Mobile Manipulation for the KUKA youBot Platform

A Major Qualifying Project Report

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Timothy Jenkel

Richard Kelly

Paul Shepanski

March 11, 2013

Approved By:

Professor Sonia Chernova, Advisor

Professor Charles Rich, Advisor

Abstract

This paper details the implementation of object manipulation and navigation capabilities for the KUKA youBot platform. Our system builds upon existing approaches taken from other robot platforms and the open source Robot Operating System, and extends these capabilities to the youBot, resulting in a system that can detect objects in its environment, navigate to them autonomously, and both pick up and place the objects with a simple user interface. This project is part of the larger Robot Autonomy and Interactive Learning Lab project to provide web-based control of the youBot to public users.

Table of Contents

Table of Figures

1 Introduction

Our project was part of a larger ongoing research project headed by Professor Sonia Chernova. The goal of that project is to provide a remote interface to a KUKA youBot. The interface should allow easy to use tools for both autonomous and not autonomous navigation and simple object manipulation within the robots workspace. The workspace is a small room with various pieces of furniture and small objects for object manipulation pictured below. The overarching project also intends to make a simulated version of both the robot and the workspace available for experimentation. The purpose of this functionality is to allow robotic tasks to be crowd sourced.

Figure : Picture of the workspace

Our project satisfied the larger projects need for object manipulation and navigation by developing software for object manipulation based on object detection with a Kinect. It also combined those two capabilities with a simplified interface for the already existing navigation code.

2 Background

To achieve our objectives, we built upon a collection of existing knowledge and implementation effort provided by the robotics community. In the sections to follow, we will briefly introduce the concepts and tools that the reader should be familiar with. These have been organized by two overarching themes:

- Tools: existing hardware and software that we investigated for this project
- Algorithms: the concepts employed by our software and how it produces useful results

2.1 Tools

In this section, we will briefly introduce each of the tools that we researched throughout the project:

- The KUKA youBot, a 4-wheel mecanum drive robot with a small industrial manipulator
- Robot Operating System (ROS), a robot software development framework and library collection
- The Gazebo simulation environment
- Tools related to vision processing and object detection
- Tools related to arm operation and object manipulation

Each of these will be discussed in the sections below.

2.1.1 KUKA youBot

We are using the KUKA youBot mobile manipulator, as shown in Figure . The KUKA youBot base uses an omnidirectional drive system with mecanum wheels. Unlike standard wheels, mecanum wheels consist of a series of rollers mounted at a 45° angle. This allows the robot to move in any direction, including sideways, which makes the robot much more maneuverable in tight areas.

Figure : The KUKA youBot

The KUKA youBot is controlled by an onboard computer running a version of Ubuntu Linux. The youBot's onboard computer has many of the features of a standard computer, including a VGA port to connect an external monitor, several USB ports for connecting sensors and other peripherals, and an Ethernet port for connecting the youBot to a network.

Connected to the base of the youBot is a 5 degree-of-freedom arm. The arm is a 5 link serial kinematic chain with all revolute joints. The arm is sturdy and non-compliant, similar to KUKA's larger industrial robot arms. The dimensions of each link of the arm, as well as the range of each joint are shown in Figure below. The rotation of each joint in the youBot's arm is measured by a relative encoder; therefore, users must manually move the arm to a home position before the arm is initialized. The youBot's wrist is equipped with a two finger parallel gripper with a 2.3 cm stroke. There are multiple mounting points for the gripper fingers that users can choose based on the size of the objects to pick up.

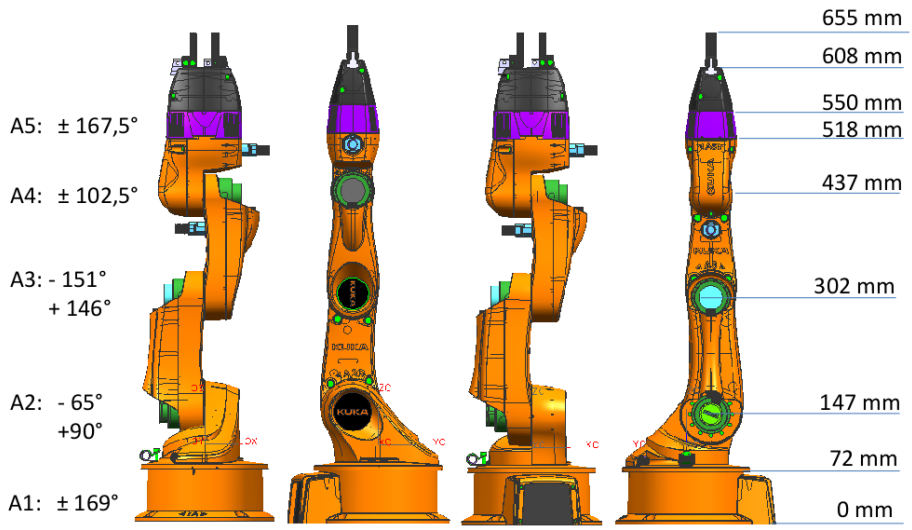

Figure : Dimensions of the KUKA youBot arm [1]

2.1.2 Robot Operating System (ROS)

ROS (Robot Operating System) is an open source software framework for robotic development. The primary goal of ROS is to provide a common platform to make the construction of capable robotic applications quicker and easier. Some of the features it provides include hardware abstraction, device drivers, message-passing, and package management [2]. ROS was originally developed starting in 2007 under the name Switchyard by the Stanford Artificial Intelligence Laboratory, but since 2008 it has been primarily developed by Willow Garage and is currently in its sixth release.

The fundamental purpose of ROS is to provide an extensible interprocess communication framework which simplifies the design of distributed systems. The building blocks of a ROS application are *nodes*; a node is a named entity that can communicate with other ROS nodes on behalf of an operating system process. At this time, ROS provides support for nodes written in C++ and Python, and experimental libraries exist for a handful of other languages.

There are three ways that nodes may communicate in a ROS environment:

1. By publishing *messages* to a *topic*.
2. By listening to messages published on a topic.
3. By calling a *service* provided by another node.

Messages represent data structures that may be transferred between nodes. Messages may contain any named fields; each field may be a primitive data type (e.g. integers, booleans, floating-point numbers, or strings), a message type, or an array of either type. ROS provides a mechanism for generating source code from text definitions of messages.

ROS *topics* represent named channels over which a particular type of message may be transferred. A topic provides one-directional communication between any number publishing and consuming nodes. Figure provides a visualization of a ROS graph; ovals represent nodes, and the directed arrows represent publish/subscribe relationships between nodes via a particular topic.

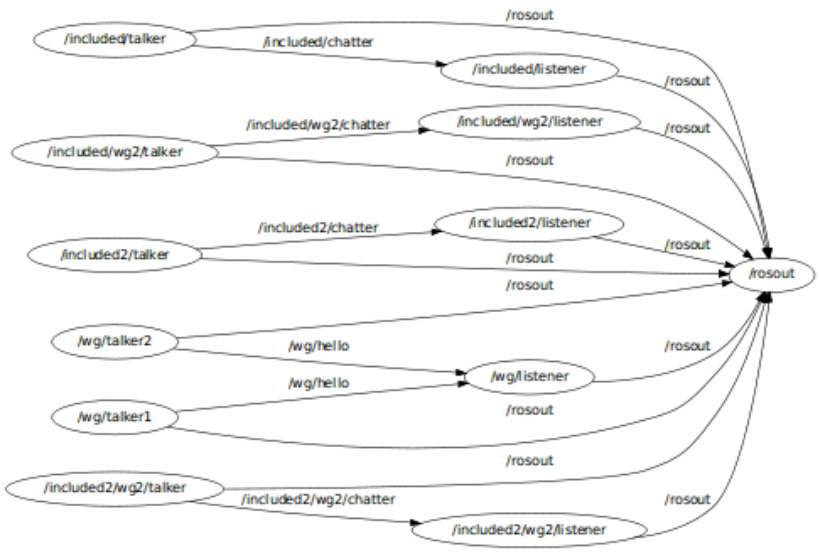

Figure : Graph of ROS application

A node that publishes to a topic is called a *publisher*. Publishers often run continuously in order to provide sensor readings or other periodic information to other nodes; however, it is possible to write a publisher that publishes a message only once.

A node which listens to messages on a topic is called a *subscriber*. A subscriber specifies which topic it wants to listen to as well as the expected message type for the topic, and registers a callback function to be executed whenever a message is received. Similar to publishing a single message, a subscriber may instead block until a message is received if only a single message is required.

Finally, a node may provide a *service* to other nodes. A service call in ROS resembles a remote procedure call workflow: a *client* node sends a request to the service provider node, a registered callback function in the service provider node performs the appropriate action(s), and then the service provider sends a response back to the client.

Since a service call is an exchange between one client node and one service provider node, they do not employ topics. Service request and response messages, however, are defined in a similar manner as standard messages, and they may include standard messages as fields.

Any node can perform any combination of these actions; for example, a node could subscribe to a topic, call a service on a message it receives, and then publish the result to another topic. This allows a large amount of flexibility in ROS applications. ROS also allows applications to be distributed across multiple machines, with the only restriction that nodes which require hardware resources must run on a machine where those resources are available. More information about ROS including tutorials, instillation instructions, and package information can be found on their website: www.ros.org/wiki.

2.1.3 Gazebo Simulator

The Gazebo simulator [3] is a multi-robot simulator, primarily designed for outdoor environments. The system is compatible with ROS, making it a good choice for representing the robot's environment. Gazebo also features rigid body physics simulation, allowing for collision detection and object manipulation. Finally, Gazebo allows for the simulation of robot sensors, allowing us to incorporate the Kinect's full functionality and the overhead cameras into the simulation.

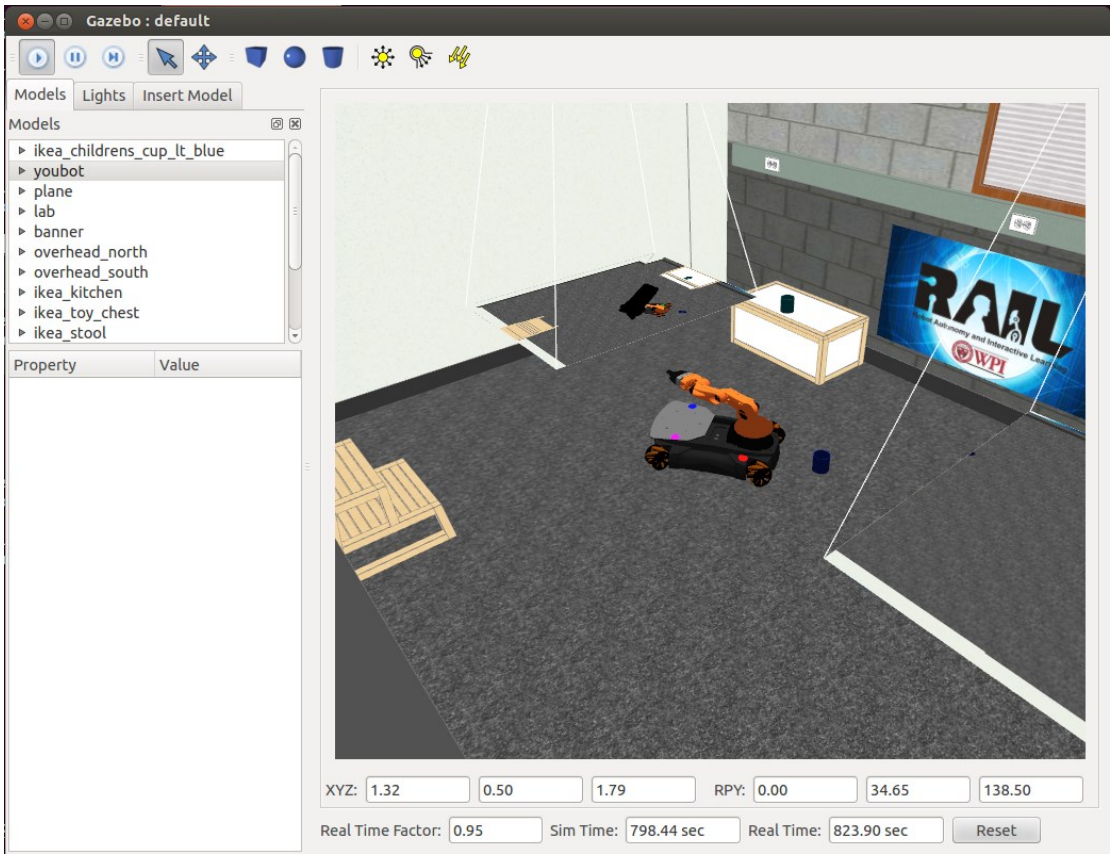

Figure : Gazebo Simulation of youBot Environment

The room as it is envisioned in the final product is modeled in the Gazebo simulation. An example image of the simulation is included below. The flat images projected in the air represent the overhead cameras' point of view, which is used in the web interface. Both the robot and arm are fully represented and capable of being manipulated within the simulation. All of the objects that the robot is expected to interact with in the room are also present and can be added and moved. Some of the more obvious objects present:

