controlling the gripper.

The following code is based off of the Pr2's gripper controller and updated to only control the left gripper. The file l_gripper_controller.launch only launches the l_gripper controller of the Pr2.

```
<launch>
  <!-- Gripper -->
  <rosparam command="load" file="l_gripper_controller.yaml" />

  <!-- Controllers that come up started -->
  <node name="default_controllers_spawner"
        pkg="pr2_controller_manager" type="spawner" output="screen"
        args="--wait-for=/calibrated l_gripper_controller" />

  <group ns="l_gripper_controller">
    <node name="gripper_action_node"
          pkg="pr2_gripper_action" type="pr2_gripper_action" />
  </group>

</launch>
```

Figure 22: Code is from kate_controllers/l_gripper_controller.launch.

The test.launch file launches the Pr2 model in Gazebo. However before that can even loaded an empty Gazebo world needs to be open. All of the launches are done separately for in-dept testing.

Thus the l_gripper_controller is not included in the test.launch[6], which loads the Pr2.

To test if the l_gripper_controller only starts that controller, the following is needed first to start the controller. It is not connect with the Pr2, so it is not automatically started.

```
roslaunch l_gripper_controller.launch
```

And to see if the controller still works by itself.

```
rostopic pub l_gripper_controller/command pr2_controllers_msgs/Pr2GripperCommand
"{position: 0., max_effort: 100.0}"
```

When rostopics is commanded, the left gripper controller is now the only one running and available. The l_gripper_controller now only controls the left gripper though the whole Pr2 is being model. Since the gripper is only part being controlled the rest should be removed from the model. So that the model is only the gripper with the gripper controller.

4.11 Simplifying Pr2's Arm

The l_gripper controller only needs the arm to be controlled, the rest of the Pr2 can be taken away. Thus only the arm and its controller would be left. This can be used to fully understand how the URDF model and the controller interact.

It is not as simple as making the shoulder link the base link. The base link has other information in it that can not be deleted. When the base_link is remove the model no longer loads, it has an error. There seems to be joint on top of each other in the model. However, they are in a tree of joints and links in the URDF file.

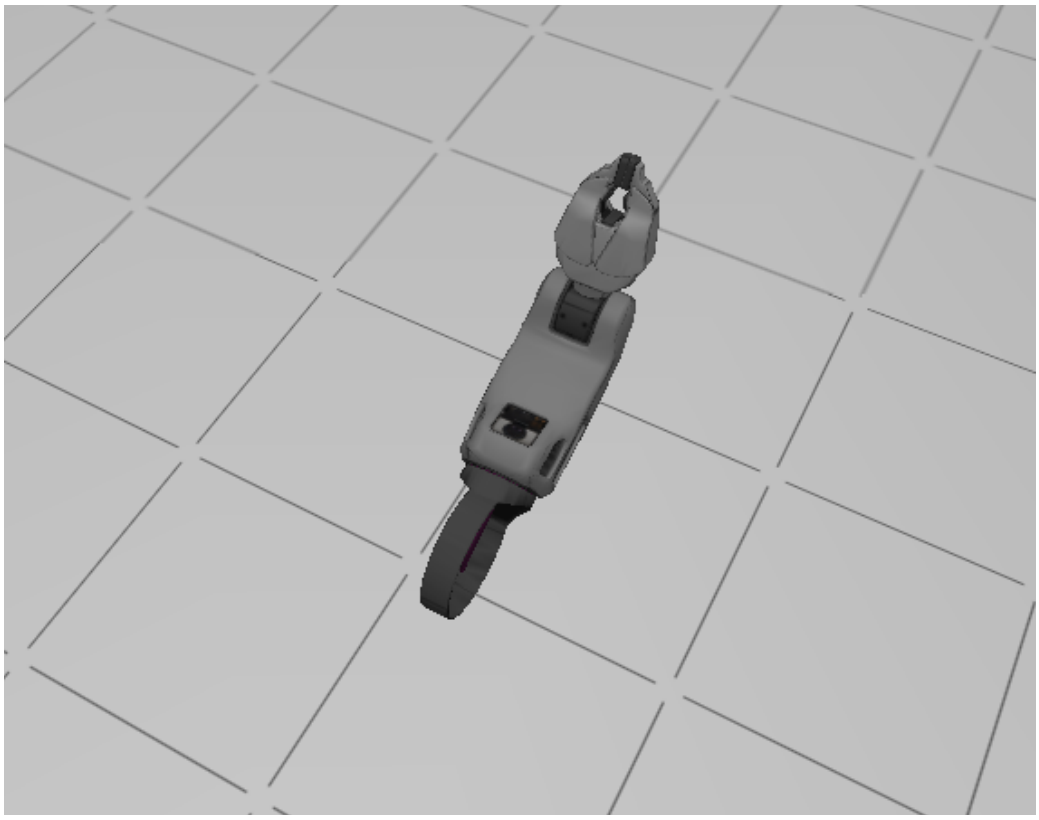

Figure 23: Just the Pr2's left arm model in Gazebo.

