End-to-End Programming Tools for Mobile Manipulator Robots

Justin Huang

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2018

Reading Committee:

Maya Cakmak, Chair

Dieter Fox

Siddhartha Srinivasa

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

End-to-End Programming Tools for
Mobile Manipulator Robots

Justin Huang

Chair of the Supervisory Committee:
Maya Cakmak
Computer Science & Engineering

Mobile manipulator robots have the potential to help people in a variety of unstructured scenarios, such as in households or in the service industry. However, with so many possible scenarios, roboticists cannot pre-program every task the robot needs to do. Instead, we need tools that make robot programming simpler, faster, and accessible to a wider audience of programmers. To this end, this dissertation presents research on two main approaches to robot programming for non-expert users: direct programming and programming by imitation. These approaches, and the technologies that support them, are situated in a conceptual framework comprising three key components: 1) perception, 2) motion specification, and 3) task scripting. In the realm of perception, we present a user-friendly system for specifying and locating task-relevant landmarks and a novel, state-of-the-art shelf detection algorithm. In the realm of motion specification, we enhanced existing programming by demonstration systems and created a new system for programming robots by visual imitation of a human demonstrator. Lastly, we developed a task scripting system that was deployed on a commercial robot. Throughout our work, we used system-level experiments, user studies, and case studies to show that non-roboticists could quickly learn to use our systems and program useful robot tasks. We conclude by describing possible extensions to our framework and envisioning directions for future work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to Dr. Maya Cakmak for five years of mentorship and guidance. She has made my graduate school journey rewarding and fun. I would also like to thank my committee members, Dieter Fox, Sidd Srinivasa, and Andrew J. Ko, for their helpful teaching, research discussions, and general support. Michael Ernst and James Fogarty also provided me with valuable advice. Oren Etzioni, Stephen Soderland (1949-2017), and Charles Elkan all helped to build my foundations as a researcher. My colleagues Sarah Elliott and Michael Jae-Yoon Chung provided years of friendship and camaraderie. Liz Chaddock's companionship and patience kept me grounded throughout the grad school process. Finally, I would like to thank my parents, Stanley and Julie Huang. Without their years of hard work and perseverance, this thesis would not be possible.

Chapter 1

INTRODUCTION

Mobile manipulator robots have the potential to improve people's lives, such as by entering dangerous situations in disaster relief scenarios [44, 73], assisting individuals with disabilities [33], or automating tedious tasks [39]. Robots like the Carnegie Mellon/University of Washington HERB [120], the Stanford PR1, or the Willow Garage PR2 were designed to perform a variety of indoor tasks such as doing the dishes, helping retrieve items for the elderly, or stocking grocery shelves [143]. As a proof of concept, the Stanford researchers released videos of human teleoperators controlling the PR1 robot to clean a room, retrieve items, and unload a dishwasher [145, 144, 146]. Later, Willow Garage demonstrated that the PR2 robot could perform similar tasks autonomously [136, 137].

Performing such tasks required the knowledge of expert roboticists with knowledge of computer vision, motion planning, autonomous navigation, and robot software systems. For example, to program the PR2 to retrieve a drink from a refrigerator, developers created special software modules for recognizing refrigerator door handles and drinks within the refrigerator, while others worked on robot arm motions for opening the refrigerator door, grasping the drink, and handing it off to a person [17]. Their work was integrated using Robot Operating System (ROS) software library [107]. More recent robot applications, such as in 2015's Amazon Picking Challenge, follow the same development process [39].

There are two important issues with the development process described above. The first is that the developers created *application-specific software* that cannot be applied to new applications that people might want. For example, the software module for detecting a refrigerator door handle cannot be used for sightly different tasks like detecting a drawer handle. Instead, we need general-purpose programming tools that can be applied to many different robot applications.

The second challenge is that robot application development *requires developers to be expert roboticists*. For example, it is unlikely that non-roboticist programmers could successfully program the PR2 robot to retrieve a drink—not without years of training to become expert roboticists themselves. As an example of how long it can take, one course teaching senior computer science undergraduate students to program a mobile manipulator with ROS required over 17 hours of lab time spread across 10 weeks [26, 27]. This means there is a long ramp-up time for new robot developers to become productive. Ramp-up time is a contributing factor to Brooks's law, "adding human resources to a late software project makes it later." [22] One of Brooks's observations is that training new developers in jobs that require expertise (*e.g.*, traditional robot programming systems) uses up expert programmer's time. Having a long ramp-up time also represents a lost opportunity, considering the fact that there are relatively few robotics programmers compared to the general population of programmers. If robot programming were simpler, faster, and more accessible to non-roboticist programmers, a greater number of them could create useful robot applications and explore new uses for robots.

As a result, our research goal is twofold: we want to simplify the robot programming process so that new developers can start making useful programs quickly (*e.g.*, in a matter of hours). At the same time, the programming tools we provide must not be overly simplified to the point where they cannot be used to develop useful robot programs. The next section provides a high-level overview of our work towards achieving this goal.

1.1 Our approach

In our research, we investigated how we could design and implement robot programming systems for non-expert users, as well as the technologies that support these systems. These tools were designed to provide users with *end-to-end* programming capabilities. That is, they are sufficient to program an autonomous execution of a mobile manipulation task without additional programming support from an expert roboticist. We investigated two main approaches: *manual programming* and *programming by imitation*. Both approaches, and our research projects in general, have similar world modeling approaches and a shared system architecture. This makes it possible to tightly

integrate all of our work and makes it simple to extend our work with improvements in the future.

1.1.1 Manual programming

In the *manual programming* approach, we designed a high-level API and a simplified interface with which non-expert programmers could develop mobile manipulation tasks. For tasks that were difficult to visualize in a scripting interface, such as creating perception primitives or moving the robot's arm, we integrated the scripting system with alternative interfaces that offer more suitable visualizations. The systems we developed for the purpose of manual programming include the *CustomLandmarks* perception system, a shelf segmentation algorithm, the *RapidPbD* motion specification interface, and the *CustomPrograms* scripting interface. We unified these components in a single, integrated system called *Code3*, and we successfully demonstrated that users new to the system could use *Code3* to program mobile manipulation tasks.

1.1.2 Programming by imitation

A different approach that we also investigated is *programming by imitation*, which we implemented in a system called *PBJ*. In this approach, users record a video demonstration of themselves performing a tabletop manipulation task. The user extracts motion information about the objects and the demonstrator from the video with the help of a semi-automated annotation tool. The robot uses this information to generate a program to replicate the manipulation task in a similar setting.

The advantage of this approach is that it puts less burden on the end-user of the system. To program a task, the user needs only to demonstrate the task in front of an RGBD camera. The same user can also perform the video annotation, although this step could be done by a different, more experienced user. In contrast, in the *manual programming* approach, the user needs to manually specify perception and motion primitives using the tools we developed. Another interesting feature of this work is that it could eventually lead to fully autonomous imitation by visual observation of a demonstrated task. In our research, we discuss the barriers towards making such a system a reality, and we illustrate existing tasks that users can program with the system.

1.2 Common research themes

To implement and study end-to-end programming tools for mobile manipulation, our work necessarily spans a wide range of topics, including perception, manipulation planning, and interface design. However, much of our work shares similar themes, world modeling approaches, and system architecture. Below, we describe the common themes that unify our work.

1.2.1 Conceptual framework

Our work fits within a common conceptual framework, which is discussed in further detail in Chapter 3. We follow the contours of Tomás Lozano-Pérez's *Robot Programming* [83], which discusses several key features of robot programming systems. In particular, we focus on what we view as absolutely necessary features for robot programming: perception, motion specification, and task-level scripting.

We model objects and parts of the environment as rigid bodies using 6D poses, although, in some cases, we make special accomodations for radially symmetric objects (*e.g.*, bowls, cans). Our perception-related work aims to produce these 6D poses, which we call *landmarks*, as output. Our motion specification tools utilize these landmarks so that users can specify end-effector poses relative to a landmark. Finally, we developed a scripting interface for programming task-level logic.

In Section 11.1.3, we also present prototype work related to another important aspect of robot programming: concurrency. We also discuss how our existing work can be adapted to work within a general concurrency framework.

As an additional layer of programming, users can configure robots to execute programs in reaction to certain events or conditions detected by its sensors. For example, a program created using the tools described above could be started at a certain time of day or when a face is detected. This approach is known as *trigger-action programming* (TAP). An example of a TAP interface is the commercially available website, *If this, then that* [68].

1.2.2 Ease of access

It is also important for both novice and experienced programmers alike that their programming tools be easily accessible and quick to program with. As part of our work, we developed web-based interfaces that programmers could use without special equipment or software, as well as video demonstration systems that only require an RGBD camera to use. In this dissertation, we present case studies showing how users took advantage of this ease of access, such as in the prototyping and deployment of programs for a commercial service robot.

1.2.3 User ecosystem

Throughout our work, we try to consider the fact that novice robot programmers do not exist in a vacuum. In a real-world environment, developers lie on a spectrum, with some experts who can engineer new features and use complex tools, some novices who want to do very little programming, and some in between [85, 105]. As a simple example, we added the ability to make copies of programs from a web-based scripting interface. In a programming workshop for high school students, expert developers created a template program, which students then copied and modified (see Section 7.6).

1.3 Robot platforms

This section gives a brief overview of robot platforms used in our research. Pictures of each robot are shown in Figure 1.1.

PR2 The Willow Garage PR2 is a dual-arm mobile manipulator. It has an omnidirectional base with a laser scanner for navigation. Its torso actuates up and down, which allows the robot's height to vary from 48 to 64 inches. The robot has numerous camera systems, including a camera embedded in each forearm, multiple stereo pairs in its head, a tilting laser on its chest, and a high-resolution camera in its head. However, in practice, we mainly made use of the Microsoft Kinect sensor mounted on top of its head. The arms each have 7 degrees of freedom with parallel jaw

Figure 1.1: Robot platforms used in our research. From left to right, the Willow Garage PR2, the Fetch Robotics Fetch, the Savioke Relay, and the Turtlebot 2.

grippers as end-effectors.

Fetch The Fetch, made by Fetch Robotics, is a single-arm mobile manipulator. Like the PR2, it has an actuated torso that allows its height to vary from 43 to 60 inches. Its arm has 7 degrees of freedom with a parallel jaw gripper as the end-effector. It also has a pan-tilt head with an RGBD sensor attached. For navigation, the Fetch has a differential drive base with an attached laser scanner. Fetch Robotics described the design of the robot in detail in [138].

Savioke Relay The Savioke Relay robot is approximately 3 feet tall and weighs 100 pounds. It can autonomously navigate indoor environments using a LIDAR, RGBD camera, and several sonar sensors. The front of the robot features a 7-inch touchscreen display that can be used to show information and receive user input. The robot has a bin with 0.75 cubic feet of storage, covered by a lid that locks when closed. It can connect to the internet via WiFi to call elevators and phones. The robot also has a docking station where it can charge its battery.

Turtlebot 2 The Turtlebot 2 is 16 inches tall. It has a Microsoft Kinect sensor in addition to internal odometry sensors and bumpers. Designed for robotics education and research, it has a variety of mount points that allow people to customize its design.

1.4 Thesis overview

1.4.1 Thesis statement

Programming tools that simplify the robot programming process, especially in terms of perception, motion specification, and task scripting, make it possible for non-roboticist programmers to quickly program mobile manipulation tasks.

1.4.2 Summary of contributions

The contributions of our research are:

- A system that simplifies the process of specifying perceptual landmarks for a robot task, and experimental results showing that novices to the system could utilize it to create detectors for unconventional landmarks in a short amount of time (Chapter 4).

- An algorithm for detecting shelf surfaces with better precision and recall than an existing algorithm deployed in a widely-used perception library (Chapter 5).

- Experimental results demonstrating the utility of a web-based task scripting system for service robots (Sections 7.3, 7.4.4) and for education (Section 7.6).

- Experimental results that characterize the challenges faced by end-users when using a trigger-action programming interface, and suggestions for improving future TAP systems (Chapter 8).

- Experimental results showing that simplified interfaces for perception, motion specification, and task scripting can be used by non-roboticists to quickly program robots to perform manipulation tasks (Section 9.4).

- A dataset and model for segmenting hands from RGBD images (Section 10.2).

- A programming by imitation algorithm that transforms an annotated video demonstration of a manipulation task into robot motions that accomplish the same task (Section 10.4).

1.4.3 Document outline

Chapter 2: Related work provides an overview of related work in robot programming, perception, motion specification, and task scripting.

Chapter 3: Conceptual framework provides an introduction to recurring concepts used in our work, and how they are integrated together.

Chapter 4: User-specified perception describes *CustomLandmarks*, a perception system designed to let non-expert users create their own perceptual detectors.

Chapter 5: Shelf perception discusses a system for segmenting the surfaces of a shelf scene, a common prerequisite for many manipulation tasks.

In **Chapter 7: Task scripting**, we describe the task scripting system we developed that integrates the other components of our system.

Chapter 8: Trigger-action programming investigates the use of an expressive trigger-action programming system, which could be used by end-users as a task scripting system for robotics. In **Chapter 9: End-to-end mobile manipulation programming**, we describe *Code3*, an integrated system for mobile manipulation programming that combines the perception, motion specification, and task scripting components from earlier chapters.

Chapter 10: Programming from RGBD video data shows how we can program robot manipulation tasks using RGBD video recordings of a person demonstrating the task. Although this approach differs from *Code3* in many ways, it also borrows many of the same concepts used throughout our research.

Finally, **Chapter 11: Conclusion** summarizes our results and proposes some extensions and other areas for future work.

Chapter 2

RELATED WORK

In this chapter, we describe prior work in the area of robot programming, and how they relate our research.

2.1 End-user programming

Researchers are actively studying end-user programming (EUP) [78, 94], with the goal of enabling ordinary people with no software development experience to write programs. Our research is generated targeted at non-roboticist programmers, not end-users. However, many of the methods and design guidelines [13, 72, 109] for EUP could apply to creating a rapid programming system.

One EUP technique is to represent the flow of the program or of the input data as a flowchart diagram [64]. Some commercially used examples of this include LabView and Simulink for engineering analysis, RapidMiner for data analysis, and Salesforce Process Builder for corporate processes. Flowchart-like languages have also been applied to programming robots [6, 15]. In our work, we use implemented task scripting systems without using flowchart metaphors because we wanted the greater expressivity of a general-purpose language. For example, it would be difficult to store high-dimensional state in a typical flowchart, which grows exponentially with the size of the state space.

Other forms of end-user programming include trigger-action programming, which trades off expressivity for high ease of use [68, 129] and domain-specific languages, which are specifically designed languages for a limited set of tasks [53, 90].

2.1.1 Visual programming interfaces

Visual programming interfaces provide a graphical editor where users can write code by connecting visual blocks representing snippets of code. This removes the need for beginner programmers to remember details about the programming language's syntax. Examples of visual programming interfaces include Scratch [86] and Alice [69], which are used to make animations and interactive applications. Visual programming interfaces have also been used to program robots in educational settings, such as with Lego Mindstorms [36, 75] or Wonder Workshop's Dash and Dot robots [139]. In our research, we developed a visual programming interface for robots, called *CustomPrograms*. Compared to educational programming interfaces, *CustomPrograms* includes more advanced programming language features like functions and list manipulation. To create the visual editor, we built *CustomPrograms* using the open-source Blockly project [54]. Blockly has been used in several educational applications, such as App Inventor, Hour of Code, and Made with Code[1].