- IKEA table
- IKEA chest
- Plastic cups
- Stool

As we determine what objects should be present in the environment based upon the design decisions that we make and the experience that we gain working with the robot, the gazebo simulation and the objects included have to be updated to reflect the changes.

2.1.4 Vision

We investigated a handful of tools to help us discover objects in our robot's environment and produce suitable representations of them in software. These include:

- The Microsoft Kinect, a structured-light 3D camera designed for the Xbox game console.
- The Object Recognition Kitchen (ORK), an effort by Willow Garage to develop a general-purpose object detection and recognition library for ROS.
- The Tabletop Object Detector, an object detection and recognition library for objects on tables
- The Pont Cloud Library (PCL), a general purpose library for working with point cloud data structures

Each of these will be introduced in the following sections.

2.1.4.1 Microsoft Kinect

The Microsoft Kinect is a consumer device originally designed by Microsoft as a peripheral for the Xbox game system. It was designed to compete against the motion-sensitive controller introduced by Nintendo for the Wii game system.

Figure : The Microsoft Kinect [4]

The Microsoft Kinect is composite device which includes the following components:

- A color digital camera
- A structured-light infrared projector
- An infrared digital camera
- A microphone array
- A tilt motor, which can adjust the pitch of the attached cameras and projector

The Kinect derives depth information from an environment by projecting a grid of infrared points in a predictable pattern. The resulting projection on the environment is viewed by the integrated infrared camera and interpreted to produce depth information for each point. An example of this projection is illustrated in Figure .

Figure : Microsoft Kinect IR Projection [5]

This structured light approach poses challenges for objects particularly near and objects particularly far from the sensor. For objects closer than 0.8 meters, the projected points appear too closely together for the sensor to measure; this results in a short-range blind spot that can have implications for where the sensor is mounted, particularly for small robots. For objects farther than 4 meters, the projected points fade into the background. This maximum range is also adversely affected by the presence of infrared noise in the environment. As such, the Kinect's range is significantly degraded outdoors.

To support game development, Microsoft has developed a software library for interpreting a human figure in this point cloud. This library allows users of the Microsoft tool chain for C++ and C# to easily discover human figures in an environment and measure determine the 3D position of the figure's extremities.

The impressive quality of the depth information produced by the Kinect and the Kinect's low price make it a very attractive sensor for robotics research. As a result, a variety of open-source drivers have been developed for the Kinect which allow one to process the information produced by a Kinect as a cloud of points located in 3D space. In addition, some of these libraries also provide depth registration, wherein each depth point is annotated with the RGB color of that point in the environment.

We have investigated the openni_camera [6] package for ROS. This package produces ROS point cloud message data structures which makes it easy to use a Kinect with many existing ROS packages and infrastructure. In addition, openni_camera also supports depth registration.

2.1.4.2 Object Recognition Kitchen

The Object Recognition Kitchen (ORK) is a tool chain for object recognition that is being developed by Willow Garage [7]. It is independent of ROS, although it provides an interface for working with ROS. The ORK is designed such that object recognition algorithms can be modularized; it implements a few different object recognition approaches, and provides an interface, called a pipeline, that new algorithms can conform to. Each pipeline has a source, where it gets data from, the actual image processing, and a sink, where the data is output. When performing object detection, the ORK can run multiple pipelines in parallel to improve results. The built-in pipelines are LINE-MOD, tabletop, TOD, and transparent objects. Tabletop is a ported version of the ROS tabletop object detector package. TOD stands for textured object detection and matches surfaces against a database of known textures. The transparent objects pipeline is similar to the tabletop object detector, but works on transparent objects such as plastic cups or glass.

We were not able to successfully test the ORK in our environment; it appears that the project is under active development, but not yet complete.

2.1.4.3 Tabletop Object Detector

The tabletop object detector is a software library originally written by Willow Garage for its flagship research robot, the PR2 [8]. The purpose of this package is to provide a means of recognizing simple household objects placed on a table such that they can be manipulated effectively.

Given a point cloud from a sensor, the tabletop object detector first discovers the surface of the table it is pointed at through a process called segmentation. Once the table surface has been discovered, the algorithm filters the original point cloud to remove all of the points that do not lie

directly above the surface of the table. Finally, the remaining points are clustered into discrete objects using nearest-neighbor clustering with a Kd-tree.

This process produces a sequence of point clouds, one for each object. As an additional optional step, the tabletop object detector also provides rudimentary object recognition. Given a database of household object meshes to compare against, the tabletop object detector provides an implementation of iterative closest point (ICP), a relatively simple algorithm for registering a sensor point cloud against a model point cloud. If a point cloud successfully compares against a model in the database, the tabletop object detector also provides the more detailed model mesh as a part of the detection result.

Although this package works well for the PR2 in a constrained environment, it has some noteworthy limitations:

- It cannot detect objects on the floor, because it expects the presence of a table plane.
- It is very sensitive to changes in perspective of an observed table.

During our testing, we found that our Microsoft Kinect sensor must be positioned at just the right height and angle with respect to our IKEA children's table in order for the tabletop object detector to properly detect the table. However, even when our table was successfully detected, the library was unable to detect the IKEA children's cups that we placed upon it.

It is worth noting that the tabletop object detector was originally designed to work with point clouds produced by computing the disparity between two cameras, and we chose to use a Microsoft Kinect instead; in addition, our IKEA children's table and children's cups are notably smaller than the standard rectangular table and household objects that the library was designed to work with. These factors likely influenced our results.

2.1.4.4 Point Cloud Library (PCL)

The Point Cloud Library (PCL) [9] is an open source project for image and point cloud processing. PCL was originally developed by Willow Garage as a package for ROS; however, its utility as a standalone library quickly became apparent. It is now a separate project maintained by the Open Perception Foundation, and is funded by many large organizations. The PCL is split into multiple libraries which can be compiled and used separately. These include libraries include support for: filtering, feature finding, key point finding, registration, kd-tree representation, octree representation, image segmentation, sample consensus, ranged images, file system I/O and visualization. This project relies on reading and writing point clouds to files, point cloud visualization, downsampling point clouds using a filter, plane segmentation, and object segmentation using Euclidean Cluster Extraction. Euclidean Cluster Extraction works by separating the points into groups where each member of a group is within a specified distance of at least one other member of the group. Figure shows the result of plane segmentation removal and then Euclidean Cluster Extraction on a sample table scene. Note how the top of the table and floor have been removed and that different colored clouds represent separate clusters.

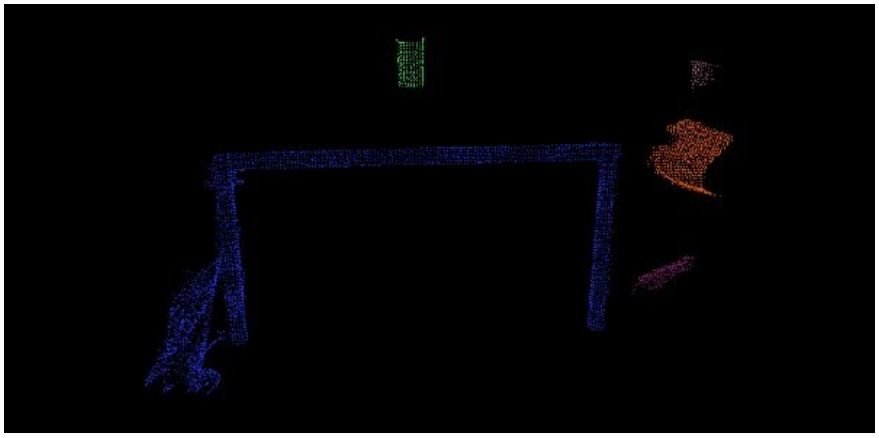

Figure : Result of PCL cluster extraction [10]

PCL also provides tutorials and examples for most of its features allowing easy implementation and modification of the PCL algorithms. ROS package is also provided to allow convenient use of the PCL in a ROS node. This package provides functions for converting between ROS and PCL point cloud types and many other features.

2.1.5 Arm

To provide control over the youBot's integrated 5-DOF arm, we focused on two ROS stacks:

- The Arm Navigation stack [11]
- The Object Manipulation stack [12]

Originally developed for the PR2, these stacks comprise most of what is called the object manipulation pipeline. This pipeline provides a robot-independent set of interfaces, message types, and tools to help one implement common object manipulation actions for a particular robot.

Each of the aforementioned stacks will be introduced in the sections to follow.

2.1.5.1 Arm Navigation Stack

The arm navigation stack was developed by Willow Garage to provide for the collision-free motion of a multiple degree of freedom robot arm. While only implemented originally for the PR2 robot arm, the stack was designed in such a way that it could be used for any arm, with the proper setup. Once that setup is complete, the stack handles collision avoidance, inverse kinematics, and publishes status updates on the arm's progress. In order to move the arm to a given position and orientation, only a relatively simple message is required to set the arm's goal. Once that is received, the stack plans a collision-avoiding route to the target location, and produces a set of joint positions to create a smooth path to the destination.

Oddly enough, the arm navigation stack did not provide anything to actually run through that path. Despite this, we chose to use this stack for the built-in features, the relative ease to set up,

and the support for further arm tasks. As previously mentioned, the motion path that is created takes collisions, both with the arm itself and objects discovered in the environment, into account. That would be very difficult to program in a timely manner, sparing us significant development time. There were only two major sections of the program that had to be created for the arm navigation stack to operate properly: the above mentioned program to take the path and execute it, and a description of the arm to be used. The first was easy to create, and the second was generated based on the robot model, which made it simple to implement once the requirements were understood. Finally, the arm navigation stack is used by the PR2 to feed directly into picking up an object with one of its arms, so if we wished to also leverage that code, it would be of a great benefit to follow the same process. In fact, arm navigation actually contains both of the kinematics models used by the object manipulation stack below.

We looked into a few other possible kinematics models and arm controllers, but none of them provided the level of functionality or support that the arm navigation stack did. The official KUKA youBot arm manipulation software was designed for a previous version of ROS, and had not been updated, in addition to not containing collision avoidance capabilities. Compared to all other options, the arm navigation stack provided the most useful features and the easiest implementation.