6 Trunk/kate_controller/my_l_gripper_controller

The arm links and joints are not what they seem in the tree. The origins and multiple joints at the same place are confusing. Replaced the Pr2's arm meshes with cylinders and circles to see interaction of links and different types of joints. There is a wrist_pan and a wrist_twist joint in the arm. Figure 23 is modeled from simplearm.urdf[7].

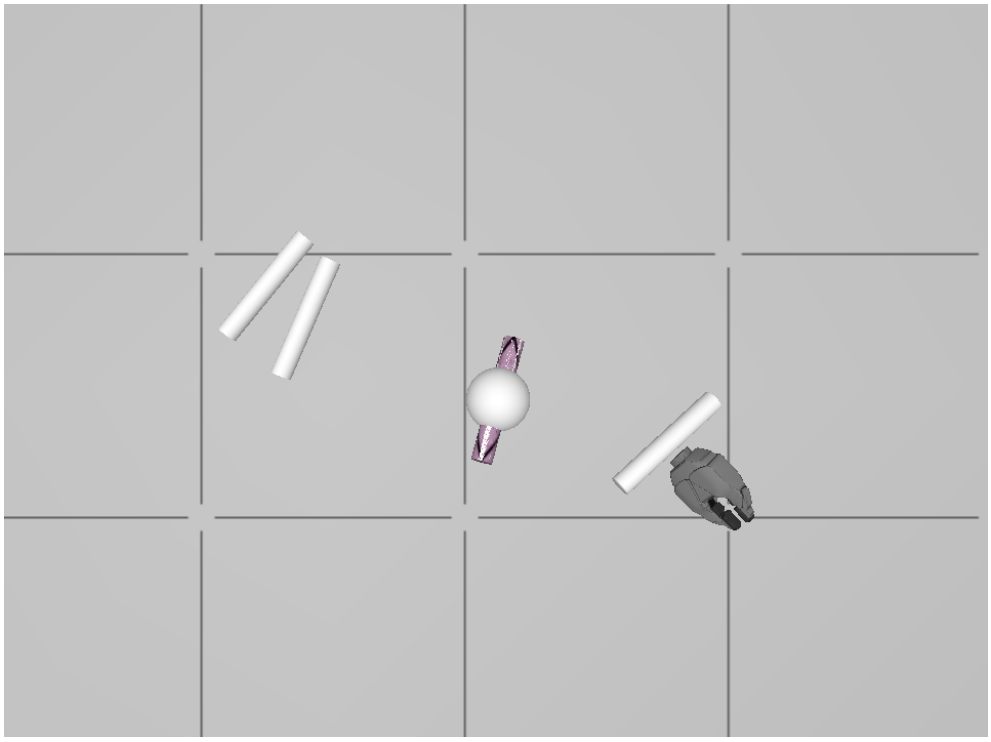

Figure 24: Pr2 left arm with meshes replaced with geometric cylinders.

The gripper can still be controller with l_gripper_controller even though it seems as the model has missing parts. The tree is still connect with all of the upper arm, elbow, and wrist joints and links, not just the gripper.

4.12 Simplifying Pr2 Gripper

Getting rid of the arm so that only the gripper is included in file. So that the controller and the model match. Deleting links in a tree are not as easy as them seem, for each link builds on each other.

7 File found in Trunk/kate_controller/

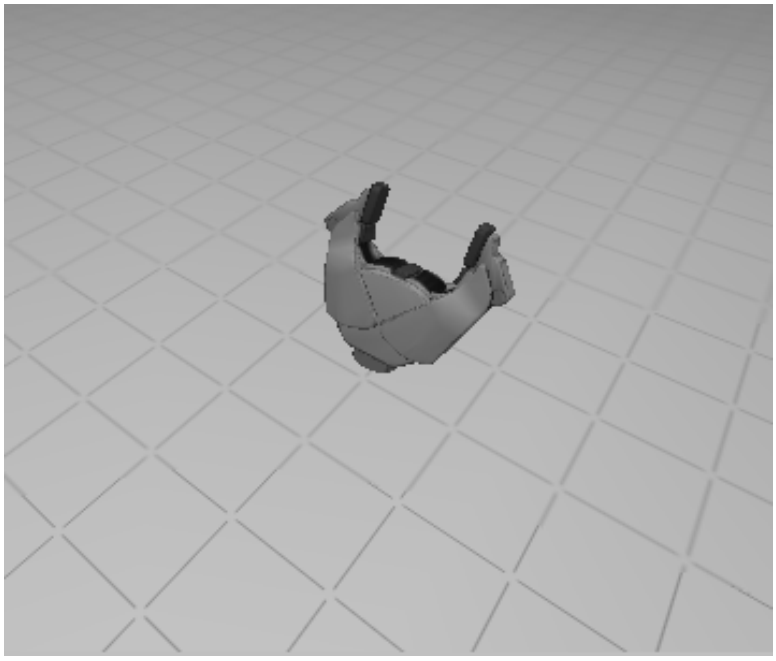

Figure 25: Just the Pr2 gripper modeled in Gazebo.