2.1.2 Primitives and robot architecture

In our research, we chose certain "primitives" that make up the programming interface for the robot. In previous work, researchers have attempted to organize robot capabilities into primitives, such as in CARMEN [92]. However, the open-source robotics community has since moved towards other robot-agnostic middleware such as Player/Stage [38] and ROS [107].

2.1.3 Trigger-action programming

Trigger-action programming is a simple method for end-user programming, deployed in commercial products such as *If this, then that*. Previous work has established the importance of TAP in the context of smart homes, by showing that it can express most behaviors desired by potential users. In the work done by Ur et al. [129], users were asked to list smart home behaviors they wanted, and the authors found that all of the behaviors which required programming could be expressed in

[1] `https://developers.google.com/blockly/about/showcase`

a trigger-action format. Similarly, Dey et al. [46] asked users to think of open-ended behaviors for smart homes, and showed that close to 80% of the described behaviors fit an if-then format.

Researchers have also developed and evaluated different TAP interfaces. Ur et al. developed an IFTTT-like interface that supported multiple triggers and multiple actions, and found that users could correctly create a given set of rules about 80% of the time [129]. Dey et. al. built a visual programming system (iCAP) in which triggers and actions could be dragged onto a rule sheet [46]. Their user study evaluating this interface showed that non-programmers were able to program rules to implement a given set of behaviors. Häkkilä et al. implemented a TAP system (Context Studio) for customizing a mobile phone [59]. Zhang et al. [152] presented work on visually debugging and exploring event-condition-action rules for robot behaviors. Welbourne et al. [133] describe the design of a system for designing and verifying location triggers by modeling them as finite state machines. Truong et al. [128] provide a different programming model for smart homes, through the metaphor of magnetic poetry. In this model, users arrange tokens to form sentences describing desired behaviors.

In our work, we classify different types and triggers and actions and discuss how these properties affect the correctness of the rule. Our work contributes a theoretical description of these distinct types as well as empirical findings demonstrating ambiguities due to these distinctions from two user studies in which trigger and action types are systematically varied.

2.2 Programming by demonstration

Programming by demonstration (PbD) is a commonly-used technique for programming robots by guiding the robot through an example of a manipulation action [9, 14, 34, 83]. Researchers have studied many different aspects of PbD, including how to represent actions [1, 31, 98, 122], learn high-level task structures [50, 91, 103], use PbD to train policies in a reinforcement learning framework [10, 118], and resolve design issues when interacting with human teachers [28, 70, 125]. User studies have shown that end-users can learn and use PbD systems to program various manipulation actions [1, 4]. Compared to previous research on PbD, our research studies a more tightly integrated system that combines PbD with flexible perception capabilities and task scripting

systems for logic and control flow.

2.3 Integrated robot programming tools

The primary goal of our research is to let a broader group of people (those with general programming experience) program robots. Other systems have been developed with similar goals.

Two related systems include ROS Commander [96] and RoboFlow [6], which integrate robot actions, including those programmed by demonstration, with a visual programming language using a data flow model [64]. Interaction Composer [55, 56] is a system that also combines robot actions (coded in C++) with a flow-based interface. The authors of [6, 56] point out that, in some cases, flow-based interfaces do not scale well, especially when the state space is large. In contrast, our system uses a general-purpose language for developing control flow, which roboticists and programming language experts interviewed in [6] said they would prefer to use over a data flow model.

Others have designed robot programming interfaces for creating social interactions. These include the TiViPE [81, 82], Choregraphe [106], and Interaction Blocks [115] interfaces for programming the Nao robot. They combine flow-based programming interfaces with support for timing, social dialogue, and libraries of gestures. RoboStudio [43] is a system for authoring UI and control flow for healthcare robots. In our work, we do not focus on developing human interaction experiences. Instead, we focus on manipulation tasks such as fetching and carrying, delivering objects, and manipulating the environment.

2.4 User-friendly perception

Perception is a key component of robot programming. Our research focuses on systems that are easy for users to customize or refine for their own tasks. Specifically, *CustomLandmarks* (Chapter 4) lets users create detectors for custom 3D shapes in RGBD scenes.

2.4.1 Object detection and pose estimation

Although the landmarks represented by *CustomLandmarks* are not limited to objects, object detection is an important and closely related area of research, with a vast body of literature behind it. Most techniques compute alignments between two point clouds by repeatedly matching 3D features sampled from the point clouds or models using different features or speed optimizations [23, 48, 113]. Our approach differs in that our landmark representation also specifies areas of empty space around the point cloud. This allows users to make distinctive landmarks out of otherwise generic point cloud segments. Many systems require full 3D object models as input, either from CAD models or from a scan of the object [37, 102]. This can improve detection performance but make the system harder to use and deploy. Because usability is one of the primary goals of our research, we only record object models from a single viewpoint.

Localizing landmarks can also be viewed as an object recognition or representation learning problem, which has seen major progress in recent 2D computer vision research [8]. However, these systems recognize only a predefined set of objects and require large amounts of training data to work well, which non-experts might not have the time or ability to assemble. Additional work has gone into learning representations of 3D shapes using deep neural networks [89, 141]. These representations could be used to compare sampled volumes from the scene to a landmark. However, the current resolution of these voxel grid representations may be too limited to represent most landmarks. Another approach is to use these representations to find correspondences between the landmark and the scene as part of a larger detection framework [124, 151].

2.4.2 Vision for manipulation

One of the most common tasks for robot manipulation is grasping. To this end, researchers have developed different techniques for inferring grasp points using vision. Many approaches use probabilistic models [65, 117], including neural networks [58, 76, 108]. All of these approaches are specifically designed for the purpose of grasping objects and cannot be applied to other perception tasks such as locating parts of the workspace.

2.4.3 User-guided vision

Human input can be helpful supplements to automated perception systems. In our research, we use human input to create perceptual landmarks and to supervise the annotation of task demonstration videos. Other researchers have looked at how robots can learn about object parts from human demonstrations. For example, Hsaio et al. explored how robots can learn how to grasp objects by comparing them to objects grasped by a teleoperator in simulation [66]. Herzog et al. extended this approach by inferring grasp templates from demonstrations [63]. Our work differs in that we study how user-defined landmarks can be used to detect non-object parts, such as the corner of a work surface. Human input can also help guide manipulation planning. For example, Dellin et al. showed how teleoperators could use virtual fixtures to introduce helpful constraints for motion planning algorithms during the DARPA Grand Challenge [44].

Outside of robotic manipulation, researchers have investigated using human input to aid with vision tasks. In particular, a common vision task that makes use of user input is interactive image segmentation, in which the foreground of a 2D image is separated from the background [16, 150]. A similar interactive segmentation technique was applied to robot manipulation settings by Butler et al. [24] This approach could be used to define landmarks from a 2D image of the scene. However, landmarks representing parts of the workspace may frequently appear to be the in background of an image, so most techniques for background subtraction are not likely to work.

2.4.4 Landmarks for programming by demonstration

Previous research on programming by demonstration has not focused on integrations with new perception capabilities. Instead, researchers have used different scene understanding techniques like fiducials [97, 111], marker-based motion capture [130], or simulated environments [98]. Others have used limited or special-purpose perceptual systems such as object detectors on flat, uncluttered tabletop surfaces [2, 4]. Our work improves on the perception capabilities of such PbD systems by detecting landmarks in arbitrary scenes.

2.5 Task tracking and perception

In some cases, users can program a robot task by having the robot watch a human demonstrator perform the task. Our *PBJ* system, described in Section 10.4, is one such system. The process of extracting relevant information from visual observation of a demonstration is a challenging subproblem by itself. In past work, researchers have utilized a variety of sensing modalities. In *lead-through programming*, the demonstrator moves a robot's arm through the steps of a task, and the arm's pose is recorded using internal sensors [83]. Most lead-through programming systems will additionally perceive the poses of objects in the scene. Some systems can find initial object poses using segmentation algorithms [5]. However, most systems track object poses over time, usually by attaching special markers to objects [98, 134]. The pose of the demonstrator can also be tracked by attaching markers to the person (e.g., [30, 130]) or by equipping the demonstrator with wearable sensors like data gloves (e.g., [51, 87]). Researchers have also collected demonstrations in virtual reality environments [3, 153], which are harder to set up but provide perfect information.

Our system does not require the use of markers for motion capture, leveraging recent advances in computer vision research. An important component of the system is to track the demonstrator's hands, where researchers have recently made progress on regressing from RGB images to hand joint positions [32, 101, 154]. We segment hands from images using a model derived from Xiang et al. [147], which gives us a hand / not hand label for each pixel. We track the pose of the demonstrator from RGBD images using an articulated skeleton tracker [131], but other recent approaches for human pose estimation [32, 154] could also be used.

2.6 Task representation and learning

Programming by demonstration approaches can be broadly divided into those that learn short-term arm trajectories (e.g., pouring a cup or opening a cupboard) and those that learn higher-level actions like changing a tire or setting a table using an existing catalog of low-level skills. In the first category, researchers have used statistical methods like Gaussian Mixture Models [31] or Hidden Markov Models [29] to find commonalities between multiple demonstrations of the same

task. Dynamic Movement Primitives (DMPs) represent trajectories as an attractor landscape that can be parameterized by a goal configuration [104]. Recently, researchers have also developed deep reinforcement learning algorithms that, given video demonstrations of a task, learn a policy that directly maps image observations to low-level robot controls [79, 119, 153].

A key challenge with the approaches described above is that they require multiple demonstrations or hundreds of videos for the system to acquire the skill. For novice users, it may not be intuitive or convenient to provide multiple demonstrations. As a result, we designed the *PBJ* system to be a *one-shot* system that imitates an action from a single demonstration.

2.6.1 Task planning and execution

Given a goal specification for a motion or a task to perform, the robot must plan its grasps and its arm trajectories while obeying task constraints. An early example of an integrated task planning system was Handey, which planned pick-and-place motions for polyhedral objects [84]. Pick-and-place continues to be an active area of research, with recent research on how to pick items from shelves [62] and how to grasp items in cluttered environments by rearranging obstacles [47, 121]. In some cases, robots perform higher-level task planning for longer actions that require multiple steps or that have hierarchical structure [91, 123]. In our work with the *PBJ* system, we focus on planning grasps such that they remain feasible during the action. During the planning process, we relax certain constraints for radially symmetric objects, which helps with the *overspecification* problem described in [83].

Chapter 3
CONCEPTUAL FRAMEWORK

In our research, we investigated several approaches to robot programming for non-experts. As described in the introduction, we focused our work on three key areas for mobile manipulation programming: perception, motion specification, and task scripting. In this chapter, we describe how we connect these different areas in a common framework. The framework defines common interfaces between perception, motion specification, and task scripting components, which makes it easy to integrate them all into a single, end-to-end programming system. The components are modular as long as they follow the interface laid out in this chapter. For example, we can have two different kinds of perception algorithms working simultaneously, and they could feed into different kinds of motion specification tools, which makes the overall system richer and more powerful.

Below, we describe in detail common concepts and definitions we use for perception, motion specification, and task scripting, including how they interface with each other. Figure 3.1 provides a graphical summary of the interfaces defined by our framework.

3.1 Perception and world modeling

To program manipulation tasks, we need a model of the robot's workspace, such as the poses of the robot or of objects. A common world modeling approach is to represent poses as homogeneous transforms (also known as a *6D pose*), which encode position and orientation, relative to a reference frame [83]. Transforms can be related through a transform graph. For example, a grasp pose may be defined relative to the pose of an object, and the object's pose may be defined relative to a fixed frame. Based on this, we can compute the pose of the grasp pose relative to the fixed frame [41]. In the context of manipulation tasks, we often refer to the poses of objects or other parts of the scene as "landmarks". We largely utilize this approach, with one exception for radially symmetric

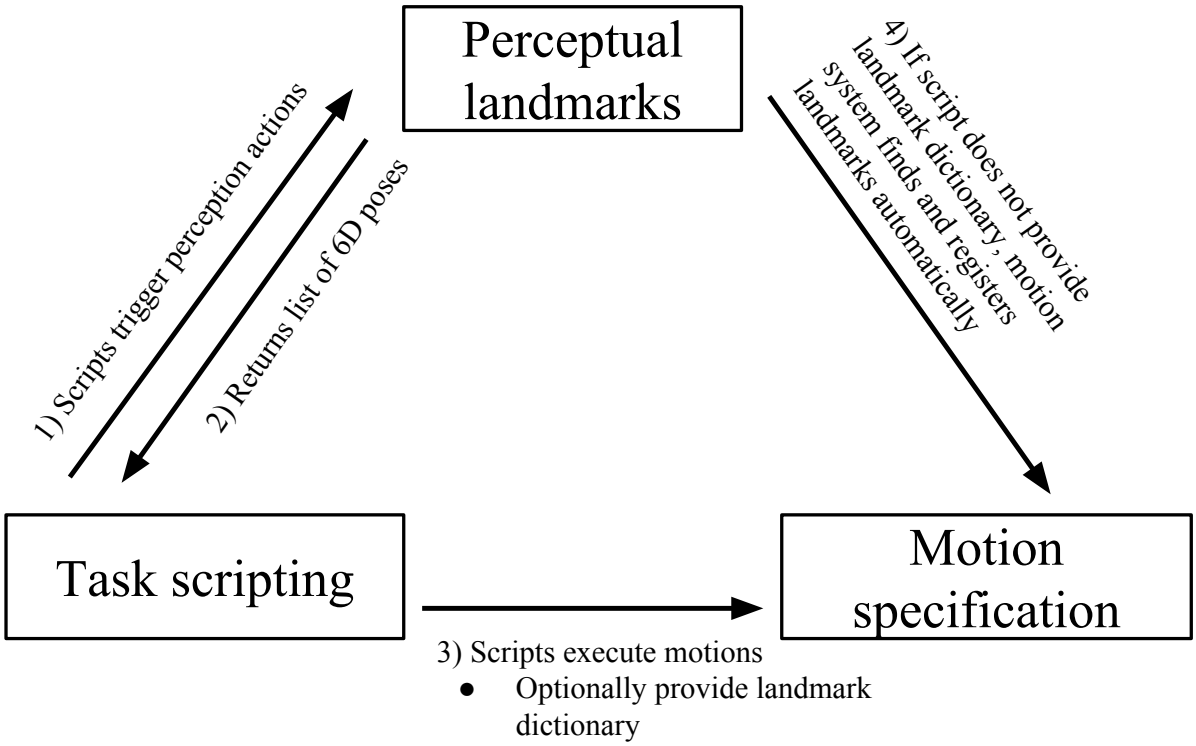


Figure 3.1: An overview of the conceptual framework underpinning our research. The figure provides a rough ordering of events for a typical program execution. 1) The developer calls the perception system to 2) get a list of landmarks. He or she may then write landmark registration code (see text) to select the relevant landmarks for a motion. 3) The script triggers the motion, optionally parameterized by the selected landmarks. 4) If the programmer does not write their own landmark registration code, the motion specification system can call the necessary perception systems and register the landmarks automatically.

objects, as we discuss below.

Below, we describe several kinds of entities that we may model for a robot manipulation task.