2.1.5.2 Object Manipulation Stack

The object manipulation stack provides the framework for picking up and placing objects using ROS. The stack is designed to be fairly robot independent, but requires some robot specific components to work properly. These robot specific components include a grasp planner, a gripper posture controller, an arm/hand description configuration file, and a fully implemented arm navigation pipeline. The object manipulation pipeline is fully implemented for the PR2 and some other robots, but not for the youBot.

The object manipulation pipeline was designed to work either with or without object recognition. For unknown objects, the grasp planner must select grasp points based only on the point cluster perceived by the robot's sensors. If object recognition is used, grasps for each item in the object database are pre-computed, and should be more reliable than grasps planned based only on a point cluster.

There are two types of motion planners used by the object manipulation pipeline. The standard arm navigation motion planner is used to plan collision free paths. However, this motion planner cannot be used for the final approach to the grasp point since it will most likely think the grasp point will be in collision with the object being picked up. For the final grasp approach, an interpolated inverse kinematics motion planner is used, which will move the gripper linearly from the pre-grasp point to the final grasp point.

The object manipulation pipeline also has the option of using tactile feedback both during the approach to the grasp point and during the lift. This is especially useful to correct errors when

executing a grasp that was planned based only on a partial point cluster. Unfortunately, the youBot does not have tactile sensors on its gripper.

Picking up an object using the object manipulation pipeline goes through the following steps [12]:

- The object to be picked up is identified using sensor data
- A grasp planner generates set of possible grasp points for the object
- Sensor data is used to build a collision map of the environment
- A feasible grasp point with no collisions is selected from the list of possible grasps
- A collision-free path to the pre-grasp point is generated and executed
- The final path from the pre-grasp point to the grasp point is executed using an interpolated IK motion planner
- The gripper is closed on the object and the object model is attached to the gripper
- The object is lifted using the interpolated IK motion planner to a point where the collision free motion planner can take over

2.1.6 tf

tf is a ROS package used to keep track of multiple changing three dimensional coordinate frames. It provides tools for changing between any two coordinate frames that are being published. tf also allows multiple types of data to be transformed in this way including all types of ROS pointclouds, and points. The tf package provides tools for new transformations to be published easily using the sendTransform call or a static transform publisher node. Finally it provides some tools such as view_frames and tf_echo for transform visualization and debugging [13].

2.1.7 YouBot Overhead Cameras

The YouBot Overhead Camera stack provides autonomous navigation using an A* algorithm. It contains two sub packages, youBot overhead localization and youBot overhead vision. YouBot overhead localization implements the logic and classes for the navigation. It also provides a path planner server which takes x y coordinates as goals. The path planner then drives the robot until it reaches that point or another goal is received. The one problem with the path planner server is that the coordinates it takes in are in term of pixels relative to the camera images [14]. The youBot overhead vision serves mostly as a backend to the localization. It takes the overhead camera data and uses the user set calibration to perform color subtraction. This allows the stack to keep track of the robots position and orientation as well as find obstacles. It also provides the tools for creating the calibration [15].

3 Objectives

Our goal of providing an always-available online human-robot interaction laboratory poses a series of challenges. Operators should still be able to effectively control the robot regardless of the initial state of the environment the robot is in. In addition, our laboratory presents multiple surfaces that a user may wish to interact with objects on, each of which presents a surface of a unique size and height from the floor.

This project's goal was to satisfy these needs using a Microsoft Kinect mounted on our KUKA youBot's arm. More specifically, the following objectives were identified:

- *The algorithm must be robust.* Object discovery should work from most viewing angles of an object or surface, and should not be affected by the initial pose of the objects or the robot. Object discovery must also not be affected by the height or size of a viewed surface.
- *The algorithm must distinguish between objects and surfaces.* Objects sitting on a surface, like the floor or a table, must be reported as distinct entities.
- *The algorithm must be able to identify surfaces that could support an object.* The vision software must be able to identify suitable locations for placing previously-grasped objects.
- *Captures of discovered objects must be reasonably correct.* Internal representations of discovered objects must be correct enough to allow for effective grasping with the youBot gripper.

We searched for existing ROS packages that could discover objects given depth information, and found the tabletop_object_detector [8] package and the Object Recognition Kitchen [7]; unfortunately, neither of these packages proved robust enough to satisfy our requirements. As a result, we devised an algorithm that would satisfy our requirements and implemented it as a ROS stack.

The following sections will discuss the methodology used to accomplish these objectives and how well the method works in our laboratory environment.

4 Methodology

4.1 Kinect Object Detection

Detecting objects and planes with the Kinect facilitated many of the other parts of the project, including picking up, placing, and moving toward objects. Since it was integral to many of the other systems this component was developed early in the project. There are a few main components to our object detection code.

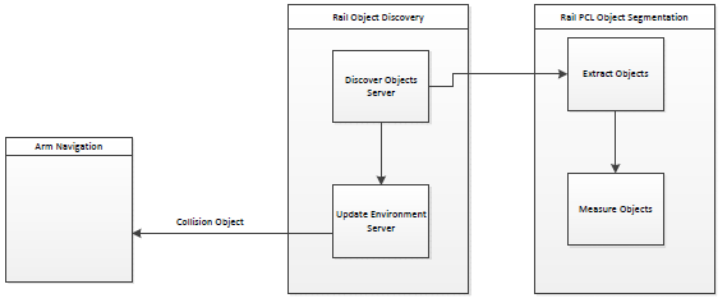

Figure : Object Detection class structure

We organized the code so the main out-looking interface is the discover_object_server. The server takes in the constraints on what objects to find and the max slope of any planes to return. Originally we had the server read a single message from the pointcloud sensor topic. This caused a problem where the server would not recognize there was a pointcloud to read and subsequently crash. We then switched to having the server subscribed to the sensor topic and store the most recent point cloud. The first action the server takes after receiving a message is to convert the current point cloud to be relative to the base_footprint frame using tf. Next the server passes the new cloud, constraints and max plane incline to the extract_objects server.

The extract_objects server takes those inputs and returns the objects and planes present in the input cloud that satisfy the constraints. This is performed using planar extraction and point clustering with a Kd-tree. More details on the extract_objects server and the algorithms used can be found in Paul Malmsten's MQP report: Object Discovery wtih a Microsoft Kinect [16].

The last step of the discover_object server is to add the discovered objects and planes to the collision environment. The collision environment is used by the arm_navigation to prevent the arm from hitting into objects. The server names all the found objects and planes and feeds them to the update_enviorment_server. Our environment server puts each of the named clouds into the planning environment which is checked later by the arm navigation.

The last aspect of our object detection is how we attached the Kinect to the robot. Our first plan was to build a mount to place the Kinect well above the back of the robot. This allowed us a good view of both the floor and tables. However it forced us into one static viewing angle and the arm and the front of the robot also blocked our view of objects close to the front of the robot. Finally the mount blocked the overhead camera calibration and was prone to vibration. All of these made this mount impractical. Our next mount location was facing forward on the arm just behind the gripper. This allowed us to view different areas by moving the arm. The largest problem with this position is that most of the ways we oriented the arm to look at objects brought the Kinect to close to see. This was an especially large problem for viewing objects on tables. The location we finally settled on was the same position on the arm but facing backward. This allowed much better viewing distances and angles for both objects on tables and on the floor.

4.2 Arm Navigation

One of the early major goals of our project was to get the newly installed arm on the youBot moving. While the low level controls, such as direct joint control, existed, there was no inverse kinematic software for the arm. We took advantage of the ROS arm_navigation open source stack in order to create several levels of inverse kinematic calculators. Included with the arm_navigation stack, these kinematic tools varied from simply avoiding self-collision, to avoiding other detected obstacles in the environment.

We decided to take advantage of the existing manipulation code developed within ROS, and implemented the arm_navigation stack. This stack was designed to generate the configuration files for the manipulator motion from the .urdf robot description file. A configuration wizard is included as a part of this stack, which allows for the selection of the joints that compose the arm. A sample screenshot of the configuration wizard is displayed below, as run on the youBot. By selecting the links that make up the arm, the configuration wizard uses the physical robot description provided to generate the files it needs to calculate the forward and inverse kinematics for that arm.

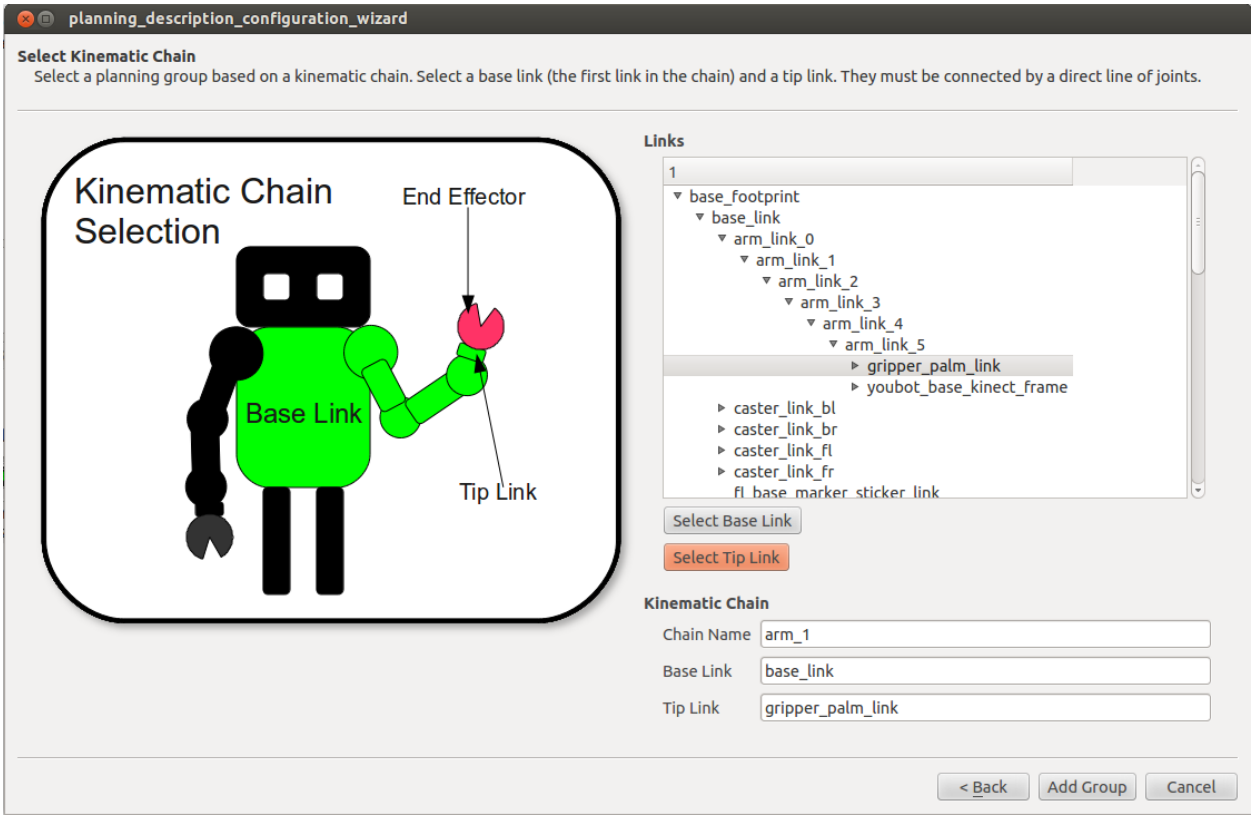

Figure : Arm navigation planning description configuration wizard

Once the arm's configuration had been generated, we had access to several useful kinematic services. The most straightforward of these was constraint aware inverse kinematics, which allowed us to specify a goal pose, and returned a trajectory of all of the joints to reach that location. This service also avoids any path that would cause the arm to collide with itself. This was used in several places as a quick but robust method to generate arm joint trajectories. [17]

A level higher than that is the move arm package, which actually takes the physical environment into account as well as self-collision when planning the joint trajectory [18]. This is the most robust kinematic solver included in the arm_navigation stack, but also has the highest computational time among the movement options. Like the constraint aware inverse kinematics, a goal pose is specified as part of a MoveArmAction, and a set of joint trajectories is returned. In fact, the move arm package uses the constraint aware inverse kinematics, after adding constraints that signify the environmental obstacles. This particular kinematic solver is used heavily in the pickup and place actions, to position the arm in the pre-grasp or pre-place pose.