After rearrange the links and joint just to get the gripper in the model, it is not true. Though the only model that is shown is the gripper. When the URDF tree is made most of the joints and links are still there. The meshes and geometry were removed. After trying to delete joints and links in the middle of the tree the tree would brake. So it was easier to make the links not show in the model.

There is only a gripper to match the l_gripper controller. If the l_gripper can be controlled with only gripper then it may be used as controller for the Katana. Also since the l_gripper only controls one joint or object, may be able to use the code to create a controller from scratch that will control one of the Katana's joints. That could then be added on to control all of the controllers. Need a simple model of the Katana before trying to control it.

4.13 Simple Katana

To start simulating Kate in Gazebo the Katana needs to be simulated first. It does not have to look like the Katana with meshes but needs the correct properties to resemble how the Katana works in the real world. The Katana has five degrees of freedom with its joint and weighs about 5 kg in total, with each link of the Katana having different weight related to their size. A simple version of a simulated Katana in available in simplearm.urdf[8].

8 File can found in /Trunk/kate_controller

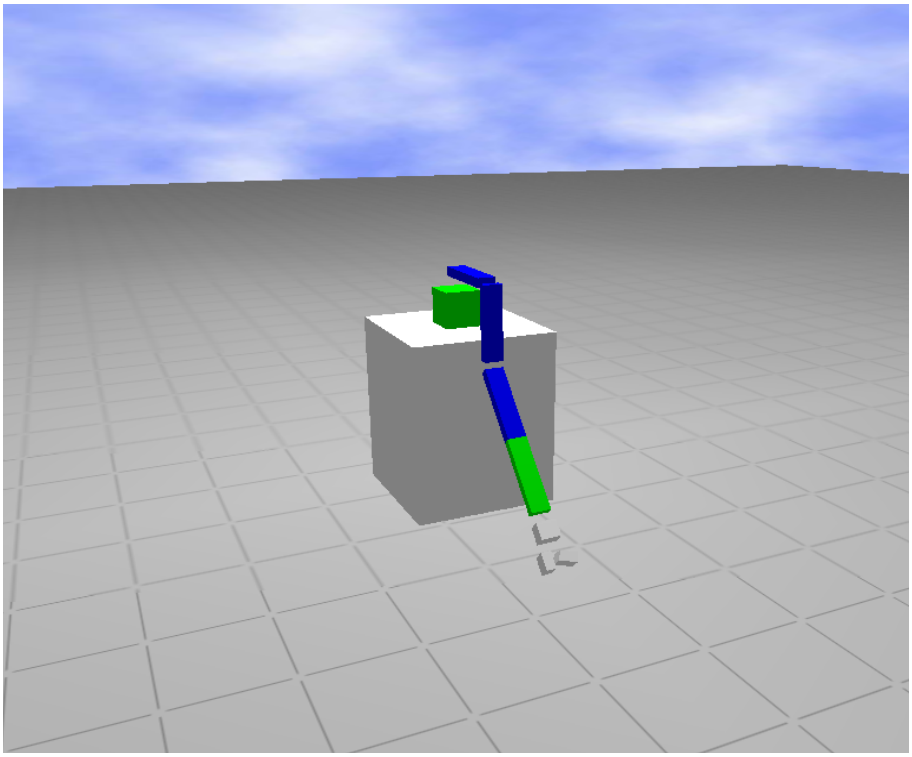

Figure 26: Gazebo model of simplearm.urdf.

The modeled Katana is made up of boxes that simplify the general movement and function of the Katana. The exact imagines are not needed for function, they are just for beautification. The meshes make the Katana look more like a finished product and a copy of the real Katana. The simple Katana is color coded depending on the joint type so that when trying to command the joints, the color will show the axis rotation. The green links twist along the z-axis. While the blue links are hinged to the previous link. The white links signify the robots base and gripper.

Now there is a simple Katana to try and control the angles of the joints. However, there is not a controller that is connected to the Katana.

4.14 Controllers

There are two main types of controllers, those that use either a joystick or keyboard for command input or those that use message commands. The joystick and keyboard commands are set commands that are mostly useful for moving object along a plane or flat surface. There are less degrees of freedom so the commands need not be complicated. A message controller though is used mostly for objects that have more than one degree of freedom. They need the extra information given with a message to understand the controller.

4.14.1 Joystick Controlling

A basic joystick controller is used to control either the Erratic and Pioneer3dx. There is already launch files included in Gazebo for using the joystick or keyboard to control either of the base robots. When tested though neither worked when launched. The robots would just stay still. The code given did not work correctly with the given joystick used.