3.1.1 Tabletop / shelf segmentation

A simple technique used throughout our research is to model table and shelf scenes (*e.g.*, Chapter 5) by fitting oriented bounding boxes around the surface and the objects. The center of the bounding box is the origin of the object's local coordinate frame. The z-axis of the object's local coordinate frame generally points "up" (the opposite direction of the force of gravity). We also generally choose the local x-axis to point along the shorter axis of the bounding box, so that the object's orientation is consistent if the object is rotated. Given two choices of x-axis, we choose the direction that more closely points in the same direction as the x-axis of the fixed frame. The bounding box representation also includes the dimensions of the box along the local x, y, and z axes.

3.1.2 Custom landmarks

In the *CustomLandmarks* system (Chapter 4), users specify RGBD templates by drawing a bounding box around a segment of the sensor data. The local coordinate frame of one of these landmarks is attached to the center of the bounding box. The orientation of the bounding box is determined by the tool used to draw the box.

3.1.3 Object models

In some cases, we require users to provide 3D mesh models of the objects in the task. These models may come from CAD software or from 3D scans. The local coordinate frame is determined by the software used to create the model. In the *PBJ* system (Chapter 10, which requires object models to be available, the local coordinate frame of the object model was located near the bottom center of the object.

3.1.4 Radially symmetric object models

Objects that are radially symmetric about their local z-axis, such as cups and bowls, have an ambiguous orientation about the z-axis (*i.e.*, they have an ambiguous *yaw angle*). Object tracking and detection systems that rely only on depth and shape information will be unable to accurately estimate the object's yaw angle, so we must design systems that take this into account. For example, given a specific grasp pose relative to a radially symmetric object, we can sample other potential grasps by rotating the grasp pose about the object's local z-axis. This assumes that the exact orientation about the z-axis (the *yaw angle*) is unimportant, which is not always true. For example, when placing food cans on a shelf, it is important to face the branding outward. Handling such situations requires advanced perception systems that use color information to accurately estimate the object's yaw angle, but is out of scope for our research.

3.1.5 Fiducial markers

Specially designed fiducial markers, which resemble barcodes, can also be used as landmarks. These are designed to be easily detected by tracking systems. Their poses can be determined using open-source software such as the ALVAR Tracker [99]. Different barcode patterns can be used to uniquely identify different markers. We do not use fiducial markers in our research, but they are simple to use and are commonly used in prototyping applications. Because of their convenience for rapid prototyping, integrating fiducial markers into our work is an area for future work.

Because most landmarks are represented as a 6D pose, we can mix and match landmarks within a single system. For example, a robot could perform tabletop object detection, and it could also locate a fiducial marker on the table. It could then execute a motion that uses both kinds of landmarks, such as grasping an object and placing it on the fiducial.

3.2 Landmark registration

Landmark registration is the process of finding correspondences between landmarks encountered previously (*e.g.*, during a user demonstration) and the landmarks encountered when the robot tries

to perform the action in a new scene. For example, suppose the user demonstrates placing a can on top of a box. When the robot performs the action, it will detect two objects, and it must determine which is the can and which is the box. Additionally, if there are multiple cans or boxes, it must ultimately select one can or box to execute the action with.

In our framework, motion specification systems assign each landmark used in the action a unique name. We denote such names in the text with quote marks, as in "Object 1" or "Box." Below, we describe different strategies we use for landmark registration.

3.2.1 Score function

In this approach, each perception system uses a scoring function that evaluates the quality of a landmark match. For tabletop/shelf segmentation, objects are represented using oriented bounding boxes, and the best match is the box with the most similar dimensions, as measured by the Euclidean norm of the difference of their dimensions. *CustomLandmarks* (Chapter 4) has its own scoring function, described in detail in Section 4.1.3. If there are two instances of the same object, such as two identical cans on the table, this strategy can lead to an arbitrary choice. However, it often works well for discriminating between different kinds of objects.

3.2.2 User programming

In cases where there are multiple instances of the same landmark, another approach is to let the user discriminate between the landmarks in code, using the task scripting system. For example, in our *CustomPrograms* task scripting system, users can write programs that read the X, Y, and Z coordinates of the landmark in the fixed frame. Based on that, the program can choose a landmark based on its X/Y/Z position in the scene. For example, if the system detects two cans that are stacked, the user could write code to choose the uppermost can (the one with the largest Z value).

3.2.3 Similarity to past poses

Another approach is to compare the position of the landmark to where the landmarks were at at a past time, such as when a user first demonstrated a task. For example, suppose a user demonstrates stacking one box on the left side of the table ("Box 1") on top of an identical box on the right side of the table ("Box 2"). At runtime, the two boxes are visible again, but in slightly different starting positions, and the robot must decide which box is "Box 1" and which is "Box 2." Because the boxes are identical, the robot must make use of other distinguishing features to determine which box is which. Using the strategy of comparing to past poses, the robot would assign the label of "Box 1" to whichever box is closer to the left side of the table and similarly for "Box 2." In practice, we use this strategy in conjunction with the *score function* strategy, so that we do not create correspondences between objects that obviously do not match.

3.3 Motion specification and planning

In our framework, robot arm motions are represented using a sequence of end-effector (*i.e.*, the gripper) poses. In some cases, we also specify a trajectory for the end-effector to follow, which we represent as a sequence of closely-spaced end-effector poses.

End-effector poses can be specified relative to landmarks. A landmark can be any of the different kinds of landmarks we described above, or it could be a fixed frame or a frame attached to the robot itself. For example, to perform a wave, it makes sense to define a sequence of end-effector poses relative to the robot's torso link. For actions that consist of multiple end-effector poses or trajectories, different poses can be parameterized by different landmarks. For example, to pick up an object and place it at the edge of a table, both the object and the table can be landmarks. To pick up the object, we can specify a pre-grasp, grasp, and post-grasp pose relative to the object. To place the object back down, we specify poses to put the object down and release it relative to the edge of the table.

When moving multiple parts of the robot, we can give users the option to create concurrent movements. For example, at the start of a task, the user may want the robot to assume a "ready"

position that involves raising the torso, moving the head to look down at the table, and opening the gripper, all at the same time. We built this capability into our *RapidPbD* interface (Section 6.3). In *RapidPbD*, an action is broken down into a sequence of steps. Within each step, the user can command multiple resources like the head or the gripper to move together in parallel. However, each step will run sequentially, waiting for all the resources to finish moving before proceeding to the next step.

Motion planning generally refers to the problem of finding a collision-free path for the robot arm to follow to move the end-effector to the desired pose. It is an active area of research, and in our work, we treat motion planning algorithms as a black box. A simple approach we use is to first consider a straight-line path from the end-effector's start pose to its goal pose. If this fails to find a collision-free path (*e.g.*, because there is an obstacle in the way), we use alternative motion planning algorithms such as a bidirectional RRT search [74].

In some situations, users may want to impose constraints on the trajectories generated by motion planners. For example, while holding a cup of water, the robot must hold the cup upright throughout the trajectory. Although none of our systems give users the ability to specify such constraints, doing so is an area for future work.

3.4 Task scripting

Perceiving the environment and making arm motions are just the beginning of a robot program. A more complete robot program generally requires control flow constructs such as loops and conditionals. It may also involve collecting user input, playing sounds, or running different arm motions in sequence. For example, a bartender robot may need to ask the user to choose a kind of drink, which changes the perception and motions the robot needs to perform. It also needs to loop back to the beginning of the program when approached by a new user and fail gracefully when error conditions occur. Accomplishing this requires a task scripting system that lets developers write application-specific logic. A specific example of this is *CustomPrograms* (Section 7.1), a drag-and-drop interface for writing high-level code.

In our framework, we assume that arm motions and perceptual landmarks have a unique identi-

fier, such as a human-readable name. These perception actions and motions have an interface that allows them to be triggered by name. Perception actions return a list of landmarks. Developers can iterate through this list and select a landmark based on attributes like its position relative to the fixed frame. Motions that are parameterized by at least one landmark accept a *landmark dictionary*, a dictionary data structure that maps a landmark's name to its pose.

For example, suppose we are programming the robot to place an object at the top of a stack. This motion could have the name "Place object on stack," and it would have two landmarks: the object to grasp and the object at the top of the stack. These landmarks could be given the names "Bottom object" and "Top object," respectively. The developer first uses the perception system to find all instances of the landmark. The bottommost landmark is saved as the "Bottom object" and the topmost landmark is the "Top object." Finally, the developer triggers the "Place object on stack" motion, passing in the poses of the "Bottom object" and "Top object" as arguments, which causes the robot to adjust the motion according to the given locations of the two landmarks.

Trigger-action programming

So far, we have discussed task scripting systems that use general-purpose programming language constructs like loops, conditionals, and dictionary data structures. However, some users may want a simpler interface for programming the robot. A simpler possibility is *trigger-action programming* (TAP), in which the robot starts an action according to some condition, such as time of day, low battery, or detected face. The action could be a robot arm motion, but it could also be an entire program developed using another task scripting system like *CustomPrograms*. This model has proven useful in the context of smart-home programming, such as with the web-based application, *If This, Then That* (IFTTT) [68], but its use in robotics has not been investigated. An example of TAP being used in our framework is to have users schedule a program built with *CustomPrograms* to run at a certain time every day.

One challenge with existing interfaces like *If This, Then That* is that they may be too simplistic to develop useful programs with. In our research, we investigated the use of a more complex conditions in a trigger-action programming interface, and we studied common errors that non-

expert users might make (Chapter 8). Our work focuses on TAP abstractly, and we studied a hypothetical interface in the context of smart home programming, not robotics. Investigating ways to make trigger-action programming less error-prone, especially in the context of robotics, is an exciting area for future work.

3.5 Extensibiilty

Using the above framework allows us to tightly integrate multiple systems together. For example, within a single *CustomPrograms* program, a developer could first trigger a PbD action that uses tabletop objects as landmarks, then trigger a *PBJ* action that uses 3D model matching and fiducial markers as landmarks. If we develop new perception systems that generate 6D poses, those poses can be used to parameterize existing motions. Likewise, if we develop new motion specification systems that accept landmark dictionaries as input, it can utilize a wide variety of landmarks. Lastly, alternative task scripting systems can be used to trigger any motion specification system, as long as those motions have unique names. We have used this framework to deploy our research on three robots, and we have used the framework to prototype new types of landmarks for future work.

Chapter 4

USER-SPECIFIED PERCEPTION

For developers to program useful manipulation tasks, they must have the capability to detect and locate task-relevant landmarks, such as parts of objects or specific areas of the workspace. Programming robots to do manipulation tasks typically entails the development of custom, task-specific perception software. Adjusting such software to minor task modifications often require additional labor from experts in robotics or computer vision. While researchers have tried to lift that burden by developing general-purpose object detectors, these advancements are not designed to detect *parts* of objects or of the scene. Another approach is to attach visual markers that the robot can easily locate (*fiducials*) to parts of the scene. However, attaching markers to all landmarks is not always possible or desirable.

As a result, one our our key research goals was to enable non-expert users to specify arbitrary perceptual landmarks. This chapter introduces *CustomLandmarks*, a system that lets users define partial 3D models of landmarks from sensor data and locate them in new scenes. A key contribution of this system is its representation of a landmark, which captures not only its geometry, but also nearby space that is expected to be empty. This representation is flexible and can be leveraged in creative ways. For example, to visually check whether the robot has successfully grasped a thin, lightweight object, a user can create a landmark of the robot's gripper with a region of empty space beneath it and program the robot to search for this landmark. We also present an algorithm for locating these landmarks that behaves predictably and is simple for users to understand.

The landmark representation requires users to adopt a different way of thinking about object and workspace detection, and a natural question to ask is whether novice users are capable of using landmarks in this way. A second contribution of our work is to show, through a user evaluation of *CustomLandmarks*, that users are capable of using the system in non-obvious ways after minimal

training.

In the rest of this chapter, we describe the representation of a landmark and provide an algorithm for locating them in new scenes. We then characterizes the performance of the system on a variety of case studies. We also demonstrate how we used the system to perform a variety of manipulation tasks on a PR2 robot. Finally, we present the design and results of our user study.

4.1 System overview

CustomLandmarks consists of three key components: the landmark representation, a user interface for creating landmarks, and an algorithm for locating landmarks in new scenes. We describe each below.

4.1.1 Landmark representation

In our system, a landmark represents a partial 3D template of a rigid object or part of a workspace. The geometry of the landmark is represented with both a point cloud and a box that surrounds the point cloud. The boundaries of the box specify a region of empty space around the landmark that is expected be unoccupied. The box does not need to be centered on the object. This allows the landmark to encode different emptiness constraints such as "a can with empty space above it" or "a box with empty space to its right." Although the point cloud may be off-center, we do require that it is completely contained in the box. Each landmark is associated with a local reference frame that is located in the center of the box. In our implementation, the orientation of this frame is the same as the robot's base frame. However, this is an arbitrary choice and could easily be modified in the future. As explained in the next section, the point cloud data is obtained by capturing data from an RGBD camera. We assume that the data comes from a single viewpoint.

Figure 4.1 illustrates the landmark representation and use of empty space, and Section 4.2 explores the consequences of this representation in detail.

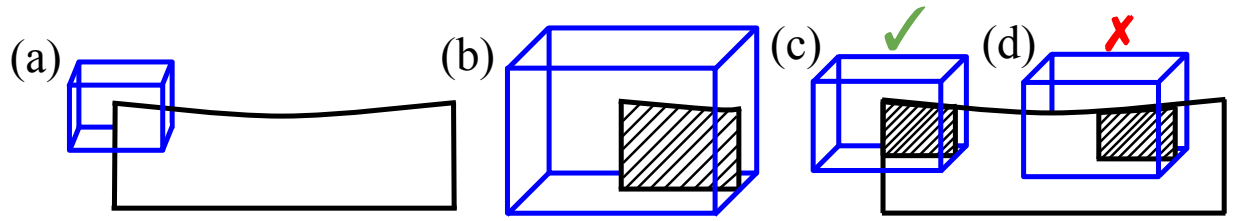


Figure 4.1: An illustrative example of custom landmark specification and search. (a) A landmark representing the top left corner of a flat vertical area of the robot's workspace (e.g., a plastic divider) is selected by drawing a box around the corner. The top and left halves of the box are empty space. (b) A close-up view of the landmark. The shaded region represents the captured point cloud. (c) The corner landmark matches well with the corner of another divider. (d) However, it will not match well when aligned with the top middle of the divider. Although all the points of the landmark (shaded region) match well with the divider, part of the scene intrudes upon the empty space of the landmark box.

4.1.2 User interface

The user interface for creating landmarks shows a point cloud view from the robot's depth sensor, as well as a 3D box-shaped selector, with controls to set its position and dimensions. A screenshot of the box selector is shown in Figure 4.7. With the landmark placed in the robot's view, the user moves and resizes the box to surround the landmark of interest, potentially including margins of empty space on the side. When the landmark is saved, the system records to a database the subset of the scene within the box, as well as the pose and dimensions of the box. For convenience, our implementation also saves the entirety of the scene, so that the landmark's box can be edited offline using the same scene. We chose to make the selector shaped like a box and aligned with the base of the robot for simplicity to the user and to the implementation. However, a more advanced interface could be used to define an arbitrary shape and orientation for the landmark. The landmark is captured from a single view from one or more point clouds, and does not include color information. In our implementation, we fused five separate point cloud readings from the same static scene to

reduce noise in the data.