Both of these kinematics solvers generate joint trajectories, an array of positions, velocities, and accelerations for all links in the arm to reach the desired location. To use this properly, the joint trajectories had to be sent to a joint trajectory action server, which essentially steps through the trajectory to move the arm properly. This ensures that the arm follows the path that the inverse kinematics solvers produced as closely as possible, as simply setting the joints to their final

configuration might cause them to collide with other links or environmental obstacles. Thus, a good joint trajectory action server is required to really take advantage of the features of youbot_arm_navigation.

Once the arm configuration wizard was run and the arm_navigation code was able to be run, the system was functional, but somewhat buggy. A big chunk of the early issues stemmed from overly tight constraints on the orientation of the desired goal position. As the youBot arm is a 5 degree of freedom arm, yaw is constrained by the x and y position of the goal, rather than being able to be set, as the arm_navigation code initially assumed. Once we wrote the code to calculate the yaw based on the desired position, the arm was able to reach all poses around the base.

4.2.1 Other Components or Arm Navigation

Arm navigation requires very few other services to run properly, as it is intended to be a low-level calculation tool. Specifically, it requires an urdf model of the robot in order to run the configuration wizard on. This gives the arm navigation stack the physical data required to calculate the joint limits and positions. As the robot configuration that the urdf file contains is the one that the arm's inverse kinematics will be calculated with, this model should be up to date and not change. If it does change, then the configuration wizard has to be run again, as we found out when we added the Kinect to the youBot arm. This resulted in bumping the Kinect against the robot frame until we ran the wizard on the updated urdf file.

Once the arm navigation stack was implemented, actually calculating the paths to any given goal does not require any outside components. However, actually transitioning the trajectory into motion requires a trajectory action server, as mentioned above.

4.3 Object Manipulation

One of the primary goals of our project was giving the youBot the ability to pick up and place objects. The ROS object manipulation stack provides a framework for pick and place actions, including standardized message types for describing the object you want to pick up or a grasp to be performed [19]. We had to add several robot-specific components to get the object manipulation pipeline to work with the youBot, including a grasp planner and our implementation of the arm navigation stack discussed in section 4.2.

4.3.1 Gripper Position Selection

The standard gripper on the youBot consists of two parallel fingers with a range of 2.3 cm. There are three mounting points for the gripper's fingers to which allow it to pick up a larger variety of objects.

The first gripper position allows the gripper to close fully, with a range of 0 cm to 2.3 cm. This allows the youBot to grasp objects that are less than 2.3 cm wide, such as an Expo marker. This position also makes it possible to grasp objects such as cups or bowls by grasping the rim of the object.

The second gripper position gives the gripper a range of 2 cm to 4.3 cm. This allows the youBot to grasp most of the blocks and cylindrical objects in our environment as well as objects such as books when placed on end. This position has a larger variety of objects that can be grasps using standard external grasps, but no longer allows cups and bowls to be picked up by the rim.

The third gripper position gives the gripper a range of 4 cm to 6.3 cm. This allows some larger objects to be grasped, but in this position the fingers are only connected with one screw, leaving them unstable which could result in inconsistent grasps.

We decided to use the second gripper position for the youBot's gripper, as seen below in Figure . This position gives us more options for objects to pick up than the other two, and does not have the stability problems of the third position. The first position would limit the graspable objects to very small objects and objects with rims. Grasping the rim of an object makes grasp planning more difficult, especially when using only a partial point cloud cluster to represent the object. Very small objects are also more difficult to find using object detection. We found that the second position gave us a good variety of objects that the youBot would be able to grasp, and also eliminated the problem of planning grasps for the rim of an object.

Figure : The youBot's gripper

4.3.2 Grasp Planning

The object manipulation pipeline supports using two different types of grasp planners. If the model of the object you want to pick up is known, a set of good grasps for the object can be precomputed and stored in a database, using software such as the "GraspIt!" simulator [20]. A database grasp planner can then simply find the model in the database that matches the target object and use the precomputed grasps. However, this requires that every object in the robot's environment has an accurate 3D model that grasps can be precomputed against. It also requires

accurate object recognition to choose the correct model that corresponds to the object you want to pick up.

The second type of grasp planner plans grasps based on the point cloud cluster that represents the target object. This type of cluster grasp planner is more flexible than a database grasp planner since it does not require a 3D model of the object to be picked up or object recognition. The downside is that the point cloud cluster for an object as seen by the robot's sensor is usually an incomplete representation of the object, which results in grasps that are generally less effective than the optimal grasps precomputed for the object's model. For the youBot, since we did not have accurate models of the objects in our environment and did not have object recognition, we designed a cluster grasp planner. Even if we had object recognition and an effective database grasp planner, the cluster grasp planner can still be used when object recognition fails or for objects that are not in the database.

4.3.2.1 Overhead Grasps

For the first version of our youBot grasp planner, we focused on finding valid overhead grasps for an object. The grasp planner first uses a cluster bounding box service to calculate the bounding box for the object's point cloud cluster. The bounding box is oriented such that the object's x-axis is aligned with the direction of the largest point cloud variance. Each overhead grasp is located at the same point directly above the object's origin. The vertical offset of the grasp is calculated using the height of the object, the gripper length, and the gripper offset from the center of the palm link. For both the x and y axes, if the bounding box dimension is within the gripper range, a grasp aligned with the axis as well as one rotated 180 degrees is added. This generates up to four possible overhead grasps for each object, each with the gripper aligned with the edges of the object's bounding box. In addition to finding the grasp point for an object, the grasp planner also sets the approach and retreat distances for the grasp and sets the gripper postures required to pick up the object, which are based on the object's width. An example the youBot picking up an object with an overhead grasp can be seen below in Figure .

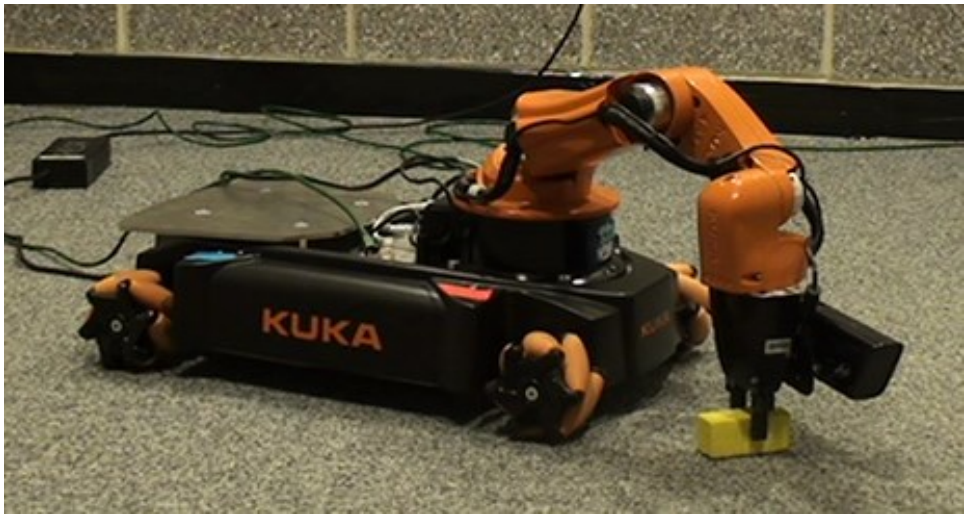

Figure : The youBot picking up a rectangular block with an overhead grasp

4.3.2.2 Angled Grasps

While overhead grasps are effective for many objects, the range of the arm can be increased if angled grasps are also used. Angled grasps are also more likely to work on tables or other raised surfaces. Since the KUKA youBot's arm is a five link serial manipulator with five degrees of freedom, it cannot reach any arbitrary six degree of freedom position and orientation. This raises problems when attempting to pick up objects with angled grasps.

When developing a grasp planner that uses angled grasps, we first attempted to use the set of grasps aligned with either the x-axis or y-axis of the object for angles between -90° and 90° from vertical. However, these grasps were almost never reachable by the youBot's 5 DOF arm. Our solution to this problem was to only use grasps that are aligned with the first joint of the robot arm. This constraint generates a set of grasps that are reachable by the 5 DOF arm. However, since the grasps are no longer aligned with the object's axes, they are only effective for cylindrical objects. For this reason, we only compute these grasps if the ratio of the x and y dimensions of the object's bounding box is below a certain threshold, in which case we assume the object is a cylinder. The yaw angle that is required for an angled grasp to be aligned with the first joint of the arm is calculated using the x and y position of the object relative to the first joint of the arm. To convert between coordinate frames for these calculations, we use the ROS tf package discussed in section 2.1.6 [13].

Figure : The youBot picking up a cylindrical block with an angled grasp

4.3.3 Other Components Required for Object Manipulation

The object manipulation pipeline requires some other robot-specific components in addition to a grasp planner and arm navigation to work with the youBot.

4.3.3.1 Hand Description

A hand description file is required for the object manipulation node to know the details about the youBot's gripper. This file contains information including the arm name, the coordinate frames of the gripper and robot base, the joint names for both the gripper and arm, the links in the gripper that are allowed to touch the object, the link that objects are considered attached to when picked up, and the approach direction of the gripper.

4.3.3.2 Grasp Hand Posture Execution Server

The object manipulation node requires a server to control the posture of the gripper during a grasp. The object manipulator node sends an object_manipulation_msgs::GraspHandPostureExecutionGoal message to the grasp hand posture execution server when it needs the gripper to move. The server then converts this message to a brics_actuator::JointPositions message, which is used by the youBot's drivers, and publishes it to move the gripper to the appropriate pose.

4.3.3.3 List Controllers Server

The object manipulator node also uses a service that lists the controllers in use and their statuses. Our list controllers server for the youBot will return the controllers for both the arm and the gripper, and will list them as running if at least one node is subscribed to the controller topic.

4.3.3.4 Interpolated Inverse Kinematics

Interpolated inverse kinematics is used when determining the approach and retreat trajectories for the grasp. We used the interpolated IK node from the experimental arm navigation stack [21] with minor changes to make it compatible with the inverse kinematics for the youBot arm.

4.3.3.5 Joint Trajectory Adapter

The object manipulator node uses publishes the joint trajectories to move the arm using a different message type than the youBot drivers expect. The object manipulator node uses pr2_controllers_msgs::JointTrajectoryAction messages, while the youBot drivers expect control_msgs::FollowJointTrajectoryAction messages. These messages both contain the trajectory to follow, however the control_msgs::FollowJointTrajectoryAction messages also allow you to specify tolerances for the trajectory. We created a simple adapter to convert between these message types to make the object manipulator node compatible with the youBot drivers.