The following commands are used to start the Pioneer3dx and the joystick controller.

Open a terminal.
2. Run
```
roscore
```

3. Open another terminal.
4. Run
```
roslaunch p2os_urdf pioneer3dx.gazebo.launch
```

5. Open another terminal.
6. Run
```
roslaunch p2os_launch teleop_joy.launch
```

Figure 27: Commands to run and control the Pioneer3dx in Gazebo.

When commanded, the Pioneer3dx should be controlled by the joystick. To do this, hold down "1" for velocity and then use the joystick to move in the wanted direction. If there is no movement check the teleop_joy.launch[9] to make sure that the buttons for the joystick controller being used match the current button mapping. Each joystick has different button mapping so that there is no default for the buttons. Originally the mapping is done for a playstation controller and does not always map correctly with the used controller's buttons [4]. The following code maps the joystick controller to a Logitech Dual Action Gamepad. Once the buttons are correctly mapped in the launch file. The joystick will move the robot.

```
<launch>
        <!-- run the teleop node to send movement velocities to the pioneer -->
        <param name="axis_vx" type="int" value="1" />
        <param name="axis_vw" type="int" value="2" />
        <param name="axis_vy" type="int" value="0" />
        <node pkg="p2os_teleop" type="p2os_teleop" name="p2os_teleop" >
                        <remap from="/des_vel" to="/cmd_vel" />
        </node>

        <!-- run a joy node to control the pioneer -->
        <node pkg="joy" type="joy_node" name="pioneer_joy_controller" />
</launch>
```

Figure 28: Example of the joysticks buttons being mapped in teleop_joy.launch[10].

Even with the Erratic and Pioneer3dx moving with the joystick, they do not stay still when they are not given a command. They slowly move in the last direction that was given. The reason is that button "1" is mapped to velocity "axis_vx". When it is pushed the robot will continues moving. So if the last command was "1" then the robot thinks it is still suppose to be moving. To stop the robot from moving when it not commanded make sure to always press "1" to stop the robot from moving.

When controlling the Erratic or the Pioneer around the world in Gazebo they are unbalanced. If there is a simple table[11] in the world and the Pioneer runs into it, the table does not move. The Pioneer does, it gets pushed back. This is not the same reaction that happens in the real world. The Pioneer would move the table. Thus the focus or weight of the Pioneer seems incorrectly proportional to the real world.

9 The file can be found under teleop_joy.launch.original in /opt/ros/cturtle/stacks/pr2_apps/pr2_teleop.
10 The file can be found in /opt/ros/cturtle/stacks/pr2_apps/pr2_teleop.
11 The default table available in Gazebo is in /opt/ros/cturtle/stacks/simulator_gazebo/gazebo_worlds/launch.

The joystick controller can be used with any robot that has two main wheels and a castor wheel at the end. Though the joystick controller will move the wheels though it is not capable of moving an arm. An arm has more degrees of free that can be commanded with a joystick and its set buttons.

4.14.2 Creating Controller

The Katana needs a messages controller, it can be created from scratch or the Joint_trajectory_controller can be used. The Joint_trajectory_controller is how the Pr2 and Care-O-Bot's arms are controlled. After comparing the Pr2's arm controller and the Care-O-Bot's though it seemed that creating a new controller from scratch would be less complicated then trying to use theirs. This was further agree with ROS' tutorials on "How to Write Real-time Controllers" [2]. There was also a tutorial on how to use the Pr2 Real time Controllers but only how to use it with the Pr2, not another other robots.

To start my_controller was supposed to only moved the amplitude and have a steady motion of moving around in circles. However, it did not work even after following the tutorial step by step. The tutorial fails to mention that you need to make sure that there are Transmissions for all of the joints(i.e. Figure 29). If there are none the system returns a "Could not compute link poses.

```
<!-- This is the first revolute joint. base to ua -->
<transmission type="pr2_mechanism_model/SimpleTransmission"
name="base_to_ua_trans">
  <actuator name="base_to_ua_motor" />
  <joint name="base_to_ua_joint" />
<!-- Not sure what this reduction should be -->
  <mechanicalReduction>36.167452007</mechanicalReduction>
</transmission>
```

Figure 29: Example of a Transmission for the joint base_to_ua_joint.

The tree or state is invalid". There needs to be transmissions for every joint that needs to be controlled or the controller does not recognized that there are joints in the URDF model. The simple Katana with transmission is in simplearm_controllers.urdf[12]. To check to see if there are transmissions for every controllable joint in the file use the following command:

```
rosrun pr2_controller_manager pr2_controller_manager list-joints
```

It gives a list of all of the actuators and the joints. There should be an actuator for every joint. If the list is missing a joint then there is no transmission found for it. Now the controller should work according to the tutorial. That is until trying to recompile my_controller.cpp and run MyControllerPlugin. The pulgin no longer exist and there is an error. Before remaking the updated my_controller.cpp, first the following pulgin code has to be taken out

```
/// Register controller to pluginlib
PLUGINLIB_DECLARE_CLASS(my_controller_pkg,MyControllerPlugin,
                        my_controller_ns::MyControllerClass,
                        pr2_controller_interface::Controller)
```

Figure 30: Code to register the controller to plugin library.