4.1.3 Landmark search algorithm

To make use of custom landmarks, we need a way of localizing them in a new scene. We first formally describe the inputs and outputs. A point p is a location vector $p = (p_x, p_y, p_z)$ and a scene S is simply a set of points. A landmark ℓ is a tuple (P, B), where P is a set of points (*i.e.*, points in the point cloud selected by the user) and B represents a box, specified by a 6-dimensional pose and a 3-dimensional size vector.

Algorithm 1: FindLandmark

Input : *scene*, *landmark*, miscellaneous *parameters*

Output: a list of aligned landmarks in the scene, or empty list if not found

1 crop *scene*;
2 downsample *scene* using a voxel grid;
3 *samples* = sample points from *scene* using a voxel grid, leaf size of 1/3 of the landmark's dimensions ;
4 *candidates* = [];
5 **for** (*sample* **in** *samples*) {
6 *c* = copy of *landmark*;
7 move *c* such that *c.cloud* is centered on *sample*;
8 *c* = run ICP to align *c.cloud* with *scene*;
9 *c.error* = `CandidateError` (*c*, *scene*);
10 add *c* to *candidates*;
11 }
12 remove all *candidates* that do not have the lowest error within a certain radius (non-max suppression);
13 *output* = $\{c \in candidates \mid c.error < threshold\}$;
14 **return** *output*;

Algorithm 2: CandidateError

Input : *scene*, *candidate* landmark

Output: error score

```
1  error = 0;
2  denominator = 0;
3  croppedScene = scene cropped to candidate.box;
4  visited = [];
5  for (scenePt in croppedScene) {
6      candidatePt = nearest point in candidate to scenePt;
7      error += distance between scenePt and candidatePt;
8      denominator += 1;
9      add candidatePt to visited;
10 }
11 for (candidatePt in candidate.cloud) {
12     if (candidatePt not in visited) {
13         scenePt = nearest point in scene to candidatePt;
14         error += distance between scenePt and candidatePt;
15         denominator += 1;
16     }
17 }
18 return error / denominator;
```

Our search algorithm takes as input a scene S, represented by the complete point cloud captured before an execution, and a landmark ℓ to be localized in that scene. It outputs a set of landmarks O, where each landmark $\ell_o \in O$ is potentially an instance of the input landmark ℓ in the scene S.

Pseudocode for the algorithm is given in Algorithm 1. First, we crop and downsample the scene

according to application parameters.[1] Then, we sample scene points and initialize an instance of ℓ at each sampled point. The sampling is done by downsampling the scene again using a voxel grid, which ensures that the entire scene is systematically searched. The leaf size of this voxel grid is set to be a fraction of the dimensions of the landmark (we found 1/3 to be a good value). Next, we run the iterative closest point (ICP) algorithm [12] to align the landmark's point cloud P with S, which produces a modified landmark ℓ'. For each landmark ℓ', we compute an error metric (Algorithm 2) using ℓ' and S. We then perform non-max suppression so that we do not produce duplicates of the same result. Finally, we filter the results by thresholding on the error metric.

The error metric (Algorithm 2) can be thought of as summing two measurements of error. The first measurement is of how well the landmark's shape matches with the scene, while the second is of how much of the scene intrudes on the empty space in the landmark's box. Formally, the first measurement is the sum of the distances between points on the landmark and their nearest points in the scene (lines 11-17). The second measurement is the sum of the distances between points of the scene (within the landmark box) and their nearest points on the landmark (lines 5-10). Adding a margin of empty space around the landmark helps eliminate false positive matches. If scene points are found where there is expected to be empty space, then they will increase the mean error, since the nearest points on the landmark are far away. This process is illustrated in Figure 4.1(c) and 4.1(d).

4.2 Case studies

In this section, we show how we used *CustomLandmarks* for 15 separate tasks and quantify the performance of our landmark search algorithm. *CustomLandmarks* can be used to model a variety of non-object landmarks. Below, we describe five categories of such landmarks and provide examples of how they can be used. Figure 4.2 illustrates these examples.

---

[1] For our research, the scene was cropped to a volume in front of the robot roughly equal to the reach of its arms. The scene was downsampled to a leaf size of 1 centimeter. The non-max suppression radius was computed to be half of the longest dimension of the landmark. We varied the threshold for our error metric for experimental purposes but generally use a value of 0.75 centimeters.

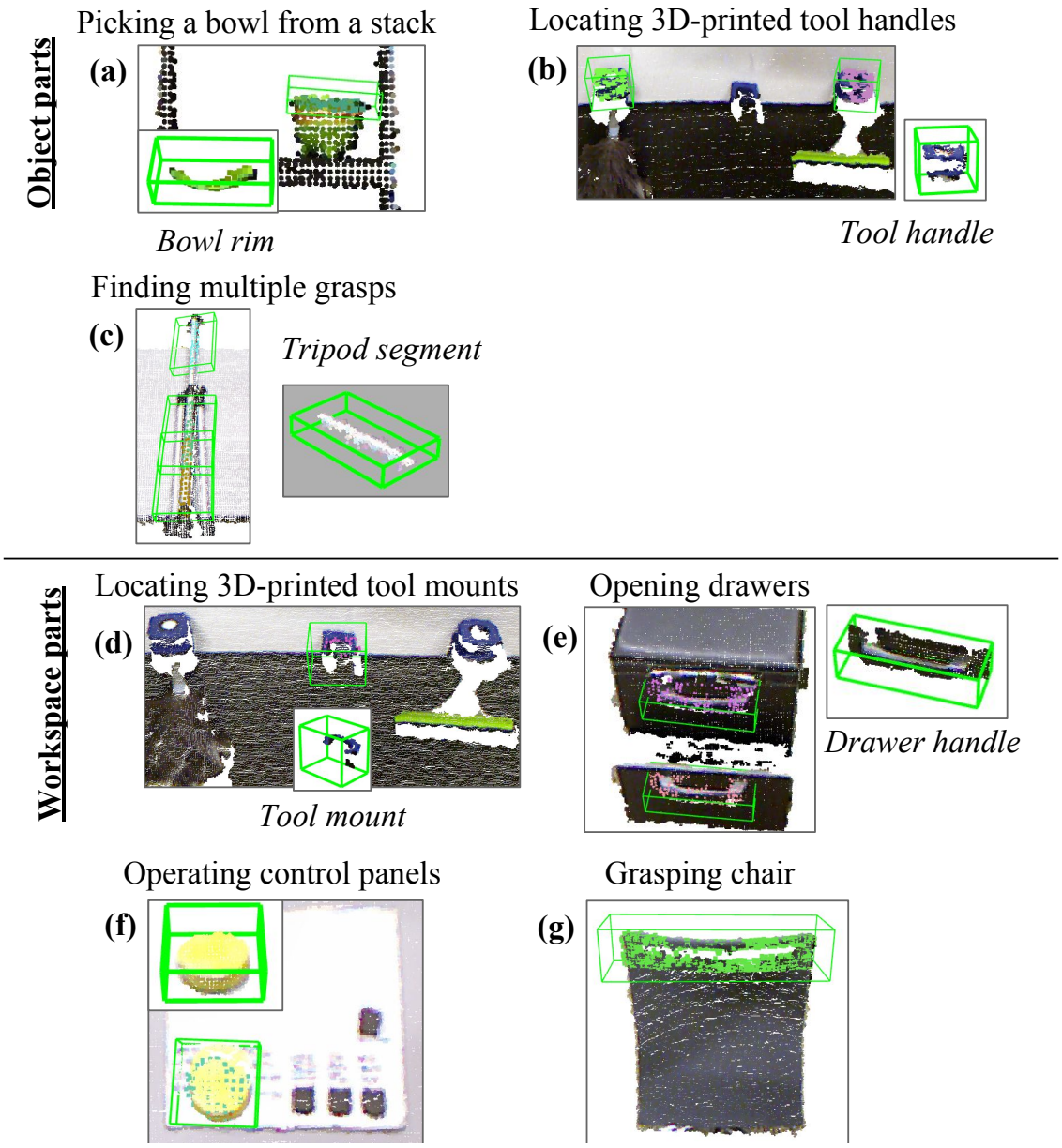


Figure 4.2a: Scenes and landmarks described in the case studies of *CustomLandmarks*. Two categories of landmarks are illustrated above: object parts and workspace parts. For each case study, we show one of the scenes we evaluated and the landmark or landmarks involved. The scenes also show examples of detections our system found, shown with green boxes.

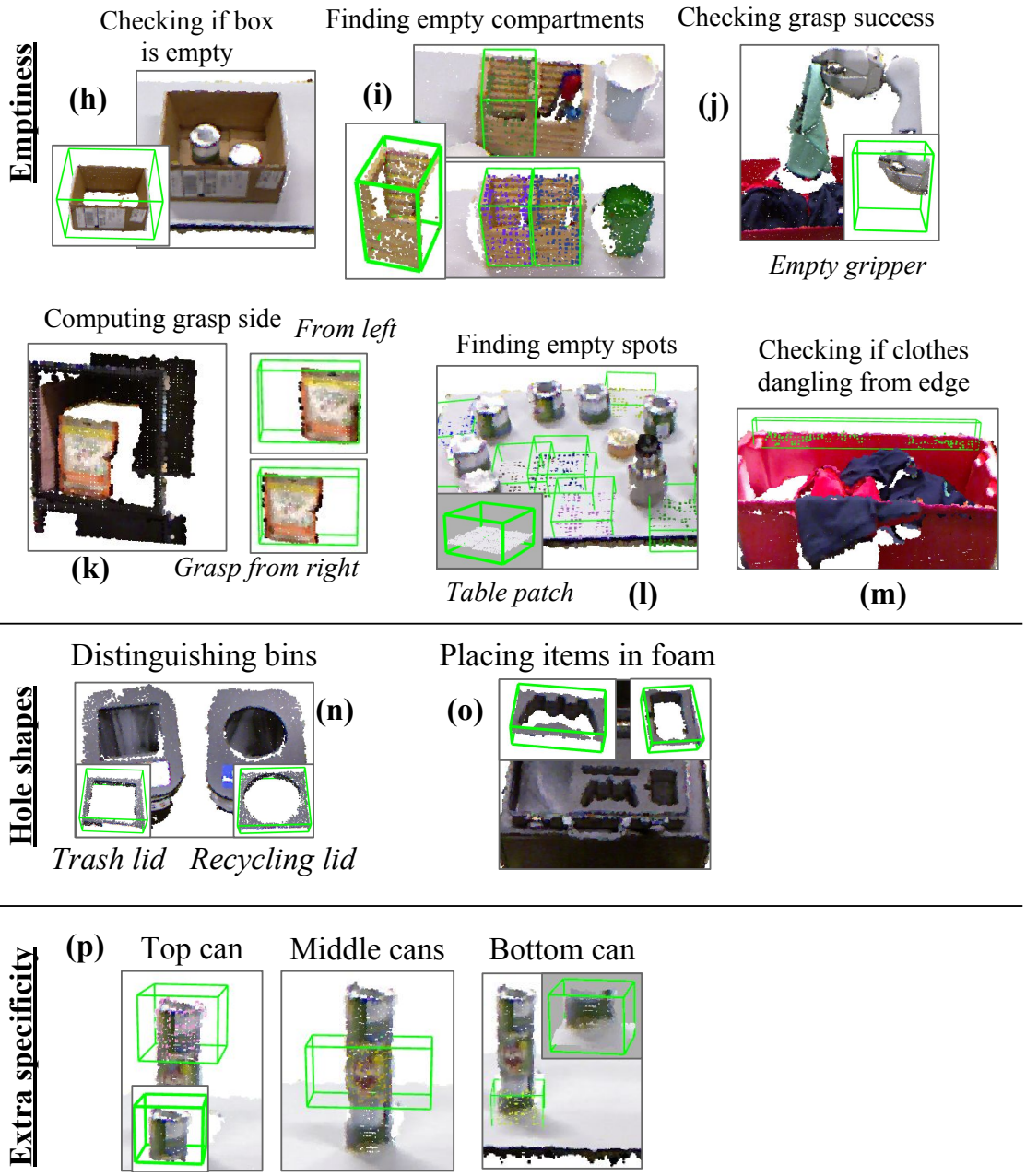


Figure 4.2b: Scenes and landmarks described in the case studies of *CustomLandmarks* (continued). Three categories of landmarks are illustrated above: emptiness, hole shapes, and extra specificity.

Parts of an object Sometimes it is helpful to model just part of an object as opposed to the entire object. For example, in a stack of bowls, only the rim of the top bowl may be visible, especially if the bowls are located above eye level on a kitchen shelf. For this case, we created a landmark of the bowl rim, which allows us to locate the top bowl (Figure 4.2a).

Multiple objects may share an identical part. For example, researchers have developed 3D-printed adaptors that robots can securely grasp [148]. These adaptors can be attached to many kinds of tools, including non-rigid tools like feather dusters that would be hard to model in 3D. We created a landmark of one of those adaptors to locate the tools (Figure 4.2b).

Finally, users can specify parts of objects that serve as grasp points. Finding multiple grasp points is helpful because some parts of the object may be out of the robot's reach. Figure 4.2(c) shows how we used a small segment of a tripod to find multiple grasp points on the tripod.

Parts of the workspace Landmarks can also represent parts of the scene or of large objects. Examples include the handles of a drawer or the top of a chair (Figure 4.2e, g). We also made a landmark of a wall mount for a tool to help a robot hang a tool in an available space (Figure 4.2d).

Other workspace landmarks can serve as anchor points for the rest of the workspace. For example, on a mockup of a control panel, we localized the buttons, which are too flat to be landmarks themselves, by creating a landmark of a nearby prominent feature—in this case, a large dial (Figure 4.2f).

As with objects, it can be useful to locate just part of the workspace. For example, landmarks of table corners can be used to locate the corners of many different-sized tables.

Emptiness Many useful checks can be programmed by creating and searching for landmarks that model emptiness in some way. A simple use of emptiness in landmarks is to check whether a container or part of a container is empty. In one example, we created a landmark of an empty cardboard box so the robot could check if it was empty (Figure 4.2h). In another example (Figure 4.2i), we made a landmark of an empty side of a two-sided pencil holder, which allowed us to determine which side or sides were empty.

There are other creative uses of emptiness in landmarks. For example, to know whether the robot successfully grasped a thin, lightweight object, the robot may need to visually inspect its gripper. We were able to program this by creating a landmark of the gripper with empty space underneath (Figure 4.2j).

We programmed the robot to determine whether it should use its right gripper or its left gripper to grasp an object from a shelf. In the confined shelf space, there was only room to grasp the object from one side. To determine which side to use, we created an object landmark with space to its right and another version with space to the left (Figure 4.2k).

Landmarks can also be used to find empty regions of the workspace. For example, we were able to sample an empty spot on the cluttered tabletop by creating a landmark of a tabletop patch with empty space above it (Figure 4.2l). In our final example of using empty space, we envisioned the robot loading clothes into a basket. To check if clothes were hanging from the edge of the basket, we created a landmark of the basket edge with some empty space around it. Figure 4.2(m) shows that this landmark can distinguish between the back edge of the basket, which has nothing on it, and the front edge, which has a shirt hanging over it.