4.3.4 Object Placement

The process of placing an object is very similar to picking up an object, only in reverse. To specify a place action we need to know the grasp that was used to pick up the object relative to

the object's coordinate frame, and the location to place the object relative to the robot. If multiple place locations are specified, the first one that is feasible will be attempted.

We created an object place location server that makes it easier to specify a place action. This server subscribes to the pickup action goal and result messages from the object pickup. The pickup action goal message includes information about the object that was picked up, while the pickup action result message includes the grasp that was used and whether or not the grasp was successful [22].

The grasp pose relative to the object is calculated by multiplying the inverse of the original object pose relative to the robot from the pickup action goal message with the grasp pose relative to the robot from the pickup action result message, using functions from the tf package [13]. This transformation is combined with the desired object pose to find the place location relative to the robot's base frame.

Our object place location server also calculates the height to place the object. When the server receives the pickup goal message for an object, it finds the support surface for the object in the collision environment, and uses it to calculate the object's height above the surface. When an object is placed, its height is set to the height of the new support surface, plus the object offset above the previous surface calculated earlier, plus a small offset so the object will be dropped slightly above the surface. If a user specifies the height of the object in their call to the place location server, that object height will be used to place the object.

4.3.4.1 Placement with Angled Grasps

When placing an object that was picked up with an angled grasp, we need to constrain the place angle to align it with the first joint of the arm like we did when picking up the object. When placing with angled grasps, the transformation for the grasp relative to the object is rotated to counter to yaw of the original grasp. We then calculate the yaw of each place location using the desired x and y coordinates relative to the first joint of the arm. This creates a place location that is reachable by the 5 DOF arm when the two transformations are combined.

4.4 Navigation

The navigation portion of our project was intended to allow easy and accurate driving by both other processes and users. By easy we mean that the input into the program would be intuitive and simple to figure out. As discussed previously the youbot_overhead_vision and youbot_over_head_localization stacks provided good path planning and localization, but required calibration for the overhead cameras and simpler input for the navigation goals. Specifically we wanted to input meters relative to the robot instead of pixels relative to the overhead camera views.

4.4.1 Calibrating the Overhead Cameras

The first step in calibrating the overhead cameras was to consider the lighting in the room. Previously the room was lit by the overhead fluorescent lights, however the camera mount is directly below the overhead lights so whenever a bulb burnt out the cameras would have to be moved and then recalibrated.

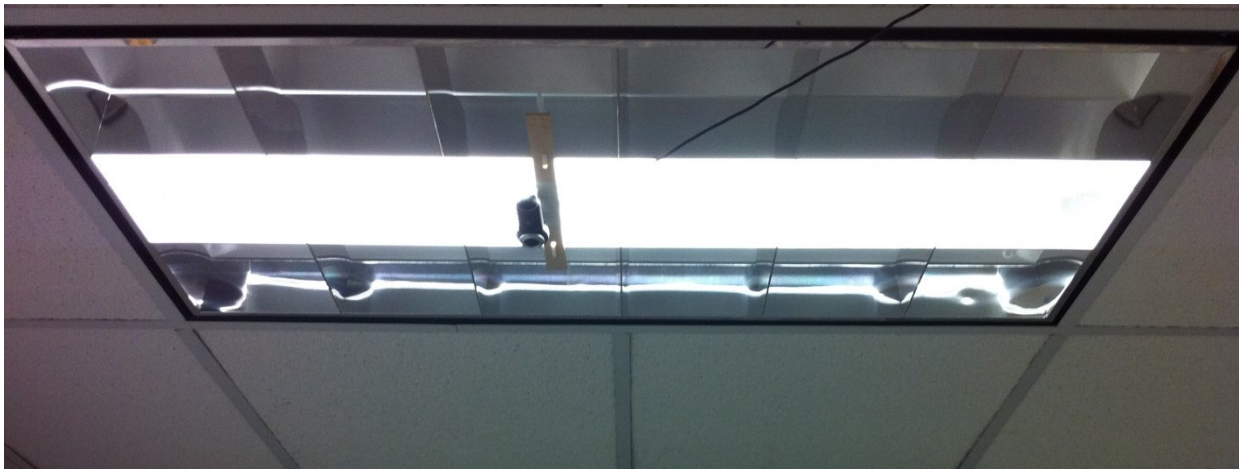

Figure : Overhead camera mount

To mitigate this we switched to using four light mounts in a rectangle, pictured below. This allowed us to replace burnt out bulbs without recalibrating.

Figure : Overhead light setup

We tried three different types of bulbs to choose the best type of lighting. The types were: 26 Watt 1750 Lumen CFLs, 15 Watt 750 Lumen indoor floodlights, and 19 Watt 950 Lumen GE Reveal CFLs. The main two concerns when deciding what lighting to use was the amount of glare the lights caused, and more importantly how close the color of the light was to white. The glare is important because it can drown out the rear two color markers, making navigation unreliable. The color matters because the entire room including the robot markers gets shaded the same color as the light making it harder to distinguish between the markers during color subtraction. A third factor was how even the light was across the room. Even lighting was useful because it means that the color change as much from location to location in the room

The overhead florescent lights served as a good baseline for what we were looking for. They provided even white light. The only problems were the amount of glare and the overriding concern of the position of the lights.

Figure : View from overhead camera using the original lighting

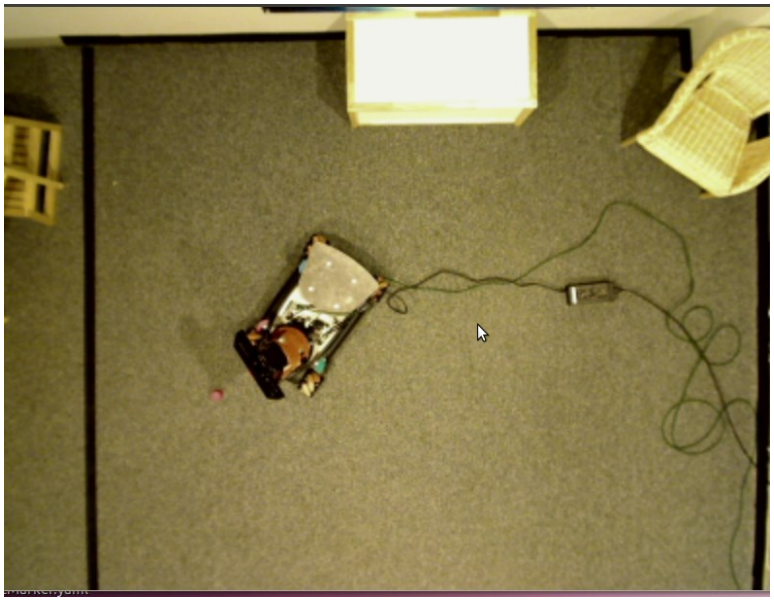

Figure : View from overhead camera using 1750 Lumen CFL bulbs

Compared to the overhead cameras the 1750 Lumen CFL bulbs were worse in every aspect. From the picture it is obvious that the light has a large orange tint compared to the original lighting, also looking from the top left to middle of the frame there is a large difference in the brightness which is not ideal. Finally though it is not apparent from the image the CFL bulbs do cause around the same amount of glare as the overhead lights.

Figure : View from overhead camera under indoor floodlights

The indoor floodlights were an improvement on the 1750 Lumen CFL bulbs in both glare reduction and the evenness of the lighting. However they had the same problem of strongly tinted light, also while they reduced glare they were almost too dark to work well.

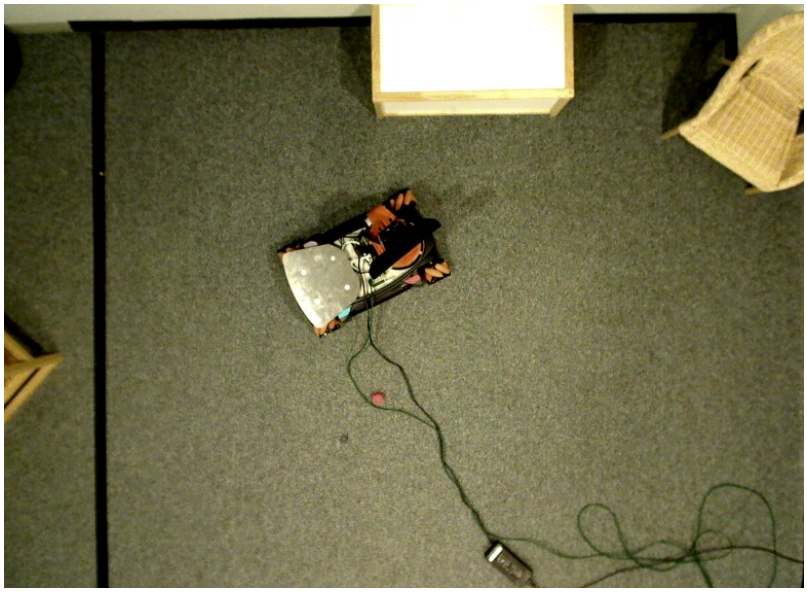

Figure : View from overhead cameras using 950 Lumen CFLs

The 950 Lumen GE CFLs did cause uneven lighting. However the color of the light, while not as good as the original lighting, was an improvement over the other too replacement light sources. Additionally these lights caused less glare than both the original lighting and the 1750 CFL bulbs. In the end we selected these bulbs for the project over the other two because the color of the light was the primary consideration.

After the lighting we performed two main steps to calibrate the overhead cameras, the first was calibrating cameras themselves with the guvcview utility and the second was to calibrate the color subtraction for the navigation stacks using the calibration application in the youbot_overhead vision package. The purpose of the camera calibration was to change the camera properties such as the frame rate, resolution, and camera output data type. We also used the camera calibration to change values like the brightness and contrast to make the images from the cameras similar for users and also to make the color subtraction more accurate.

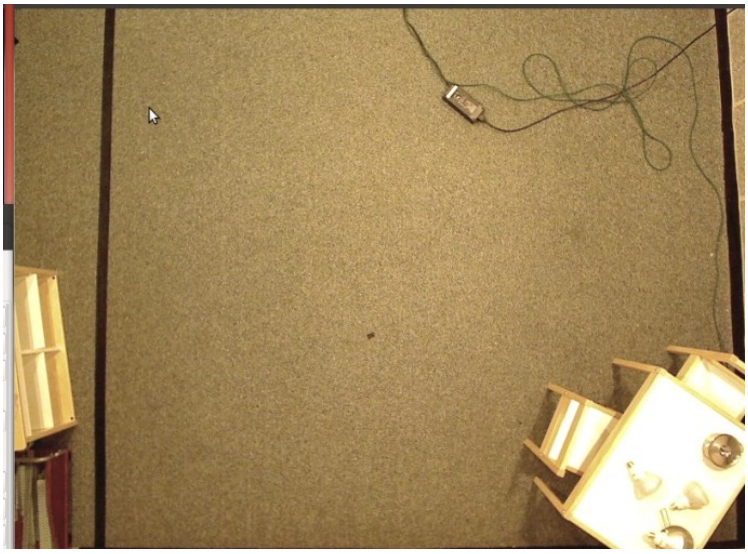

Figure : View from bottom camera without calibration

The major values we changed for the camera properties were the auto exposure and auto focus, as well as the resolution and output type. We set both the auto exposure and auto focus to false to prevent the exposure and focus from changing during operation and messing up the color subtraction calibration. We set the camera output YUYV and the resolution to 800X600 to integrate well with the ROS stacks. The other values were set through trial and error mostly to minimize glare and maximize the difference in color between the four markers on the robot base. One stumbling block in the calibration of the cameras is that they reset every time the computer is shut down. The simple solution is to remember to reload the calibration files, we were unable to find a way to automate this or keep the calibration through shut down.