12 File can be found in /Trunk/kate_controller

and the class recompiled. Before running the controller though the plugin needs to re-add the code and recompile the project. The reason is that the first time making the controller only the code is getting compiled. The second time the controller is compiled is to declare it as MyControllerPlugin, a plugin and a controller. This is only stated in the tutorial when you first create your controller. However, without the compiling the code and then compiling the plugin the controller will not be register as a controller plugin and will no show up as a rostopic. Meaning commands can not be sent to it.

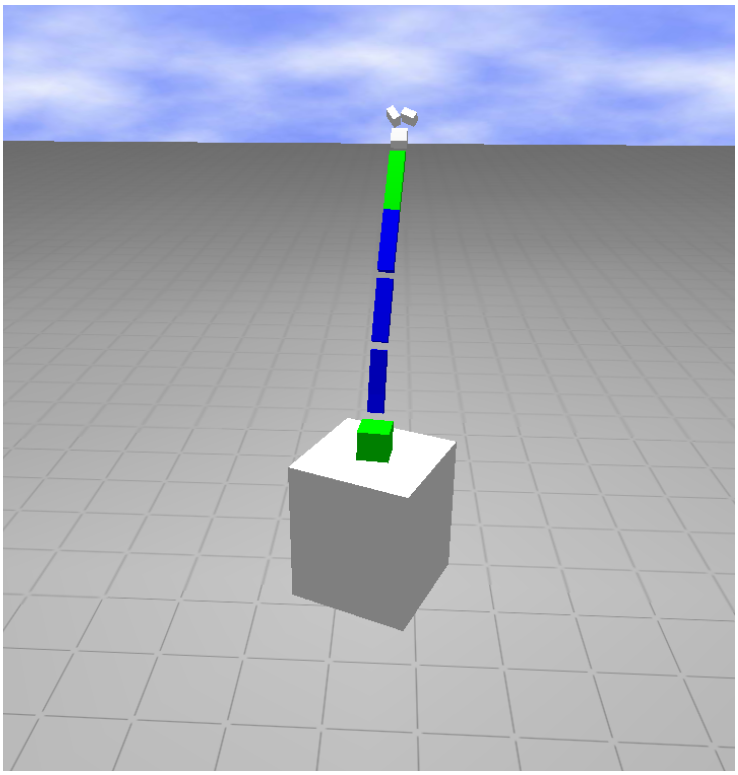

Figure 31: Gazebo model of the simple Katana being controlled with my_controller.

In creating a controller from scratch it can control any of the joint states, including velocity, amplitude, and position [8]. After understanding how controllers are created and work my_controller could change the Katana's amplitude and velocity, next was the angle. The problem was that the joint's position in joint_states is not in angles but in N/m. Converting from angles to N/m is not a simple conversion.

4.14.3 Joint_Trajectory_Controller

Since creating a simpler controller from scratch seemed to be more complicated than originally thought it was decided to use the Pr2's joint_trajectory_controller instead of creating a N/m to angle convertor. To use joint_trajectory_controller there needs to be a launch file that loads a yaml file that corresponds with the JointSplineTrajectoryController and uses the joint trajectory_action package. For examples look at kate_joint_spline_controller.launch and kate_joint_spline_controller.yaml[13] They are named the same to know what yaml file goes with what launch file.

The yaml file includes the type of controller, JointSplineTrajecotryController, list of joints

13 Files can be found in Trunk/kate_controller/Joint_spline_trajectory.

for controlling, pid controller[14] information, and constraint. The hardest to figure out is the pid controller information, seeing as it is a try and error to get correct pids. If the pids are incorrect when the joint is commanded to move the whole Katana moves with it. As the pid gets closer to being to correct, the Katana will not move as a whole but the joint will still over shoot the angle. Once the pid are correct, when the joint is commanded a new position[15] it will move to that angle and stay without creating an reaction of the rest of the Katana.

When my kate_joint_spline_controller is launched the joint_trajectory_controller is loaded and running. Thus commands can be used to control the joints. There needs to be another terminal open to send the commands for the joints. There are two ways of sending commands, there can be a header with a time stamp that will wait until that time to run the command or the command can

```
rostopic pub /arm_controller/command trajectory_msgs/JointTrajectory '{header:
{stamp: {secs: 120,nsecs: 0}} ,joint_names: ['base_twist_joint',
'base_to_ua_joint', 'ua_to_fa_joint', 'fa_to_wr_joint', 'wr_to_gr_joint'],
points: [{positions:  [-0.88939999999999997, -1.5624, 0.85919999999999996,
1.7863, 1.3653], velocities: [], accelerations: [], time_from_start: {secs: 3,
nsecs: 0}}]}'
```

Figure 32: Command for moving the Katana's joints to the new positions.

be ran as soon as it is sent. The trouble with giving a time stamp to the command is if the time has already passed the command will not be executed. Deleting the header with the time stamp will remove the waiting and execute the command instantly. Make sure that the “time_from_start” is a long enough time to move the joints at a reasonable speed. If the time is too short the the motion and force will move the whole Katana or robot. The above code is an example of a command[16].