Hole shapes The use of empty space in our landmark representation means that users can create landmarks of workspace areas that are predominantly hole-shaped. For example, some trash bins and recyling bins have lids with differently-shaped holes. We were able to differentiate them using landmarks of the lids (Figure 4.2n). Die-cut foam packaging, which has shapes cut out for holding items during shipping (Figure 4.2o), is another example. Using hole-shaped landmarks, we were able to locate the spaces for two different objects in the foam packaging.

Extra specificity Figure 4.2(p) shows how landmarks can be designed slightly differently to represent different task requirements in the same scene. By adding empty space to the top of a landmark of a can, the system can locate the can on the top of a stack. In contrast, a landmark of a can that includes part of the tabletop represents the can on the bottom. Cans in the middle of a stack are harder to detect because a generic can landmark, without the extra features of the top or

bottom cans, will have many matches along the stack as the boundaries between cans is invisible. When working with stacks of objects, we expect that most users will be interested in locating only the object on the top.

4.2.1 Case study performance

In this section, we quantify the performance of the search algorithm on the case studies described in Section 4.2.

Precision and recall To measure the accuracy of the system, we recorded the precision and recall of the system using different error thresholds (Algorithm 1, line 13). For each case study, we created a landmark in one scene and searched for the landmark in a different scene, in which objects and/or the workspace changed positions. We labeled each scene with the correct locations of landmarks. For some scenes, we also searched for landmarks that were absent from the scene. For examples like finding an empty patch of tabletop, which had many possible correct answers, we manually graded the output of the algorithm instead of labeling the scene. Figure 4.3 shows the precision and recall curve for this experiment. In this experiment, we were not able to obtain 100% recall with any error threshold, because the non-max suppression step of Algorithm 1 prevents some results from being output. Table 4.1 shows the precision and recall numbers for several values of the error threshold.

Speed Most of the time spent in *CustomLandmarks* is in the loop between lines 5 and 11 of Algorithm 1, which evaluates a sample position in the scene. The number of samples, in turn, is determined by the size of the scene. Figure 4.4 shows a plot of the speed of the algorithm against the number of points in the scene. This experiment was conducted on a computer with an Intel® Core™ i7-4770 CPU and 16 gigabytes of memory.

These results show that the system is mainly useful for functional tasks in which speed is not critical. Implementing improvements, such as using GPU computing or algorithmically eliminating excess work, could make the system faster in the future. However, experienced robot program-

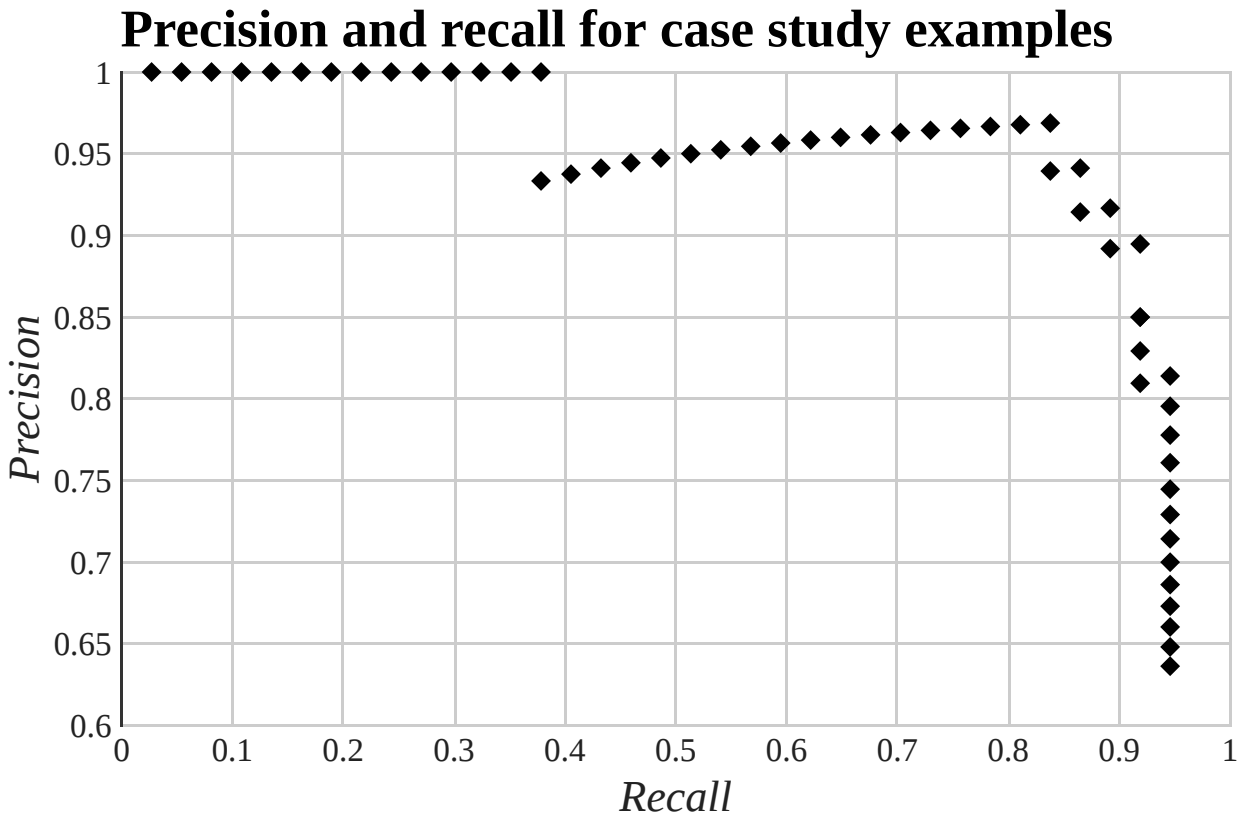


Figure 4.3: The precision and recall of the search algorithm for the case study examples described in Section 4.2. The vertical axis is shown with a minimum value of 0.6. The large drop in the top left is due to a false positive match that occurs at a relatively low error threshold.

Table 4.1: Numeric values of precision and recall as shown in Figure 4.3 for several values of the error threshold.

Error threshold	**Precision**	**Recall**
0.0049	1.00	0.38
0.0062	0.97	0.84
0.0064	0.94	0.87
0.0068	0.92	0.90
0.0070	0.90	0.92
0.0075	0.81	0.95

mers can mitigate this issue by preprocessing the scene in ways that are appropriate for the task. For example, if the landmark is in a tabletop scene and the tabletop is not a relevant landmark, the scene could be preprocessed to filter it out.

4.2.2 Limitations

Other than speed, *CustomLandmarks* has two main limitations, illustrated in Figure 4.5. The first is that the system relies on landmarks having a unique shape in the scene. The system works best in semi-structured environments where there are few distractions in the scene. To avoid this limitation, users can redesign the landmark so that it is unique, although it may not always be possible to do so.

The other main limitation is that the system can be brittle to viewpoint changes, such as when an asymmetric object is rotated. This is because we only capture landmarks from a single view of a point cloud. One way to mitigate this limitation is for users to create multiple landmarks representing rotated versions of the same object.

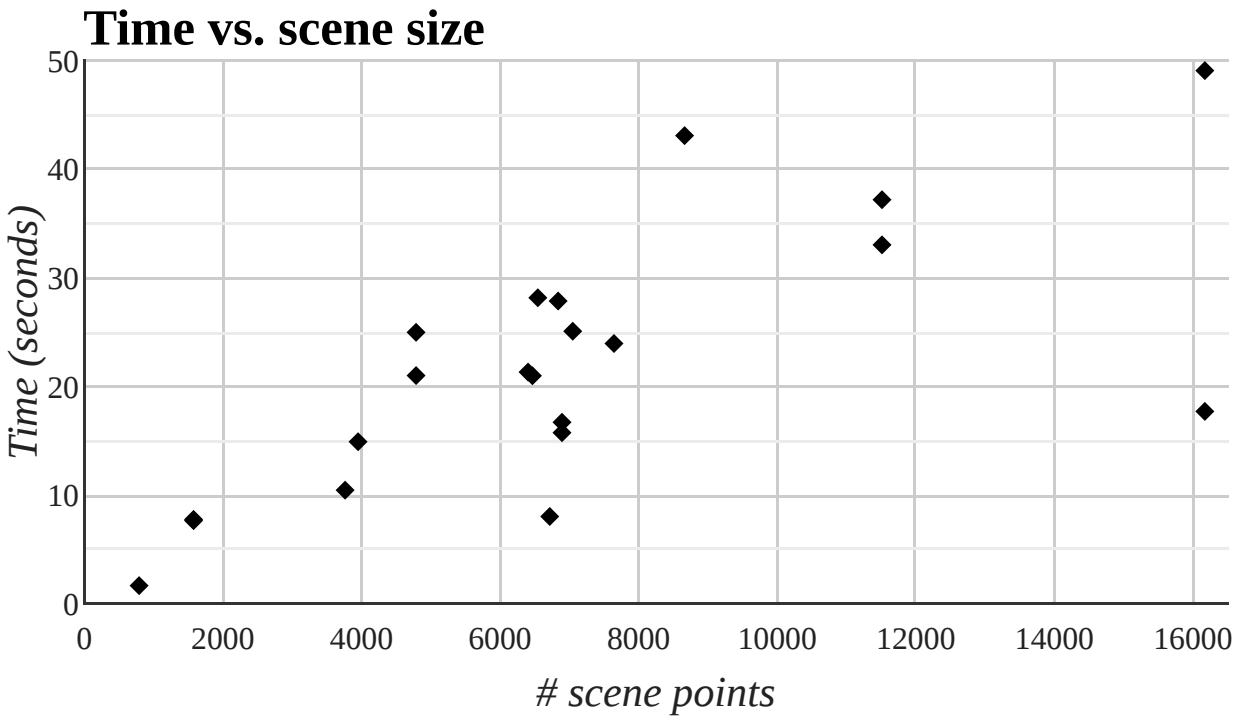


Figure 4.4: Time taken to search for landmarks over different scene sizes.

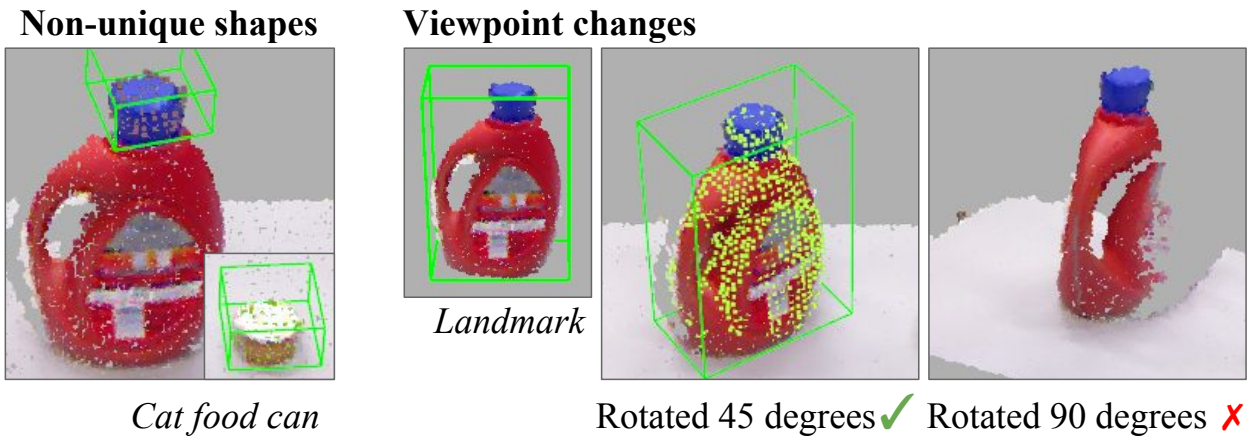


Figure 4.5: Examples of the limitations of *CustomLandmarks*. The left example shows how a landmark of a can food can could be confused for the cap of a laundry detergent bottle. The example on the right shows how a landmark of the bottle can be detected after a small viewpoint change (a 45 degree rotation), but not after a larger change.

4.3 Robot experiments

We ran a set of experiments to evaluate how well the results from our offline experiments transfer to real manipulation tasks. To this end, we used *CustomLandmarks* in conjunction with programming by demonstration (PbD) to program six manipulation actions on a PR2 robot.

4.3.1 Programming by demonstration with CustomLandmarks

To program manipulation actions on the PR2, we modified the PbD system in [4]. In this system, users program manipulation actions by guiding the robot's arms through a sequence of end-effector poses. Poses are defined either relative to the robot's base or relative to a landmark. Once the poses have been saved, the robot can repeat the action later. First, it searches for the necessary landmarks, then it adjusts the poses defined relative to those landmarks. Finally, the robot's arms move through the sequence of poses, opening or closing the grippers as programmed. Using *CustomLandmarks* allows users to define perceptual landmarks for their task, and PbD lets users create manipulation

actions based on those landmarks, all without writing any code.

4.3.2 Task descriptions

The tasks were based on the case studies described in Section 4.2. Each action we programmed was tested in 5 variants of the task, in which we moved objects and the workspace around, or we added or removed objects.

Tool rack We programmed the robot to either remove a tool from a tool rack or place a tool in an empty spot on the rack. The rack consisted of three 3D-printed wall mounts, and all of the tools had the tool adaptor. Both are shown in Figure 4.2(b, d). The tasks were varied by changing where the tools were placed. We also included scenarios where the tool rack was either full or empty, and the correct action was to not act. All but one of the tests succeeded. In the failed test, the robot located an empty wall mount but used too much force when placing the tool on it, knocking the mount off the wall.

Grasping a chair Previously, the PbD system we used only worked in tabletop scenes. PbD could work in non-tabletop scenes using our system, we programmed the robot to detect and grasp a chair (Figure 4.2g). We varied the scenes by setting the chair to different heights and moving it around. One of the five tests failed because our system could not locate the chair, showing how our system could fail if the viewpoint changed too much.

Picking bowls from a shelf In this task, the robot picked bowls from a shelf. The tasks were varied by placing bowls on different levels of the shelf, placing them in stacks, or removing all the bowls entirely. Because the shelf was at eye level and some bowls were stacked, it was necessary to create a bowl rim landmark, as illustrated in Figure 4.2(a). The robot successfully grasped a bowl in all five scenarios.

Waste bins For this task, we wanted the robot to either deposit an item into the trash bin or the recycling bin shown in Figure 4.2(n). In the different scenarios, we varied where the bins were

placed and which bin the robot needed to drop the item into. In one scenario, we removed the bin the robot needed to drop the item to see if it would mistakenly do so. All five scenarios we tested worked as expected.

Cluttered tabletop In this task, the robot needed to drop an item onto an empty spot on a table (Figure 4.2l). We varied the items and their positions on the table. One of these scenarios failed because we placed a large, flat box on the table, which our system thought was an empty patch on the table. This failure shows how our system needs landmarks to have a unique shape in the scene.

Control panel In this task, the robot pushed buttons on a mockup of a control panel (Figure 4.2f). Instead of locating the buttons directly, the robot instead located a dial, and pushed the buttons based on the location of the dial. We varied the scenarios by moving the control panel around and increasing its incline angle. The system failed to locate the dial in one scenario where the control panel was inclined.