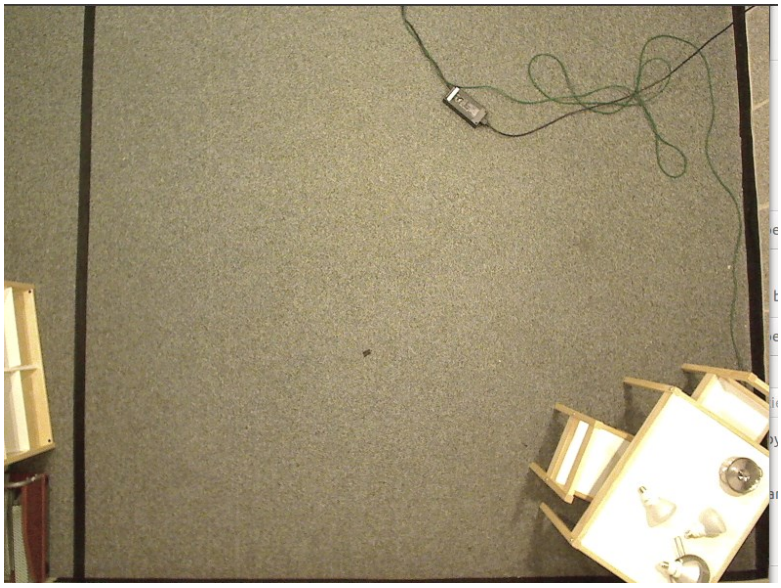

Figure :View from bottom camera after calibration

The calibration of the color subtraction is both more simple but also more time consuming. The first step in setting up for the calibration is to publish the images from the overhead cameras to ROS topics. This is also necessary to use the stack for actual navigation. We launch two usb_cam_node nodes from a launch file. The nodes read the video from /dev/video0 and /dev/video1 and publish to the ROS topics /logitech_9000_camera1 and /logitech_9000_camera2. The calibration tools and navigation both listen to these topics for the overhead camera information. The one problem with this method is the cameras can switch weather they are video0 or video1 whenever the computer is restarted. This is a problem because the youbot_overhead_vision stack is hardcoded so the camera1 topic always corresponds to the top camera. None of this is necessary in the simulated gazebo environment because it handles publishing the camera internally.

The next step in the overhead navigation calibration was the color subtraction. The first step in the color subtraction was to subtract out the floor. However we tried two different ways of calibrating the location of the color markers. The first method we tried was to for each camera we moved the robot to four different positions. At each location we rotated the robot four times, 90 degrees each time. At every position and orientation we calibrated the color of the markers on the robot. The other method we tried was to calibrate the markers at one position then drive the robot some distance and calibrate the markers again. We repeated this until we had good coverage of the room and the robot was consistently able to find its position after the drive without calibration. Second method resulted in a more robust calibration and was also easier to perform.

4.5 Implementing a Better Interface

The largest problem that we had with the existing youbot_overhead_localization stack was that the input for goal coordinates was in pixels relative to the overhead camera image. To improve this we wrote another ROS action serve to translate locations in meters to pixels and pass them on to the path planner. This server is in the move_server.cpp file under the rail_movement package. At first we tried using the coordinate_conversion server provided in the overhead vision package. However we had problems with it returning incorrect output and switched to simply inverting the math used by the path planner to convert from pixels to meters.

The other aspect we wanted to change about the overhead navigation system is that it took world coordinates as input for the goal point and we wanted to use coordinates relative to the robot. To accomplish this we changed the move_server to add a tf transform point cloud call. We also had to write a tf publisher to publish the transform between the robot base frame and the map coordinate frame for the transform call to work. We wrote the publisher to subscribe to the pose2d topic and publish that location as the transform from the map to the robot. This worked in simulation because there is no frame above the base_footprint.

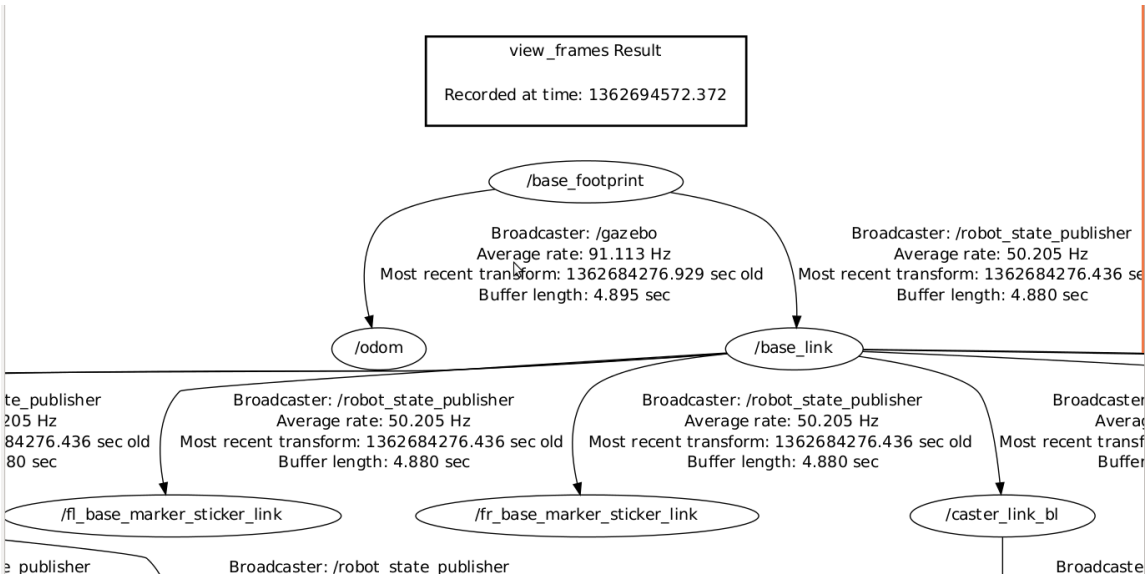

Figure : Simulation robot tf tree

On the real robot there is an additional /odom frame which prevents this approach because tf allows only one parent frame.

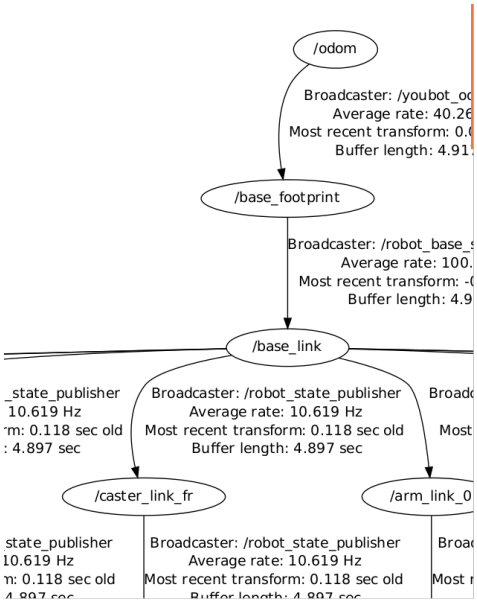

Figure : Real robot tf tree

Our solution was to publish the map frame as a separate tree with a robot frame. The robot frame is identical to the base_footprint frame in location but there is no connection between them.

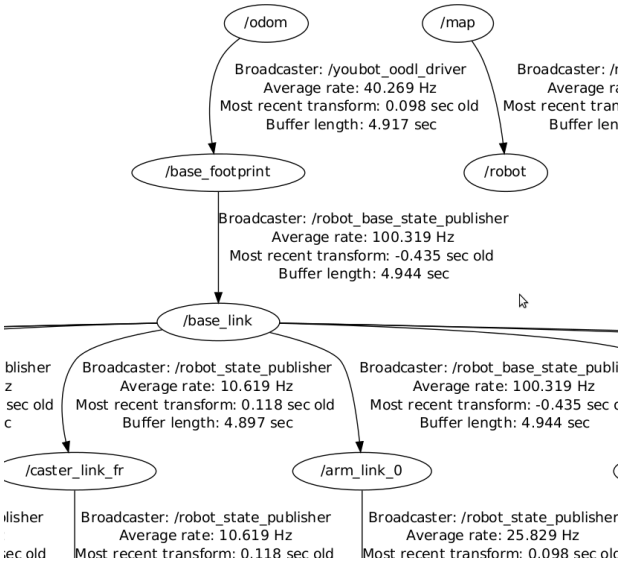

Figure : Real robot tf with map frame

The disadvantage of this approach is that it is not possible to directly convert from one of the other frames, such as the Kinect or arm, to the map frame. On the other hand it allowed the map frame to be above the robots, which makes sense conceptually, and was easy to implement.

5 Results

5.1 Graspable Objects

We tested the youBot's ability to pick up the objects shown in Figure . For this test, we placed each object centered approximately 15 cm in front of the youBot, within overhead grasping range. Table below shows the success rates for each of the objects tested. All of the objects chosen had sizes within the gripper's range. Some objects, including the expo eraser and book, are only graspable when in certain orientations, since the dimension of some of their edges is too large for the youBot's gripper. If the robot is being controlled remotely, and one of these objects falls into an orientation that is not graspable, the user will not be able to pick up the object or move it to a position that is graspable. Two of the objects tested were not successfully picked up by the youBot. The small wood block was too small to be found by the object detection, and therefore couldn't be picked up. The screwdriver was partially picked up by the youBot three of five times, lifting the handle off the ground. However, the screwdriver slipped out of the gripper before it could be fully lifted. The average success rate for the objects tested, excluding the objects that were never successfully picked up, was 88%. During the failed pickups for these objects, the gripper was usually close to making a successful grasp, but was slightly off and hit the object during the approach.

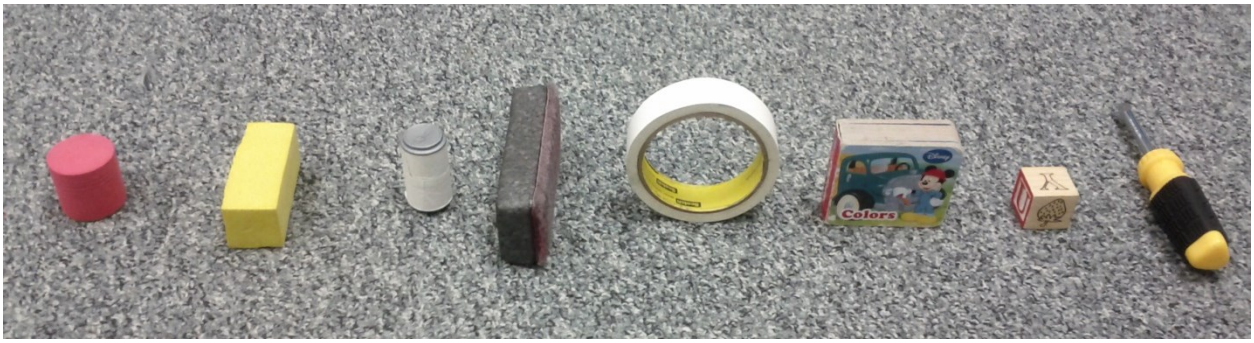

Figure : Objects used for the grasping tests

Table : Success Rates of Picking up Various Objects

Object	Attempts	Successful	Success %	Notes
Red Cylindrical Block	10	9	90%	
Yellow Rectangular Block (aligned with robot)	5	5	100%	
Yellow Rectangular Block (rotated 90 degrees)	5	5	100%	
Yellow Rectangular Block (rotated 45 degrees)	5	4	80%	
Toy Can	5	4	80%	
Expo Eraser (on long edge)	5	5	100%	Not graspable when laid flat
Expo Eraser (on short edge)	5	3	60%	Not graspable when laid flat
Masking Tape (on end)	5	4	80%	Not graspable when laid flat
Small Book (on end)	5	5	100%	Not graspable when laid flat
Small Wood Block	3	0	0%	Too small to be recognized by object detection
Screwdriver	5	0	0%	Lifted handle off ground 3 times without a good enough grip to fully pick up

5.2 Arm Workspace for Grasping

We tested the workspace where the youBot can successfully pick up objects off the floor, and compared the workspace when using only overhead grasps to the workspace when angled grasps were allowed. For these tests we used the red cylindrical block, which is graspable by both overhead and angled gasps. The workspace for picking up objects is circular, centered at the first joint of the arm. Using overhead grasps, the block can be picked up at a distance of up to 31.5 cm from the first joint of the arm. This corresponds to 18 cm from the edge of the robot if the object is centered in front of the youBot.