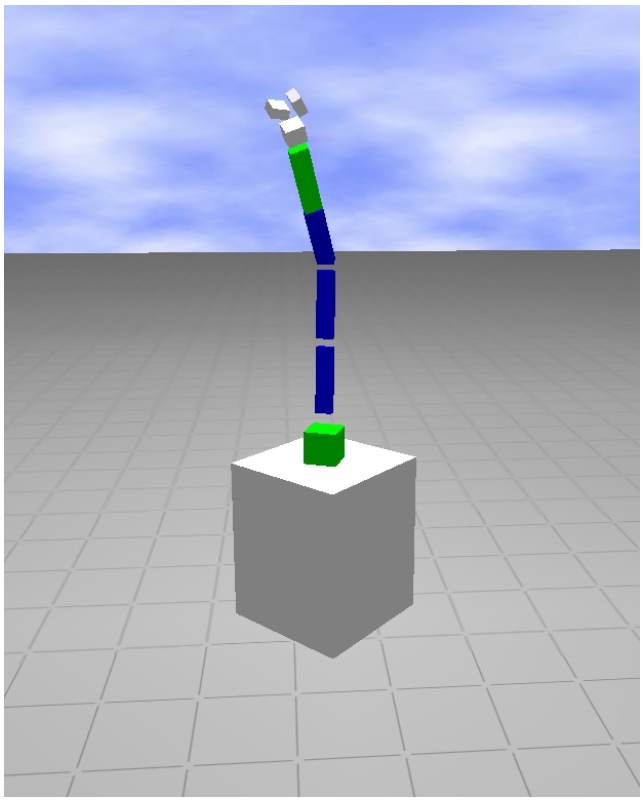

Figure 33: Gazebo model of the simple Katana being controlled with joint_trajectory.

14 proportional–integral–derivative controller
15 Joint positions are in radians.
16 Uses joints that are defined in simplearm_controller.urdf.

Using the joint_trajectory_controller the Katana's joints' positions, velocities, and accelerations can be commanded from the terminal. The above figure shows the Katana being controlled. Also a sequence of movements can be commanded, so that the user can tell the simulation the joints configuration for a set command. Do to time constrictions kate_joint_trajectory_controller only works with the first joint. The other joints can not be commanded this way because of their pid controller information in the .yaml file and their mechanical reduction in the transmissions is incorrect.

5 Future Improvements

The following will be future improvements and development. Currently the simulated Kate is in two different section, a Pioneer3dx base and a simple Katana. The Pionees3dx can only be commanded to move. However, its weight or friction are incorrect. It bounces off chairs and tables when it runs into them. Normally the Pioneer would slowly move the objects. This needs to be looked into and fixed so that is represents the real world. The simulated Katana is not fully controllable with a joint trajectory controller and is made up of geometric shapes. It seems that the pid controller information or the mechanical reduction are not correct with the actual Katana. The Katana links also need to be changed from shapes to meshes so that it looks like the actual Katana in detailed and not just function. Meshes were never created or found and since it is just for visual they were considered unimportant at the time. At some time after the above have been implement sensor and cameras need to be added the simulated Kate. The over all goal of a completely simulated Kate in Gazebo has many more steps before being finished.

6 Lessons Learned

6.1 Learning Curve

The Learning Curve for Gazebo took took longer than expected. It was not just learning Gazebo but how Gazebo and ROS interact to simulate and control objects.

6.2 Get all Command Information

The Katana's original controller did not accept angles as commands for position but N/m as an effort. This is not how the actual Katana and software control the joints. This was found out after the fact so the controller had to be redesigned. The Katana's controller went from creating a controller for each joint to using joint trajectory controller.

6.3 Existing vs New Controllers

Make sure to really look at creating own controllers versus already implemented controllers. When falsely asserted it can lead to lots of work that will end up not being use. At first look the joint trajectory controller was far more complex than creating a controller. Thus was further agreed with tutorials teaching how to make and build controllers for Gazebo and ROS. Though it looked

more complex is was what is needed to control the Katana. The first synopses was incorrect. Originally the controllers for the Pr2 were only for the Pr2 but eventually were updated from all robots, this was unclear at first look. Though creating a new controller for the Katana was not used it helped in understanding the joint trajectory controller. The second look at the joint trajectory controller after creating a controller seemed simpler.