4.3.3 Discussion

Overall, the robot succeeded in 26 out of the 30 total scenes it was evaluated in. Our search algorithm detected all of the different kinds of landmarks described in Section 4.2. It also correctly recognized all of the cases where needed landmarks were missing, which avoided having the robot execute an action it was not supposed to.

The actions we programmed using PbD did not require writing any code. However, adding programming logic to the system could have helped us avoid some of the failure cases we encountered. For example, when searching for an empty patch on a table, we could have filtered out all detections that were not located at the table's height.

4.4 User evaluation

Although *CustomLandmarks* can be used to design non-standard landmarks, it is important to know if users can actually understand and utilize the system in such a way. To address this question, we

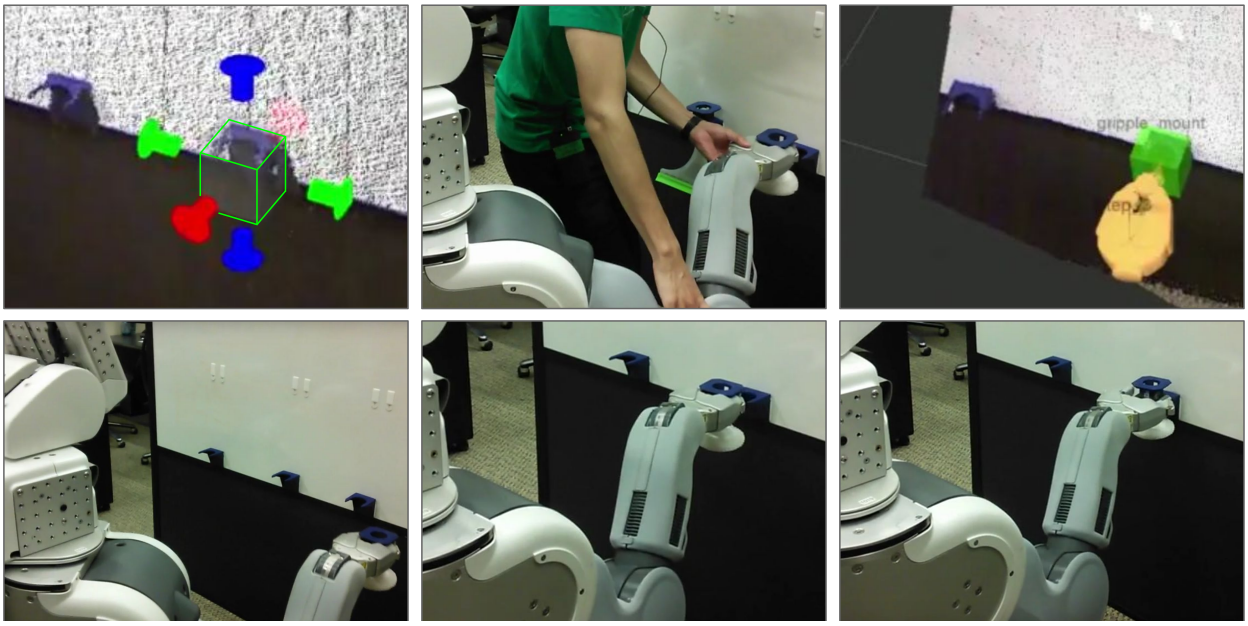


Figure 4.6: The PR2 robot using *CustomLandmarks* to place a tool on a tool rack. The top row shows the user defining the landmark and demonstrating the action and the robot localizing the landmark in a new scene. The bottom row shows the robot executing the task.

conducted a user study in which participants created landmarks for different robot tasks.

4.4.1 Study design

We asked users to create landmarks that would enable a robot to execute a set of three tasks. Tasks were assigned in random order. For each task, users had to create their landmark in one scene but test it in a different scene. This was to emulate real-world usage of the system. In the experiment interface (Figure 4.7) users could switch between editing and testing modes as many times as needed. An experimenter observed their actions and informed users when they had solved a task correctly.

The robot tasks were:

a. Placing an object on an empty spot on a cluttered table.

b. Picking bowls out of an eye-level shelf (including from the top of a stack of bowls).

c. Placing an object in an empty compartment of a two-compartment pencil holder.

Our main research question was whether users could design unconventional landmarks using *CustomLandmarks*. The correct landmarks were either object parts or scene parts. The first task required users to create a landmark of an empty patch of the table surface. To prevent trivial solutions with patches that were too small or too large, we required that the landmark be found in the test scene at least six times with no false positives. In the second task, users initially saw unstacked bowls, so the obvious solution was to make a landmark of a bowl. However, in the test scene, we turned one of the bowls into a stack, which required users to modify the landmark to represent a bowl rim. In the third task, we showed users an empty pencil holder, but in the test scene, one of the sides was filled with objects. As a result, users had to figure out that the landmark should be a single side of the container, as shown in Figure 4.7.

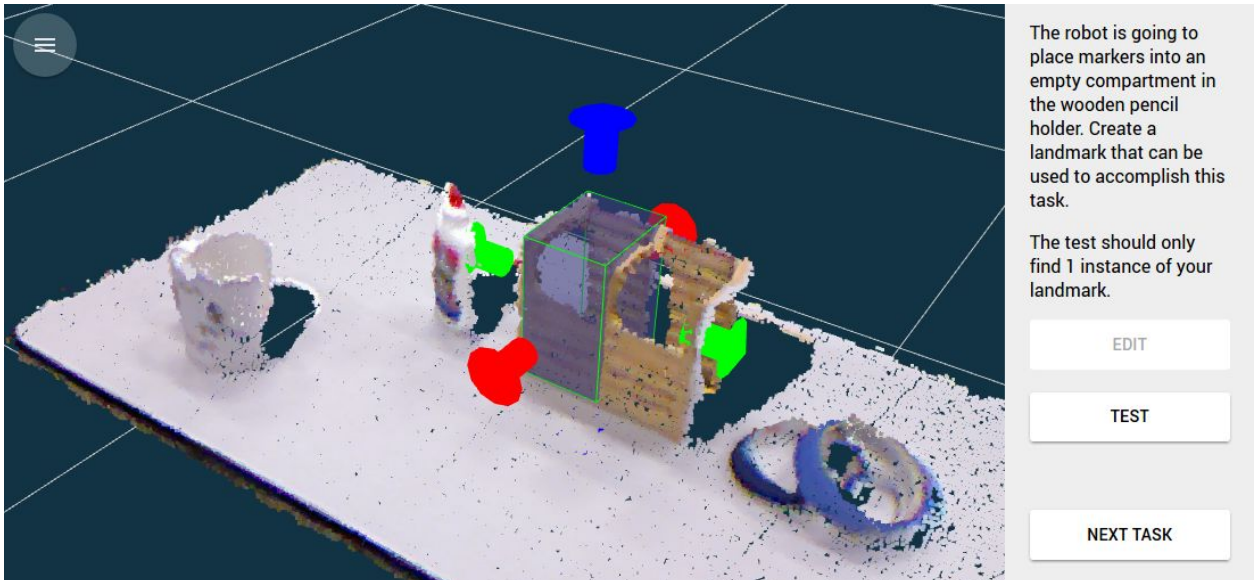


Figure 4.7: The *CustomLandmarks* user experiment interface in editing mode. The interface to create landmarks is shown on the left and a user study-specific display is shown on the right.

4.4.2 Training

To train users, an experimenter read from a script to explain the concept of landmarks. Users performed one training task where there was a line of cans on a table. Users practiced creating a landmark of a can. They were then introduced to the concept of a landmark representing empty space by modifying their landmark to only match with the rightmost can. The training session took approximately five minutes to complete.

4.4.3 Participants

We recruited participants through email and social media posts targeting undergraduate and graduate engineering students at the University of Washington. All participants were required to be novices to *CustomLandmarks*. This represented the population that would benefit from the system the most: technical, college-educated people who are not necessarily experts in computer vision or robotics. We recruited 15 participants, 9 male and 6 female. Their ages ranged from 18 to 32 and

Table 4.2: Mean task performance measures from the *CustomLandmarks* user evaluation. Task numbers are as given in Section 4.4.1.

Task	Total time	Edit mode time	# of edits
Task A	5:55	3:16	5.64
Task B	4:15	2:23	4.5
Task C	3:38	2:33	3
All tasks	4:34	2:44	4.35

averaged 23.3. All participants received a $10 gift card for completing the study.

4.4.4 Measures

Because an experimenter told users whether each task had been completed correctly, we measured task performance using time spent per task and number of tasks completed within the 30 minutes allotted for the study. The experiment interface also measured the time spent editing and testing landmarks. Finally, we administered the widely-used NASA Task Load Index (TLX) survey [61] to measure the subjective workload of the tasks.

4.4.5 Results

14 out of the 15 participants correctly completed all of the tasks within the time limit of the study. The average time to complete a task correctly was 4:34 minutes, with 2:44 minutes of that time spent in editing mode. Performance measures for each task are summarized in Table 4.2.

There are many ways to interpret TLX survey ratings [61], so Table 4.3 reports the average ratings for each workload category. The Raw TLX score, an average of all the categories, was 44.9, which is in the 40th percentile of TLX scores according to one meta-analysis [57].

Table 4.3: NASA-TLX survey results from the *CustomLandmarks* user evaluation.

NASA-TLX subscale	**Mean workload rating**	**Stdev**
Mental demand	54.67	4.13
Physical demand	25.00	4.28
Temporal demand	29.00	3.99
Performance	29.00	4.49
Effort	54.33	3.48
Frustration	32.67	5.38
Raw TLX score	44.93	15.75

4.4.6 Discussion

Our results show that users can learn to use *CustomLandmarks* and design unconventional landmarks in a short amount of time. Additionally, users were able to generalize from the training task, in which they created a landmark of a whole object, to tasks in which the landmarks represented parts of objects or parts of the scene.

In the study, users could rapidly switch between editing and testing modes to refine their landmark. However, when actually using *CustomLandmarks*, there may be a larger time gap between when a landmark is created and when it is used. As a result, we suggest that any tool for creating custom landmarks should incorporate a testing function that lets users try out the landmark on pre-recorded scenes.

Chapter 5

SHELF SEGMENTATION

As we described earlier, tabletop segmentation is a simple perceptual primitive that developers have used for robot manipulation tasks (*e.g.*, [4]). Tabletop segmentation algorithms assume that there is just one surface. However, recent events like the Amazon Picking Challenge [7] have challenged developers to work in and around shelves. In this section, we describe an algorithm that extends tabletop segmentation to shelf scenes or other scenes with multiple horizontal surfaces.

In tabletop segmentation, we assume that the scene comprises a tabletop surface with some objects on it, with the objects spaced apart. The robot senses the table as a point cloud using an RGBD camera. The system uses an algorithm based on random sample consensus (RANSAC) [52] to locate the tabletop surface. Points above the surface are clustered using an algorithm like Euclidean cluster extraction [112], and clusters separated by a certain distance are considered to be separate objects.

One advantage of such a system is that it works in a predictable and understandable way, as long as developers are willing to constrain themselves to work within certain limitations. For example, suppose the object clustering algorithm incorrectly considers two closely-spaced objects to a single, larger object. Developers can immediately understand the cause of the error and ensure that there is more space between the two objects in the future. Alternately, they can tune the parameter of the algorithm that determines the minimum spacing needed between separate objects. It is also straightforward to visualize the output of tabletop segmentation by coloring or drawing bounding boxes around the tabletop and the objects.

Our shelf segmentation algorithm works in a similar way to tabletop segmentation. This maintains the predictability and interpretability of the algorithm. We compared our shelf segmentation algorithm against an existing algorithm, Organized Multi-Plane Segmentation (OMPS) [127],

which was implemented and deployed in the widely-used Point Cloud Library (PCL) [114] and achieved higher precision and recall.

5.1 Shelf segmentation algorithm

Our shelf segmentation algorithm builds on the RANSAC algorithm used by tabletop segmentation. To locate a tabletop surface, the algorithm samples three points at random from the scene, and fits a plane through the three points. It filters out planes that are not horizontal, according to a user-specified angular threshold. The algorithm then computes the number of points in the scene that are nearby to the plane. These nearby points are called *inliers*. The algorithm records the number of inliers associated with that plane. This process is repeated with a different set of three randomly-chosen points, up to a certain number of iterations. Then, the algorithm outputs the plane with the largest number of inliers. With a large enough number of iterations, it will be the case that the plane with the most inliers corresponds to the tabletop surface.

Our algorithm alters this algorithm slightly. Rather than output the plane with the largest number of inliers, we output all planes with a certain number of inliers. The exact number of inliers required is a tunable parameter that depends on the type of surface and the resolution of the camera. We assume that we are only searching for horizontal planes. This allows us to only sample a single point at a time, and fitting a horizontal plane through that point. Doing so makes the sampling process more efficient, because we are not spending time sampling non-horizontal planes, only to throw them out. Finally, because it is possible to sample many points on the same surface, we have another tunable parameter that specifies the minimum distance between two planes. This allows us to ignore point samples that are very close to a plane we have previously found.

5.2 Comparison with OMPS

To evaluate our shelf segmentation algorithm, we compared it to the Point Cloud Library's implementation of Organized Multi-Plane Segmentation (OMPS). We created a dataset with 26 point clouds of shelf and tabletop scenes and ran both our algorithm and OMPS on each scene. Below, we describe the procedure, measurements, and results for this evaluation.

5.2.1 Preprocessing

For many robot manipulation tasks, programmers will crop the scene from the robot's RGBD camera to an area representing the robot's workspace. This helps to eliminate unncessary processing of distractions such as faraway walls or the floor. OMPS requires the RGBD data to be *organized*, meaning that the point cloud data is arranged in sequential order (*e.g.*, row-major order). However, cropping a point cloud typically involves removing point cloud data, which destroys the sequential ordering of the data. To accomodate this, we don't remove any point cloud data when cropping. Instead, we set the point data of a cropped point to be *not a number* (NaN), which is used to indicate that a pixel did not receive a depth reading from the sensor.

OMPS also requires normal vectors for each point (our algorithm does not). In our evaluation, we computed normal vectors for each point using PCL's parallelized normal estimation method. This works by sampling a local region around the point and fitting a small planar patch to those points, which determines the normal vector to the patch. Because cropping a point cloud can affect this computation, we compute normals for the point cloud before cropping it.

5.2.2 Parameter tuning

Both algorithms have several tunable parameters, such as the minimum number of points a surface must have. OMPS also has a default set of parameters in its PCL implementation. For a fair comparison, we chose a set of default parameters for our algorithm and compared both algorithms with their default parameters. We also tried hand-tuning the parameters for both algorithms in each scene we tested. This covers two types of use cases: one where the developer is unsure of how to set the parameters and wants to use a reasonable set of defaults and one where the developer wants the best possible performance for a specific task. The parameters we tuned and their default values are shown in Tables 5.1 and 5.2.

Name	Description	Default value
iterations	Number of RANSAC iterations to run	300
angular_threshold	Maximum allowable off-horizontal angle	10 degrees
distance_threshold	Maximum distance of an inlier to the plane	1 cm
min_inliers	Minimum points a surface must have	8000

Table 5.1: Tunable parameters for our shelf segmentation algorithm. For similar parameters, we use the same names as OMPS does.

Name	Description	Default value
normal_radius	Radius around a point from which to estimate a normal vector	3 cm
angular_threshold	Maximum allowable off-horizontal angle	3 degrees
distance_threshold	Maximum distance of an inlier to the plane	2 cm
min_inliers	Minimum points a surface must have	1000

Table 5.2: Tunable parameters for the OMPS algorithm.