Next, we tested the workspace for picking up the same block using angled grasps. We found that the block could be picked up with angled grasps at a distance of up to 42.3 cm from the first joint of the arm. For objects centered in front of the robot, angled grasps increase the pickup range to 28.8 cm from the front of the robot, which is a 60% increase over using only overhead grasps. Objects that are not cylindrical are only graspable using overhead grasps, limiting the maximum range that they can be picked up from

The minimum range for picking up objects is depends on several factors, making it not well defined. For an object to be picked up, it needs to be visible by the Kinect and must be recognized by the object detection. If the object is too close to the robot, it may not be successfully recognized or may be combined with part of the robot by the object detection. The grasp must also be reachable without collision for a grasp to be attempted. The Kinect, which is mounted to the last link of the arm, would be in collision with the robot body during many of the grasps for objects very close to the robot.

5.3 Table Pickup

Picking up objects off a table or other elevated surface is more difficult than picking up objects off the floor. The increased height approaches the edge of the arm's reachable workspace, making overhead grasps no longer reachable by the arm. Since table pickups need to use angled grasps, they are generally only effective for cylindrical objects, since the grasps must be aligned with the base of the youBot's arm to be reachable. In addition to the grasp position being reachable, the approach and retreat must also have valid inverse kinematics solutions. We reduced the approach and retreat distances which makes them more likely to stay within the arm's reachable workspace. It is also important that the table is correctly added to the collision environment to avoid attempting grasps that would cause the arm or Kinect to collide with the table. On the floor, grasps where the Kinect is below the arm are usually used. When picking up object off the table, grasps with the Kinect above the arm, like the one shown below in Figure , keep the Kinect away from the table. However, due to joint limit limitations, this Kinect position can only be achieved when the first joint of the arm is facing backwards, which limits the arm's range. The Kinect cannot be in the position above the arm if the object is centered in front of the robot, since the first joint cannot rotate a full 360°. Object detection is also more difficult on tables. When we use the same arm position for the Kinect to look at the objects as we use for objects on the floor, the Kinect is too close to the object on the table to see it due to the sensor's minimum range of approximately 0.8 m [23]. To overcome this problem, we added an arm position for table pickups that moved the Kinect farther from the object. With these modifications, the youBot can successfully grasp cylindrical objects off tables as shown in Figure .

Figure : The youBot picking up a block off a table

5.4 Object Placement

On the floor, the youBot can consistently place objects that were successfully picked up as long as the specified place location is reachable by the arm. Objects are released about 2 mm above the floor, which avoids collisions with the floor before releasing but is low enough so the block falls into the desired location without bouncing. The orientation for the object to be placed can be specified if the object was picked up with an overhead grasp. This is demonstrated in Figure , which shows the youBot after placing the yellow block at a 45° angle. For objects picked up with angled grasps, the place orientation cannot be specified. An example of the youBot placing an object that was picked up with an angled grasp can be seen in Figure .

We could not get placement on tables to work correctly. Since the table is at the edge of the arm's workspace, we were unable to find a place location that was reachable by the arm. Placement may be more effective on a lower table than the 29 cm table we used. Table placement may also be more effective if we removed the constraint that the object must be placed at the same orientation as it was picked up in. This would allow the object to be placed at an angle that is reachable by the gripper, but would place the object in an unstable orientation that would fall into an unpredictable position. Even though table placement does not work, objects picked up off a table can be placed in a different location on the floor.

Figure : The youBot after placing a block at a 45° angle

Figure : The youBot placing a cylindrical object that was picked up with an angled grasp

5.5 Kinect Calibration

The calibration of the exact position and orientation of the Kinect sensor on the youBot's arm plays a major role in the success of an object pickup. If the Kinect calibration is not accurate, the position of objects it detects will also not be accurate. When the youBot tries to pick up an object, it will miss if the Kinect calibration is incorrect, often hitting the object with the gripper during the approach to the grasp point. We calibrated the Kinect's position manually, which does not result in an ideal calibration. Also, if the Kinect is bumped it may move slightly, which

changes the calibration. An accurate automatic method of calibrating the Kinect would solve these issues. One possible automatic calibration method would be to recognize a part of the robot's base with a known position, and use its location as seen by the Kinect to set the calibration. Implementing some type of automatic calibration method is recommended for future work.

5.6 Object Detection

We measured the accuracy of the object detection code by placing a few of our objects at set positions and then recording the distance the Kinect detected them at. The first set of positions was a row two meters away from the robot, one directly in front of the robot and the other two .75 meters on either side.

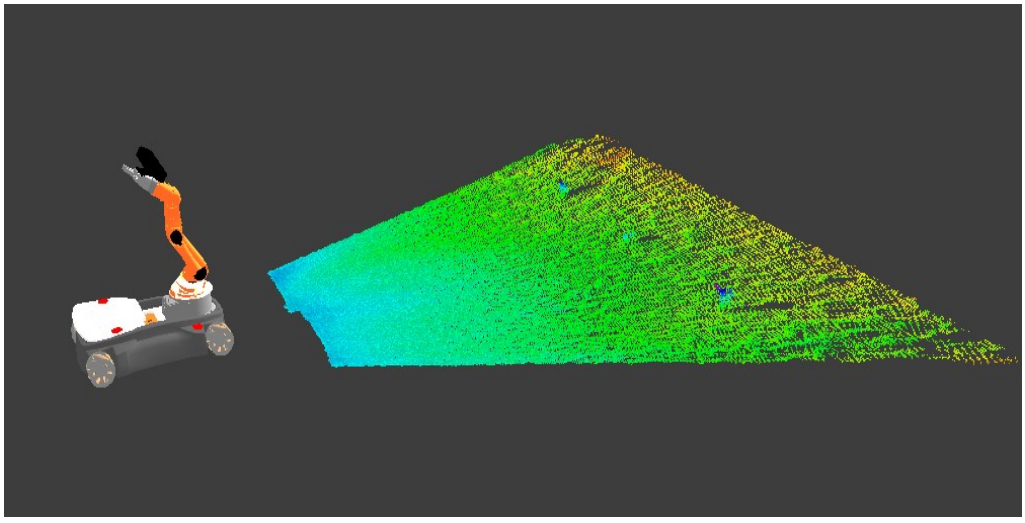

Figure : Far object detection test

The second set of positions was similar to the first but only one meter away with the other two positions only half a meter away on either side. We put the positions on either side closed for the short distance because the Kinect's field of vision narrows closer to the robot.

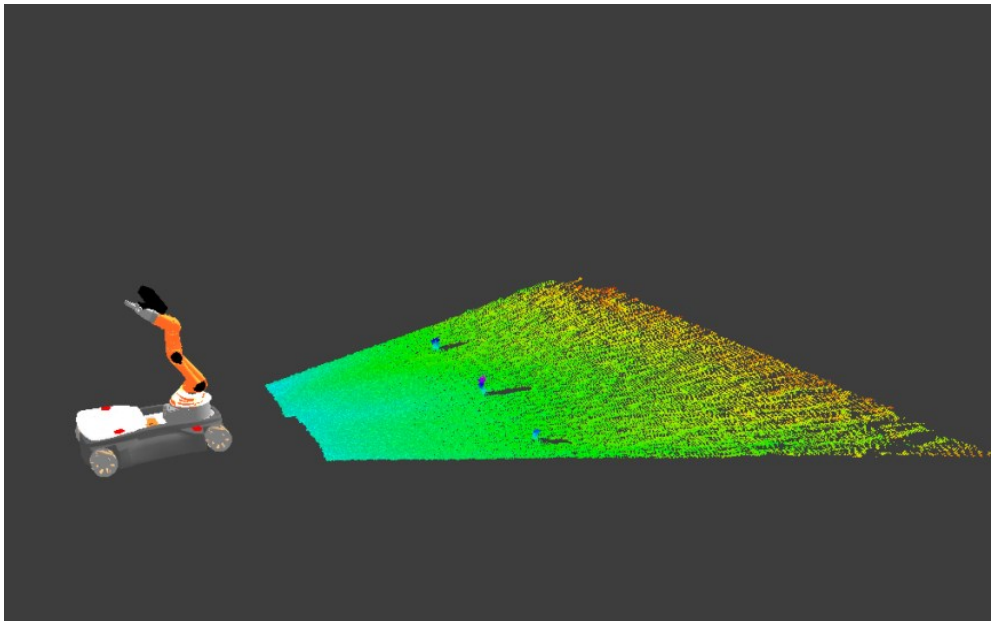

Figure : Close object detection test

The Kinect was able to detect each type of object at each position but with a small error. The Kinect measured distances are shown in the table below.

Table : Kinect Measured Object Distances

Locations:	Far Left		Far Middle		Far Right		Near Left		Near Middle		Near Right	
	x(m)	y(m)	x(m)	y(m)	x(m)	y(m)	x(m)	y(m)	x(m)	y(m)	x(m)	y(m)
Actual	2	0.75	2	0	2	-.75	1	0.5	1	0	1	-0.5
Soup Can:	2.04	0.84	2.04	0.06	2.08	-0.7	0.98	0.53	1.01	0.03	1.03	-0.46
Red Block:	2.05	0.84	2.04	0.06	2.08	-0.69	0.98	0.53	1	0.03	1.04	-0.45
Yellow Block:	2.04	0.84	2.04	0.06	2.08	-0.7	0.98	0.53	1.01	0.03	1.04	-0.45

Looking at the measurements we can see that the error for each position was almost constant for each of the different types of object at each position. Specifically the largest difference between the measurements for any two objects at any one position was only one centimeter. Another interesting aspect of the data is that the measurements for the y position were always skewed in the same direction. For example all the measured y positions for the far row were between .05 and .09 meters to the left of the correct position. In the near row all the measured y positions were similarly offset but only varied from .03 to .05 meters. This indicates that the Kinect was taking correct measurements but our calibration of where the Kinect was facing was slightly off.

5.7 Navigation

The goal of the navigation portion of our project was to provide accurate and reliable autonomous driving throughout the workspace. To quantify this we took two statistics, in the first we told the robot to drive one meter straight from a fixed point and recorded the actual distance driven. The second experiment was similar but instead of one fixed point we used locations throughout the room.

The Average accuracy of the fixed one meter drive was .99 meters; however this is not the most telling statistic as the robot went both over and under shot the goal point. Far more informative is the standard deviation which was .03 cm. This tells that around 95 percent of the time the robot will be within 6cm of the goal point when driving in a single direction. This is a reasonable accuracy. The average when driving from different points was .96 and the standard deviation was .017. While the standard deviation was lower it should be noted that is because the actual drive distances were clustered at slightly less than a meter. The overall result is that the navigation is reasonably accurate and works similarly well throughout the room.