Creating a controller showed how they are made as rostopics and then used in Gazebo. Previous implemented controllers are not as scary as originally when realizing that they can be reused. In the end the my_controller may not have been used with controlling the Katana's joints in angles but it was useful with the experience of learning how controller are made.

7 Conclusions

Initially the project was to simulate a completely working Kate in Gazebo. However, as the project evolved it became more of researching how to use Gazebo to simulated Kate and mainly the Katana, while being able to control it though a terminal or interface. Though the original goal was not reached there were steps taken to reach it.

To begin with, the learning curve for Gazebo took much more time than expect. However, no one really understand how Gazebo worked, just that is was a 3D simulator that other people had used to simulate robots. Now with the new understanding though others can use that information and will not need such a huge learning curve.

It was a good learning experience, that was a start to a final goal of simulating Kate. The end product is a Pioneer3dx that can be controlled either through the keyboard or a joystick and a graphically simplified Katana that can be controlled through terminal commands.

Work Cited

1. Conley, Ken. "Cturtle/Installation/Ubuntu - ROS Wiki." *Documentation - ROS Wiki*. 14 Jan. 2011. Web. 02 May 2011. <http://www.ros.org/wiki/cturtle/Installation/Ubuntu>.
2. Conley, Ken. "Pr2_mechanism/Tutorials - ROS Wiki." *Documentation - ROS Wiki*. 26 July 2010. Web. 29 Apr. 2011. <http://www.ros.org/wiki/pr2_mechanism/Tutorials>.
3. Conley, Ken. "Cturtle - ROS Wiki." *Documentation - ROS Wiki*. 14 Feb. 2011. Web. 02 May 2011. <http://www.ros.org/wiki/cturtle>
4. Gassend, Blaise. "Ps3joy - ROS Wiki." *Documentation - ROS Wiki*. 11 Nov. 2010. Web. 01 May 2011. <http://www.ros.org/wiki/ps3joy>.
5. Glaser, Start. "Pr2_controllers_msgs - ROS Wiki." *Documentation - ROS Wiki*. 8 Jan. 2010. Web. 27 Apr. 2011. <http://www.ros.org/wiki/pr2_controllers_msgs>.
6. Gossow, David. "Pr2_simulator/Tutorials/BasicPR2Controls - ROS Wiki." *Documentation - ROS Wiki*. 7 Jan. 2011. Web. 01 May 2011. <http://www.ros.org/wiki/pr2_simulator/Tutorials/BasicPR2Controls>.
7. "Player." Willow Garage. Web. 01 May 2011. <http://www.willowgarage.com/pages/software/player>.
8. "Pr2_mechanism_model: Pr2_mechanism_model::JointState Class Reference." Documentation - ROS Wiki. 02 May 2011. Web. 02 May 2011. <http://www.ros.org/doc/api/pr2_mechanism_model/html/classpr2__mechanism__model_1_1JointState.html>.
9. "Pr2_simulator - ROS Wiki." Documentation - ROS Wiki. 3 Aug. 2010. Web. 02 May 2011. <http://www.ros.org/wiki/pr2_simulator>.
10. "Robot_mechanism_controllers: Controller::JointSplineTrajectoryController Class Reference." Documentation - ROS Wiki. 02 May 2011. Web. 02 May 2011. <http://www.ros.org/doc/api/robot_mechanism_controllers/html/classcontroller_1_1JointSplineTrajectoryController.html>.
11. "ROS." Willow Garage. Willow Garage. Web. 01 May 2011. <http://www.willowgarage.com/pages/software/ros-platform>.
12. Rousseau, Jeff. "Simulator_gazebo/SystemRequirements - ROS Wiki." Documentation - ROS Wiki. 02 Mar. 2011. Web. 01 May 2011. <http://www.ros.org/wiki/simulator_gazebo/SystemRequirements>.
13. "Simulator_gazebo/Tutorials/Gazebo_ROS_API - ROS Wiki." *Documentation - ROS Wiki*. 4 Aug. 2010. Web. 01 May 2011. <http://www.ros.org/wiki/simulator_gazebo/Tutorials/Gazebo_ROS_API>.
14. "Simulator_gazebo/Tutorials/StartingGazebo - ROS Wiki." *Documentation - ROS Wiki*. 27 Sept. 2010. Web. 01 May 2011. <http://www.ros.org/wiki/simulator_gazebo/Tutorials/StartingGazebo>.
15. "Urdf/XML/Joint - ROS Wiki." *Documentation - ROS Wiki*. 04 Jan. 2011. Web. 02 May 2011. <http://www.ros.org/wiki/urdf/XML/Joint>.
16. "Urdf/XML/Link - ROS Wiki." *Documentation - ROS Wiki*. 04 Apr. 2011. Web. 01 May 2011. <http://www.ros.org/wiki/urdf/XML/Link>.
17. Watts, Kevin. "Xacro - ROS Wiki." *Documentation - ROS Wiki*. 10 Feb. 2011. Web. 03 May 2011. <http://www.ros.org/wiki/xacro>.
18. "Welcome at the Collaborative Center for Applied Research on Service Robotics." *ZAFH - Autonome Mobile Serviceroboter*. Web. 01 May 2011. <http://www.zafh-servicerobotik.de/en/index.php>.