5.2.3 Measurements

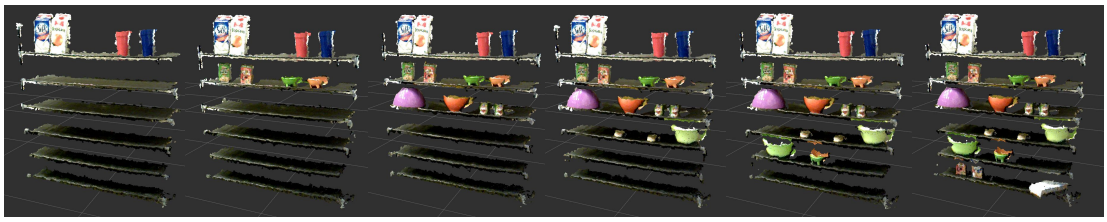
To evaluate the two systems, we measured the precision, recall, and speed of shelf detection for each algorithm and scene. To determine whether a detected surface was correct, we fit an oriented bounding box around it and inspected a visualization of the result.

OMPS requires normal vectors for each point in the input scene. Normal vectors are used for other point cloud processing algorithms, so it is not necessarily fair to add the time spent computing normal vectors to the total time cost of OMPS. For a fair comparison, we measured the time spent computing normal vectors separately, which allows us to see the time it takes to run OMPS with and without normal estimation. We performed the evaluation on a computer with a 3.4 GHz Intel Core i7-4770 CPU and 16 GB of RAM.

5.2.4 Dataset

To test the two algorithms, we created a dataset with 26 scenes, shown in Figure 5.1. We collected the dataset using a Fetch Robot equipped with a Primesense Carmine 1.09 RGBD camera at a resolution of 640x480 pixels. The dataset includes three types of shelf and one tabletop surface. One of the shelves is a cloth shoe rack that has many closely-spaced levels. One of the other shelves has side walls and a back wall, while the third shelf has narrow columns holding up each shelf. We included one tabletop surface to demonstrate that the shelf segmentation algorithm can also be used for tabletop segmentation, since a table can be thought of as a single-level shelf. Each scene has a different arrangement of objects on the shelf. Some have no objects, while others have large numbers of objects on each level. In general, having more objects on the shelf makes it harder to correctly locate the shelf surfaces, because the tops of the objects themselves could be incorrectly interpreted as surfaces. Figure 5.1 shows how we divided the scenes into groups. Each group shares the same view of the same shelf, with only the object arrangement differing.

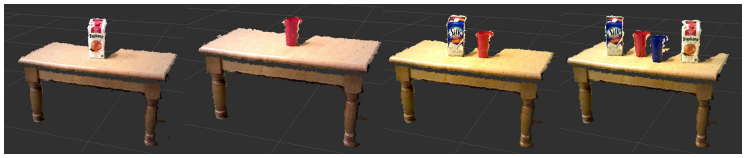
(a) Shoe rack



(b) Wooden shelf

(c) Wooden shelf rotated

(d) Tabletop



(e) Black shelf top

(f) Black shelf bottom

(g) Shoe rack random

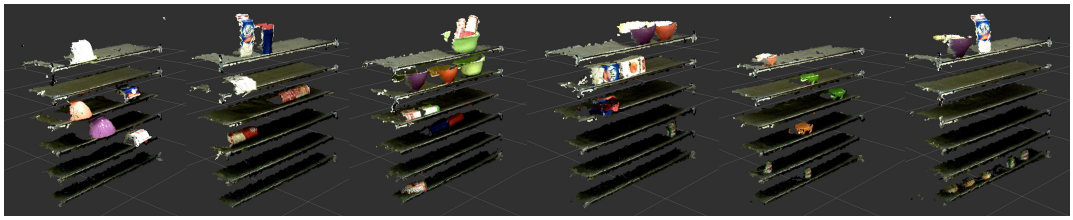


Figure 5.1: The dataset used to evaluate our shelf segmentation algorithm. (a) is a shoe rack with progressively more objects placed on it. (b), (e) and (f) are similar, but with different kinds of shelves. (c) is the same as (b), but rotated slightly (demonstrating that the shelf need not be axis-aligned). (d) is a set of tabletop scenes. (g) is the same shoe rack as (a), but with objects placed in more random configurations.

5.3 Results

The precision and recall results are reported in Figures 5.2. Using the default set of parameters, our algorithm achieved 98.78% precision and 91.60% recall, while OMPS achieved only 50% precision and 14.15% recall. After tuning parameters for both algorithms on each scene, we improved the performance of our system to 99.25% precision and 99.25% recall. Parameter tuning also benefited OMPS, boosting its performance to 97.92% precision and 44.34% recall.

Figure 5.3 shows the time each algorithm spent finding surfaces in the dataset. In Figure 5.3a, we report the time spent using OMPS excluding the normal estimation process, since developers sometimes will estimate normals in a scene for other purposes. By this measure, OMPS takes only 194-228 ms to find surfaces in a scene. Our algorithm takes longer, about 949 ms per scene on average. When tuning the parameters for each scene, we reduced the time spent to 489 ms on average. This was mainly achieved by reducing the number of RANSAC iterations for each scene.

However, if the developer has not already precomputed the normal vectors for the scene, they will need to do so in order to use OMPS. Figure 5.3b shows that the time spent using OMPS in this case will be dominated by the normal estimation time. Including normal estimation, the average time spent to use OMPS on a scene is 1429.9 ms, compared to just 194 ms spent using OMPS without normal estimation. In this scenario, our algorithm runs faster than OMPS does.

Some examples of correct and incorrect surface segmentations are shown in Figure 5.4.

5.4 Discussion

In our evaluation, we found that our shelf segmentation algorithm achieves much higher precision and recall than the OMPS algorithm does in our dataset. This held true after hand-tuning the OMPS parameters. While we were able to improve its precision to be comparable with our algorithm's precision, its recall remained low, just 44.34% on average. It may be the case that our dataset differs from what the creators of OMPS intended. Although we did not specifically analyze the effects of individual parameter changes on precision and recall, in almost every scene, we increased both the `distance_threshold` and the `min_inliers` parameters. One idea for future work

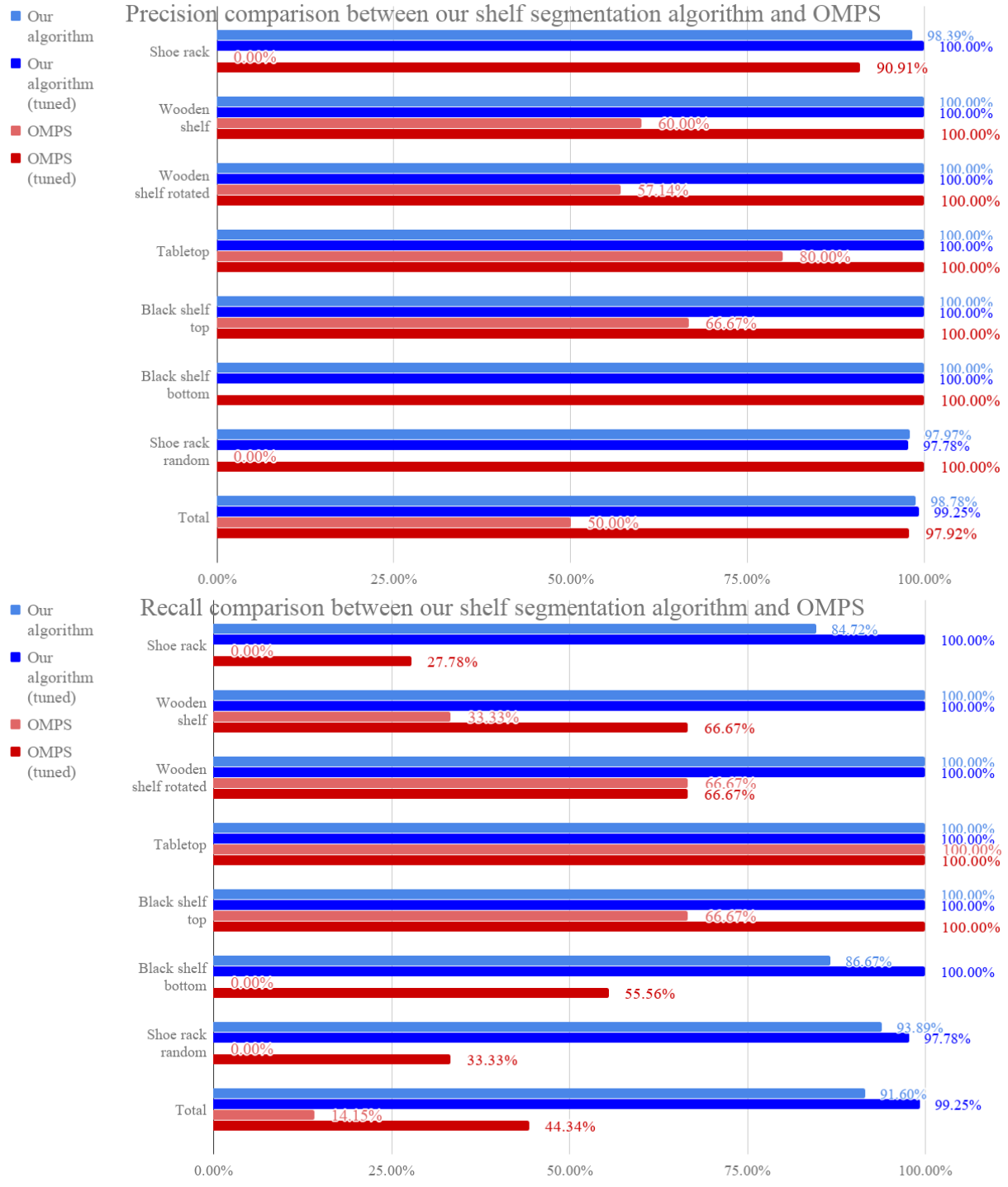
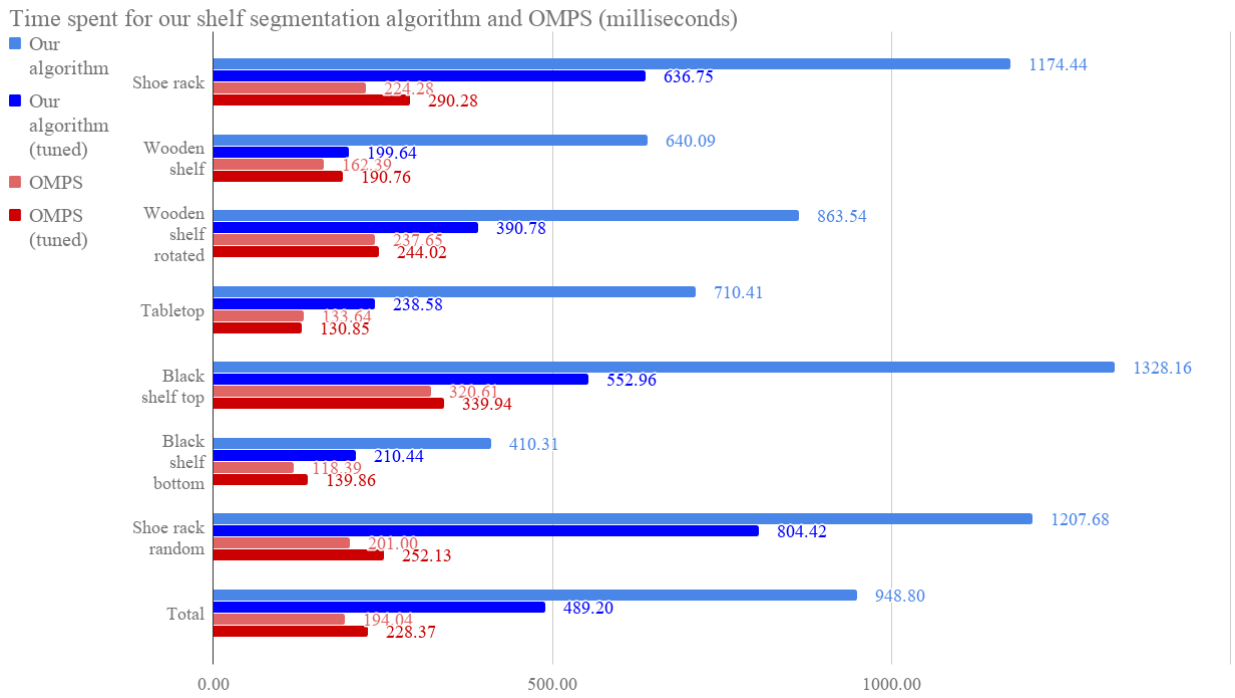
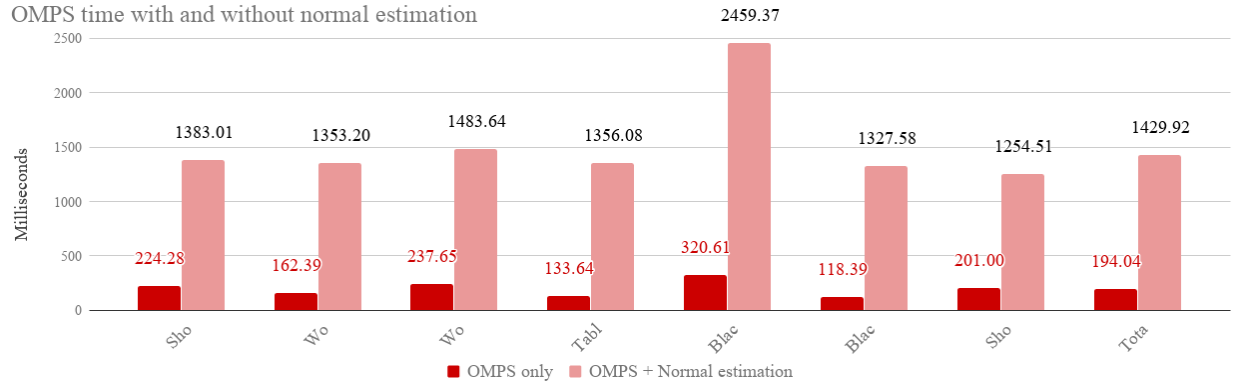


Figure 5.2: Precision and recall comparison between our shelf segmentation algorithm and OMPS.



(a) Time each algorithm spent finding surfaces in our dataset. For OMPS, we exclude the time spent estimating normal vectors in the scene.



(b) Time spent using OMPS with default parameters including and excluding normal estimation.

Figure 5.3: Speed comparison between our shelf segmentation algorithm and OMPS.