5.8 Combined

We measured how well the navigation and arm worked with each other and the object detection with two tests. In the first we put the robot at a point and the object at another fixed point then instructed the robot to detect and move to the object. After the robot moved we measured the distance from the arm to the object and if the arm was able to pick up the object. The other test was similar but the robot and object's position were varied to test if the results were similar throughout the workspace.

In the first test the average resulting distance from the object was 27 cm and the standard deviation was 7 cm. Only one move out of five failed to reach a position where it could grab the object. That move resulted in a distance of 40 cm from the object. In the second test the average distance was 26 cm and the standard deviation was 8 cm. Similar to the first test only one of five failed to reach a position where it could grab the object. In that case it ended 41 cm away from the object. It is important to note that the goal for this test was not to reach exactly zero centimeters away as this would mean the object was under the base of the robot. Instead around 20 to 30 centimeters was an ideal range.

6 Limitations and Future Work

6.1 User Interface

The system is currently operated almost exclusively from the command line, which is effective for development purposes, but needs to be made user-friendly for online use. For this reason, an interface between the online component and the code developed as part of this MQP is an important future work. Before the portions that we created can be used as part of the final product, a simplified user interface is required, one that is easily controllable over the web interface.

6.2 Kinect Auto-Calibration

One of the major difficulties and time sinks during our development periods was dealing with Kinect miscalibration. Even a small bump could upset the Kinect's position on the arm, creating discrepancies of several centimeters when determining the object locations. While this was not a major difficulty for the navigation control, due to the small gripper size, these errors made it almost impossible to actually grasp objects. As the arm has very little margin for error, the Kinect requires some method of self-calibration. During the project, we made due with manual calibration, by placing objects in front of the robot and using visualization software to determine where the Kinect saw the objects and changing the calibration numbers manually.

An improved design should involve some known pattern that can be quickly used to calibrate the Kinect. Our suggested idea would be to place some kind of AR tag on the youBot's plate, and have the Kinect look at it upon startup to calibrate properly. This would help to maintain the arm's accuracy and ensure that pickup functionality is consistent.

6.3 Better Navigation

The current navigation code has some major issues that should be resolved to improve the entire system's functionality. The most glaring issue is the inability to select a facing direction at the desired destination. The youBot can make it to almost any point in the room, but there is no telling which direction the robot will be facing once it gets to that location. This leads to inconsistent results when attempting to pick up an object, as the robot may move close to the object, but be facing in a direction which does not allow the arm to actually reach the object, and there is no method to correct the orientation of the robot in the current implementation. As the robot's wheels are holonomic, there should be no issue with approaching a destination in any orientation, making the oversight particularly glaring.

The youBot localization code can also be improved substantially. The system uses the overhead cameras to find colored markers on the corners of the youBot, which is used to determine the robot's position. The issue comes from the fact that the lighting is not entirely consistent, requiring frequent recalibration of the color detection. For this reason, the localization should be

revised to be more robust, possibly with the use of an AR tag to determine the robot's position. This could be the same AR tag used for the Kinect auto-calibration, to reuse the same systems. Another possibility might utilize better cameras to hopefully reduce the error caused by changing light conditions.

6.4 Gripper Improvement

The standard youBot gripper is extremely limited in its versatility, which greatly hampers number of applications for the system as a whole. With only 2.3 centimeters of travel, there is very little margin for error in grasping objects. Given the maximum size of 4.3 centimeters, the gripper cannot grab any large objects, and many of the small objects that it can pick up can be difficult to detect with the Kinect. There is also no force feedback within the system, making every grasp attempt a blind reach, and precluding the attempt to grab fragile objects. For this reason, it would be beneficial to redesign or otherwise improve the gripper mechanism.

6.5 Power Connection

The youBot requires some form of improved power and internet connection. The current system is tethered to the wall, with the cords laying on the ground. These are obviously obstacles for the youBot, and especially given the fact that the robot uses holonomic drive, they often run the risk of throwing off the youBot's odometry, not to mention the risk of getting pulled out of the wall and disconnecting the youBot. For this reason, a system needs to be developed to keep the cords off the ground and out of the youBot's path. We considered working on this during our project, but decided to focus on implementing the robot software architecture instead.

6.6 Object Recognition

A possible improvement to the youBot's pickup and place functionality would be to add the ability to perform object recognition rather than just object detection. This would require adding a library of the common 3D models that the youBot can pick up, and attempting to match the model to the objects detected by the Kinect. This would provide several benefits, such as being able to pre-process the most efficient grasps for that object, providing greater feedback to the user by specifying which object is detected, and allowing for better recognition of obscured objects.

7 Conclusion

In this project we designed a system to allow for arm movement, navigation, and object manipulation for the youBot platform. To do this, we took advantage of existing ROS architecture that dealt with manipulation, and tailored it to the youBot's physical setup. By using these open source architectures, we were able to reduce our development time and focus on fine-tuning the robot's performance. We also modified the youBot hardware to include a Kinect and used this addition to perform accurate object manipulation.

The arm movement code is robust, taking into account both self-collision and environmental hazards to avoid damaging the arm. The system has also been optimized towards the youBot's specific manipulator, by constraining the system to be consistent with the arm's own 5 degree of freedom constraints. Several methods of controlling the arm were implemented, from a simple prompt for a desired pose, to a virtual interactive marker, to an automated system that moves the arm towards detected objects. Finally, a set of often-used arm configurations was created, to allow for easy movement of the arm to positions that provide the Kinect with a useful field of view, or minimize the chance that the arm will block the overhead detection's view of the corner markers.

The navigation system has been updated with an improved interface, in order to move based on the robot's local coordinates. Several systems were implemented to convert destinations detected locally by the Kinect into a drivable location in the world frame, despite major differences between the two systems. The lighting was also updated to be more consistent, improving system reliability, and the overhead cameras were calibrated accordingly. The navigation has also been connected with the object detection code, in order to provide for automated movement to a detected object.

Object manipulation code was implemented, to pick up and place objects detected by the Kinect. This involved extensive customization of the ROS open source object manipulation code, to determine and evaluate grasps based on the Kinect's 3D images. The final system can pick up objects both on the floor and on table surfaces that the arm can reach. Once it picks up an object, the youBot can place it in any reachable location on the floor, in the same orientation that it was picked up in. A simple prompt-based interface was also created to allow for quick access to this functionality.

All of these components were designed to function as a low-level control interface for the youBot as this project goes forward. While these functions are not at the point where they are easily accessible to a common user, they provide a solid framework on which to build a more complex system. This project took advantage of open source code and customized it to the youBot system, focusing on providing robust and consistent performance in these low level interfaces.

8 Bibliography

[1] Locomotec, "KUKA youBot User Manual," 6 October 2012. [Online]. Available: http://youbot-store.com/downloads/KUKA-youBot_UserManual.pdf. [Accessed 11 October 2012].

[2] Willow Garage, "Willow Garage - Software Overview," 2011. [Online]. Available: http://www.willowgarage.com/pages/software/overview. [Accessed 11 December 2012].

[3] Open Source Robotics Foundation, "Home - Gazebo," [Online]. Available: http://gazebosim.org/. [Accessed 8 December 2012].

[4] E. Amos, "Wikipedia - File: Xbox-360-Kinect-Standalone.png," 2 August 2011. [Online]. Available: http://en.wikipedia.org/wiki/File:Xbox-360-Kinect-Standalone.png. [Accessed 9 October 2012].

[5] Kolossos, "Wikipedia - File: Kinect2-ir-image.png," 20 April 2011. [Online]. Available: http://en.wikipedia.org/wiki/File:Kinect2-ir-image.png. [Accessed 8 October 2012].

[6] "ROS Wiki: openni_camera," [Online]. Available: http://www.ros.org/wiki/openni_camera. [Accessed 8 October 2012].

[7] Willow Garage, "Object Recognition Kitchen," 2011. [Online]. Available: http://ecto.willowgarage.com/recognition/. [Accessed 10 October 2012].

[8] Willow Garage, "ROS Wiki: tabletop_object_detector," 2012. [Online]. Available: http://www.ros.org/wiki/tabletop_object_detector. [Accessed 8 October 2012].

[9] Open Perception Foundation, "Point Cloud Library," [Online]. Available: http://pointclouds.org/. [Accessed 8 December 2012].

[10] Open Perception Foundation, "Euclidean Cluster Extraction," [Online]. Available: http://www.pointclouds.org/documentation/tutorials/cluster_extraction.php. [Accessed 6 October 2012].

[11] E. G. Jones, "ROS Wiki: arm_navigation," [Online]. Available: http://www.ros.org/wiki/arm_navigation. [Accessed 8 December 2012].

[12] M. Ciocarlie and K. Hsiao, "ROS Wiki: object_manipulator," 14 January 2011. [Online]. Available: http://www.ros.org/wiki/object_manipulator. [Accessed 11 October 2012].

[13] T. Foote, E. Marder-Eppstein and W. Meeussen, "ROS Wiki: tf," 26 February 2012. [Online]. Available: http://www.ros.org/wiki/tf?distro=fuerte. [Accessed 5 March 2013].

[14] D. Kent, "youbot_overhead_localization," [Online]. Available: http://www.ros.org/wiki/youbot_overhead_localization. [Accessed 6 3 2013].

[15] F. Clinckemaillie, "youbot_overhead_vision," [Online]. Available: http://www.ros.org/wiki/youbot_overhead_vision. [Accessed 6 3 2013].

[16] P. Malmsten, "Object Discovery with a Microsoft Kinect," 15 December 2012. [Online]. Available: http://www.wpi.edu/Pubs/E-project/Available/E-project-121512-232610/. [Accessed 5 March 2013].

[17] S. Chitta, "ROS Wiki: arm_kinematics_constraint_aware," 23 May 2011. [Online]. Available: http://www.ros.org/wiki/arm_kinematics_constraint_aware?distro=fuerte. [Accessed 8 March 2013].

[18] I. Sucan and S. Chitta, "ROS Wiki: move_arm," 13 October 2010. [Online]. Available: http://www.ros.org/wiki/move_arm?distro=fuerte. [Accessed 8 March 2013].

[19] M. Ciocarlie and K. Hsiao, "ROS Wiki: object_manipulator," 1 December 2012. [Online]. Available: http://www.ros.org/wiki/object_manipulator?distro=fuerte. [Accessed 5 March 2013].

[20] M. Ciocarlie, "GraspIt!," 2012. [Online]. Available: http://www.cs.columbia.edu/~cmatei/graspit/. [Accessed 5 March 2013].

[21] K. Hsiao, "ROS Wiki: interpolated_ik_motion_planner," 13 March 2010. [Online]. Available: http://ros.org/wiki/interpolated_ik_motion_planner. [Accessed 6 March 2013].

[22] M. Ciocarlie, "object_manipulation_msgs Msg/Srv Documentation," 22 February 2013. [Online]. Available: http://ros.org/doc/fuerte/api/object_manipulation_msgs/html/index-msg.html. [Accessed 6 March 2013].

[23] Microsoft, "Kinect Sensor," 2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/hh438998.aspx. [Accessed 7 March 2013].

[24] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM,* vol. 24, no. 6, pp. 381-385, 1981.

[25] M. Ciocarlie, "The household_objects SQL database," 21 July 2010. [Online]. Available:

http://www.ros.org/wiki/household%20objects. [Accessed 5 March 2012].

[26] K. Hsiao, S. Chitta, M. Ciocarlie and G. E. Jones, "Contact-Reactive Grasping of Objects with Partial Shape Information," October 2010. [Online]. Available: http://www.willowgarage.com/sites/default/files/contactreactive.pdf. [Accessed 11 October 2012].