Appendix

A Directory Trees

There are two directories that are not included with their trees. They are opt and Examples. The opt directory has the ROS cturtle and Gazebo installation. While Examples has Care-O-Bot and Pr2 examples. All software can be found on the cd[17].

A.1 svn/Document

```
.
|-- CPE400
|-- Pictures
|-- Presentation
`-- Research

4 directories
```

A.2 svn/Trunk

```
.
|-- cob_mod
|   |-- cob_apps
|   |   `-- cob_bringup
|   |       `-- ros
|   |           `-- launch
|   `-- cob_common
|       `-- cob_description
|           |-- bin
|           |-- build
|           `-- ros
|               |-- calibration
|               |-- gazebo
|               |-- meshes
|               |   |-- arm_v0
|               |   |   |-- convex
|               |   |   `-- iv
|               |   |-- base_v0
|               |   |   |-- convex
|               |   |   `-- iv
|               |   |-- lbr_v0
|               |   |   |-- convex
|               |   |   `-- iv
|               |   |-- sdh_v0
|               |   |   |-- convex
|               |   |   `-- iv
|               |   `-- torso_v0
|               |       |-- convex
```

17 The opt and Example directory trees are include in /Tunk/Documents.

```
|               |       `-- iv
|               |-- robots
|               `-- urdf
|                   |-- arm_v0
|                   |-- base_v0
|                   |-- base_v0_ipa
|                   |-- base_v1
|                   |-- head_v0
|                   |-- lbr_v0
|                   |-- sdh_v0
|                   |-- sensors
|                   `-- torso_v0
|-- kate_controller
|   |-- joint_spline_trajectory
|   |-- kate_arm
|   |-- my_controller_pkg
|   |   |-- bin
|   |   |-- build
|   |   |   `-- CMakeFiles
|   |   |       |-- clean-test-results.dir
|   |   |       |-- CMakeTmp
|   |   |       |   `-- CMakeFiles
|   |   |       |       `-- cmTryCompileExec.dir
|   |   |       |-- CompilerIdC
|   |   |       |-- CompilerIdCXX
|   |   |       |-- my_controller_lib.dir
|   |   |       |   `-- src
|   |   |       |-- ROSBUILD_genmsg_cpp.dir
|   |   |       |-- ROSBUILD_genmsg_lisp.dir
|   |   |       |-- ROSBUILD_genmsg_py.dir
|   |   |       |-- ROSBUILD_gensrv_cpp.dir
|   |   |       |-- ROSBUILD_gensrv_lisp.dir
|   |   |       |-- ROSBUILD_gensrv_py.dir
|   |   |       |-- rosbuild_precompile.dir
|   |   |       |-- rosbuild_premsgsrvgen.dir
|   |   |       |-- rospack_genmsg_all.dir
|   |   |       |-- rospack_genmsg.dir
|   |   |       |-- rospack_genmsg_libexe.dir
|   |   |       |-- rospack_gensrv_all.dir
|   |   |       |-- rospack_gensrv.dir
|   |   |       |-- test.dir
|   |   |       |-- test-future.dir
|   |   |       |-- test-results.dir
|   |   |       |-- test-results-run.dir
|   |   |       `-- tests.dir
|   |   |-- include
|   |   |   `-- my_controller_pkg
|   |   |-- launch
|   |   |-- lib
|   |   |-- msg
|   |   |   `-- lisp
|   |   |       `-- my_controller_pkg
|   |   |-- msg_gen
|   |   |   `-- cpp
|   |   |       `-- include
|   |   |           `-- my_controller_pkg
|   |   |-- src
|   |   |   `-- my_controller_pkg
|   |   |       |-- msg
```

```
|   |   |       `-- srv
|   |   |-- srv
|   |   |   `-- lisp
|   |   |       `-- my_controller_pkg
|   |   |-- srv_gen
|   |   |   `-- cpp
|   |   |       `-- include
|   |   |           `-- my_controller_pkg
|   |   `-- yaml
|   |-- my_l_gripper_controller
|   `-- pr2_arm
|-- kate_description
|   |-- build
|   |-- calibration
|   |-- launch
|   |   `-- mine
|   |-- meshes
|   `-- urdf
|       |-- arm
|       |   |-- cob_urdf
|       |   `-- mine
|       |-- pioneer
|       `-- sensors
`-- kate_teleop

110 directories
```

C Glossary

Katana	The brand of Kate's robotic arm.
Kate	The name of the ZAFH Servicerobotik service robot.
ROS	Robot Operating System
URDF	Unified Robot Description Format
XML	Extensible Markup Language