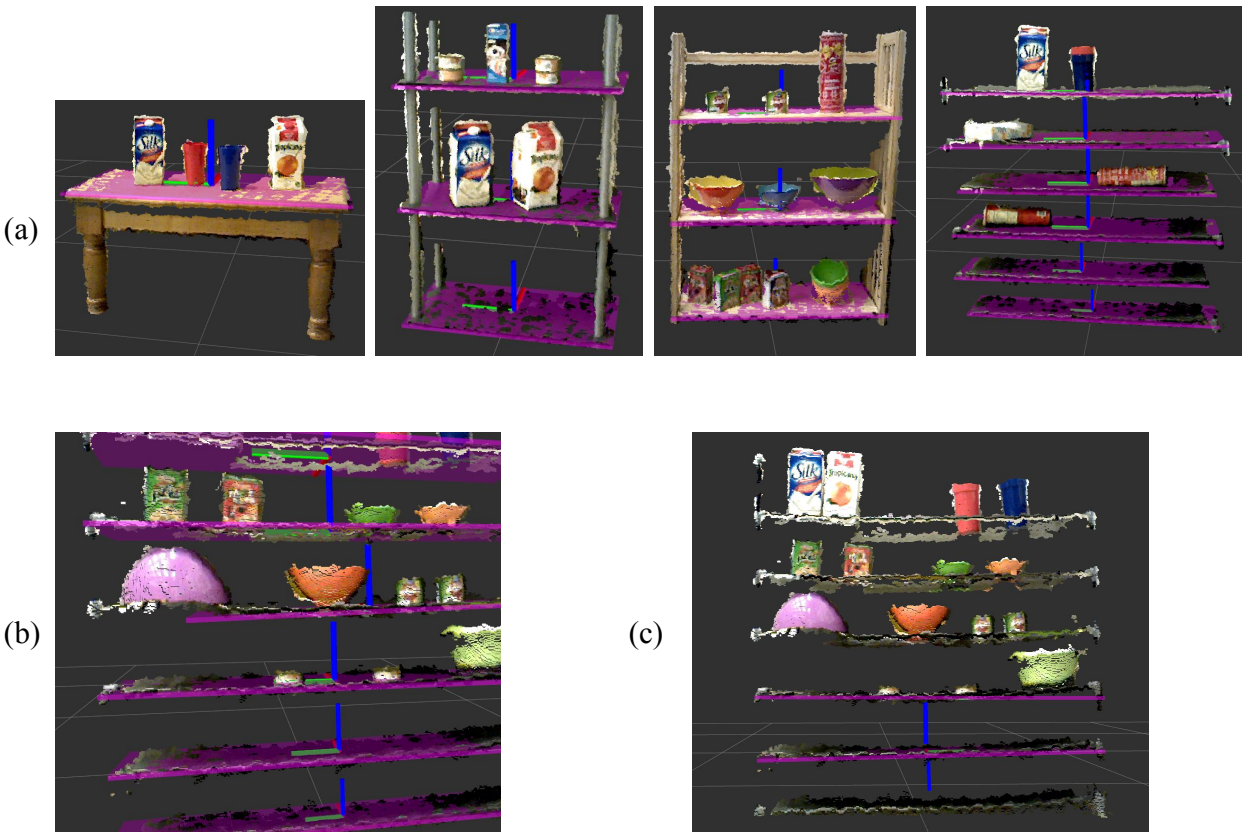


Figure 5.4: Examples of surface segmentations. Detected surfaces are visualized with a purple box. (a) Examples of our surface segmentation algorithm on different shelves. (b) An example of an incorrect surface detection by our algorithm. The large object on the left occludes the left edge of a surface, so we do not extend that surface fully to the left. (c) An example of an incorrect detection by OMPS, which failed to find all the surfaces in this scene.

is to create a tool that helps developers tune parameters for a target set of scenes.

We found that our algorithm is slower than OMPS if the developer has pre-computed normal vectors for the scene, otherwise our algorithm is faster. We did not optimize our algorithm for speed, but doing so would be a useful area of future work. Our implementation is available as an open-source ROS library, and documentation on its usage can be found at `https://wiki.ros.org/surface_perception`.

Chapter 6

MOTION SPECIFICATION AND PLANNING

A key characteristic that distinguishes mobile manipulator robots from other intelligent software systems is the robot's ability to effect physical movements in the environment. To program these motions, the developer must have a way to specifying the kind of motion to achieve and what the parameters of the motion are. Unless this specification is given in terms of low-level commands, the robot must be equipped with planning software that generates motions while avoiding collisions with obstacles. Motion specification must also work in conjunction with perception systems so that the robot can locate objects and obstacles.

For our research, we wanted users to have simple, easy to understand ways to specify motions. This goal is largely met by existing approaches such as programming by demonstration (PbD), which we described in Section 2.2. In this chapter, we provide background information on how we used and extended these systems. We limit our discussion to motions for manipulation tasks by robot arms with parallel-jaw grippers for end-effectors, such as those on the PR2 or Fetch robots described in Section 1.3.

6.1 Programming by demonstration

Programming by demonstration (PbD), sometimes referred to as learning from demonstration or lead-through programming [83], is a common technique for specifying robot motions. As discussed in Section 2.2, there are many variations on programming by demonstration. In this section, we will briefly discuss PbD in the context of our research.

The common idea behind PbD is that expressing arm motions as sequences of joint angle values can be unintuitive for human users. So, PbD systems let users specify arm motions in a more intuitive way. In our work, we focus on *kinesthetic* programming by demonstration, where pro-

grammers can physically move the robot's arm to a desired position, which is more intuitive. This approach can only be used with robots designed to be used safely around humans, and generally does not apply to larger robots with potentially hazardous arms.

Once the user has moved the arm into the appropriate pose, the robot records the arm pose using internal sensors. The robot can then be guided through a sequence of such poses. During or in between movements, the robot can open or close its gripper (or activate a different kind of end-effector like a suction cup). Replaying the arm poses and end-effector commands forms what we call a *PbD action*.

6.1.1 Pose representations

Typically, the lowest-level motion control interface that a robot application programmer will work with is a *joint trajectory* interface. This is true of both the PR2 and the Fetch robots. A joint trajectory is a list of *joint trajectory points*. Each joint trajectory point specifies an angle for each of the robot's joints as well as a timestamp that specifies when the arm should achieve the joint angles. There are two big downsides to specifying arm movements in terms of joint angles. First, it is unintuitive. Developers are more familiar with thinking about objects in Cartesian space, because that is how people experience the world in day-to-day life. Second joint angles cannot be parameterized based on perception data. Joint angles specify the pose of the end-effector relative to the base of the robot arm. However, developers typically want to specify an end-effector pose relative to a landmark in the scene.

An alternative that solves both problems to specify an arm pose in terms of the end-effector pose. That is, the developer states where the gripper should go in Cartesian space. The rest of the arm's pose is not explicitly specified. Instead, the robot can use an inverse kinematics (IK) solver to determine what joint angles would put the gripper in the desired location. An end-effector pose can be parameterized with according to the pose of landmarks using transformation arithmetic. The downside to this approach is that a given end-effector pose might not be reachable by the robot. There may also be multiple joint angle configurations that achieve the same end-effector pose, in which case the robot might arbitrarily choose one of the valid configuration.

6.1.2 Parameterizing PbD actions with landmarks

As discussed in Section 3.1, the robot must be able to make slightly different motions when the locations of objects or other task-relevant landmarks change. There are two main points in the programming process when this becomes relevant. The first is when the user records the demonstration, and the second is when the robot plays back the demonstration. Below, we describe how our implementations of PbD handle these two situations.

When the user is recording a demonstration, he or she must, for each arm pose, choose which reference frame (*i.e.*, landmark) the pose should be described relative to. For example, a sequence of arm poses relative to the robot's base might be useful for creating gestures such as a wave. More commonly, arm poses will be relative to objects in the scene, so that the robot can pick up an object and place it back down. In our work, we often automatically select a reference frame based on the following heuristic: the reference frame for the arm pose is the closest object frame to the robot's gripper that is also within a certain distance d. If no landmarks are within d distance of the gripper, then a frame affixed to the robot's base is chosen. The user can then manually change the reference frame using the programming interface if the heuristic is incorrect. Finally, the robot saves the pose of the end-effector relative to the chosen reference frame.

When replaying a PbD action, the robot is given the end-effector pose relative to the chosen reference frame (*i.e.*, the landmark). Generally, the robot's high-level motion planning interface requires the pose to specified in the frame of the robot's base or another fixed frame in the world. As we will show below, this can be done simply once the robot identifies the location of the landmark relative to the fixed frame. If the landmark refers to an object in the scene, then the robot must first locate the object. If the object is not present in the scene, the PbD action cannot be completed. It may also be the case that multiple instances of the same object are present in the scene. In that case, the robot must somehow decide which object to use. This is the *landmark registration* step described in Section 3.2.

6.2 Common transformation formulas

We use the common world modeling approach of representing landmark locations with 6D poses. These poses can be mathematically represented using homogeneous transformation matrices. A good primer on the use of transformation matrices and transformation arithmetic can be found in Chapter 2 of [41]. However, we will briefly provide common formulas we use in the process.

We use the same notation from [41], where ${}^{A}_{B}T$ is a homogeneous transform matrix that denotes a reference frame T of landmark A, described from the perspective of reference frame B. Let W refer to a fixed *world* frame, G refer to a frame attached to the robot's gripper, and O refer to a frame attached to an object. In our system, the robot uses its internal sensors to read the current end-effector pose in the world frame, the perception system provides the poses of objects in the world frame, and the motion planning system requires end-effector poses to be given in the world frame.

When recording an arm pose relative to an object, we are given the pose of the gripper in the world frame, ${}^{W}_{G}T$ and the pose of the object in the world frame, ${}^{W}_{O}T$. The pose of the gripper in the object frame is

$$ {}^{O}_{G}T = ({}^{W}_{O}T)^{-1}{}^{W}_{G}T \tag{6.1} $$

To replay an arm pose, we are given the pose of the gripper relative to a landmark. Suppose that landmark is an object O. The gripper pose is therefore ${}^{O}_{G}T$. Suppose the perception system locates the object relative to the world frame, ${}^{W}_{O}T$.

The pose of the gripper in the world frame is then

$$ {}^{W}_{G}T = {}^{W}_{O}T{}^{O}_{G}T \tag{6.2} $$

A more advanced use case is when specifying a grasp relative to a radially-symmetric object. In this case, we may want to sample a variety of poses about the object's axis of symmetry. Let $Rot(\hat{z}, \theta)$ be the homogeneous transform matrix for a rotation of θ about the z-axis, assuming the z-axis is the axis of symmetry. We define a virtual *rotated object frame*, R, which is a coordinate

frame with the same position as the object, but with a different orientation about the z-axis. This represents the object as if it were rotated about its axis of symmetry.

$$ {}^{O}_{R}T = Rot(\hat{z}, \theta) \tag{6.3} $$

$$ {}^{W}_{R}T = {}^{W}_{O}T\,{}^{O}_{R}T \tag{6.4} $$

We can then compute the grasp pose as we did in Equation 6.2, substituting the object frame (${}^{W}_{O}T$) for the rotated object frame (${}^{W}_{R}T$).

$$ {}^{W}_{G}T = {}^{W}_{R}T\,{}^{O}_{G}T \tag{6.5} $$

$$ {}^{W}_{G}T(\theta) = {}^{W}_{O}T\,Rot(\hat{z}, \theta)\,{}^{O}_{G}T \tag{6.6} $$

The transform_graph library

For expert roboticists, it is useful to have a *frame manager* library that can automatically compute frame transformations such as those described above. Rather than have developers write separate mathematical equations to compute different kinds of transformations, a frame manager solves transformation equations generally. First, developers specify the relationships between pairs of frames, which can be thought of as adding edges to a graph. Then, they can query the transformation between any two frames. A common open-source library for this is the ROS TF library (`https://wiki.ros.org/tf`). However, using TF can be confusing because it is a concurrent, asynchronous system. In our experience, TF is best suited for computing transformations between permanently-existing robot frames that change over time, and it is not well-suited for temporary analysis of temporary or hypothetical frame relationships.

To aid the software development of our research projects, we developed a C++ frame manager library called `transform_graph`. This is a synchronous library that allows users to specify frame relationships in an arbitrary topology, including non-tree structures, disconnected graphs, and cyclic graphs (in comparison, TF requires the topology of frame relationships to be a tree

structure). Users can then query the transformation between any two frames that are connected. The algorithm uses breadth-first search to find the shortest path between any two frames and returns the transformation between them. Open-source code for this library can be found at `https://wiki.ros.org/transform_graph`.

6.3 Web interfaces for PbD

In our research, we used and extended existing PbD interfaces for specifying motions (see Chapter 9) or eschewed the use of PbD entirely (see Chapter 10). However, based on feedback from user evaluations (Section 9.4), we also developed a prototype interface for programming by demonstration, called *RapidPbD*.

6.3.1 Motivation

The development of a new interface addressed several issues. First, existing PbD interfaces that we used (*i.e.*, [4]) were Linux-based applications designed for desktop or laptop computers. The system had specific software requirements and no simple installation or deployment process. In contrast, web-based interfaces are easy to deploy to wide variety of platforms, including non-Linux computers and all major mobile phone and tablet operating systems.

In Chapter 7, we describe another web-based interface that was deployed at a commercial robotics company. Non-software engineering employees were able to use the interface because it was accessible via the web. Ease of access also enabled a variety of other uses of this interface (see Section 7.5.4). The success of a web-based scripting interface also encouraged us to develop a web-based PbD system. Ultimately, our goal is to provide a fully web-based suite of programming tools for non-expert users.

6.3.2 Interface overview

Figure 6.1 shows a screenshot of *RapidPbD*. In the interface, motions are composed of *steps*, and steps are themselves composed of *actions*. An action represents a specification of a particular kind

of motion or perception primitive. The possible motion types are *Move to joint angles*, *Move to Cartesian pose*, *Open gripper*, *Close gripper*, and *Move head*. In our implementation, we only implemented a single perception primitive: shelf segmentation (from Chapter 5). In particular, it segments the surfaces in the scene and fits bounding boxes around objects resting on the detected surfaces. However, we have also experimented with integrating the *CustomLandmarks2D* prototype into *RapidPbD*, as described earlier in Section 11.1.1.

In *RapidPbD*, the robot will perform all of the actions of a single step in parallel. It will wait for all of the actions of a step to complete before executing the next step. This gives users the flexibility to efficiently run some actions in parallel while still enforcing ordering constraints. For example, a common use case is for the first step to be a "set up" step, in which the robot simultaneously moves its head to look at the workspace, moves it arms to the side to avoid blocking the camera view, and opens its grippers. On the second step, the robot is guaranteed to have a clear view of the workspace, so it can run the shelf segmentation system to locate objects in the workspace.

To simplify the development process, we developed additional software libraries that help integrate the robot with web interfaces. This software is described in greater detail in [67].

Although we have not formally evaluated *RapidPbD* with non-expert users, we incorporated user feedback from previous experiments with other PbD interfaces. For example, in Section 9.4, the interface was primarily controlled using a voice interface. We found that users had difficulty using the voice interface, especially for users with accents and in noisy environments. *RapidPbD* presents a touch interface that is more reliable. In other PbD systems, it is possible to use non-voice interfaces if the system is deployed on a laptop or nearby desktop. However, having to keep a laptop near the robot while programming can be obtrusive. *RapidPbD* mitigates this problem by shrinking to fit on smaller devices like smartphones, which are less obtrusive to use.

Aside from these differences, the operation of *RapidPbD* is similar to that of previous PbD interfaces. Users press a button to unfreeze the robot's arm, and then move the robot's arm to a desired pose. They then press a button to save the robot's arm pose. If the robot ran a perception action in a previous step, the arm pose can be defined relative to one of the discovered landmarks. There are also two special landmarks that are always available: the robot base and the base of the

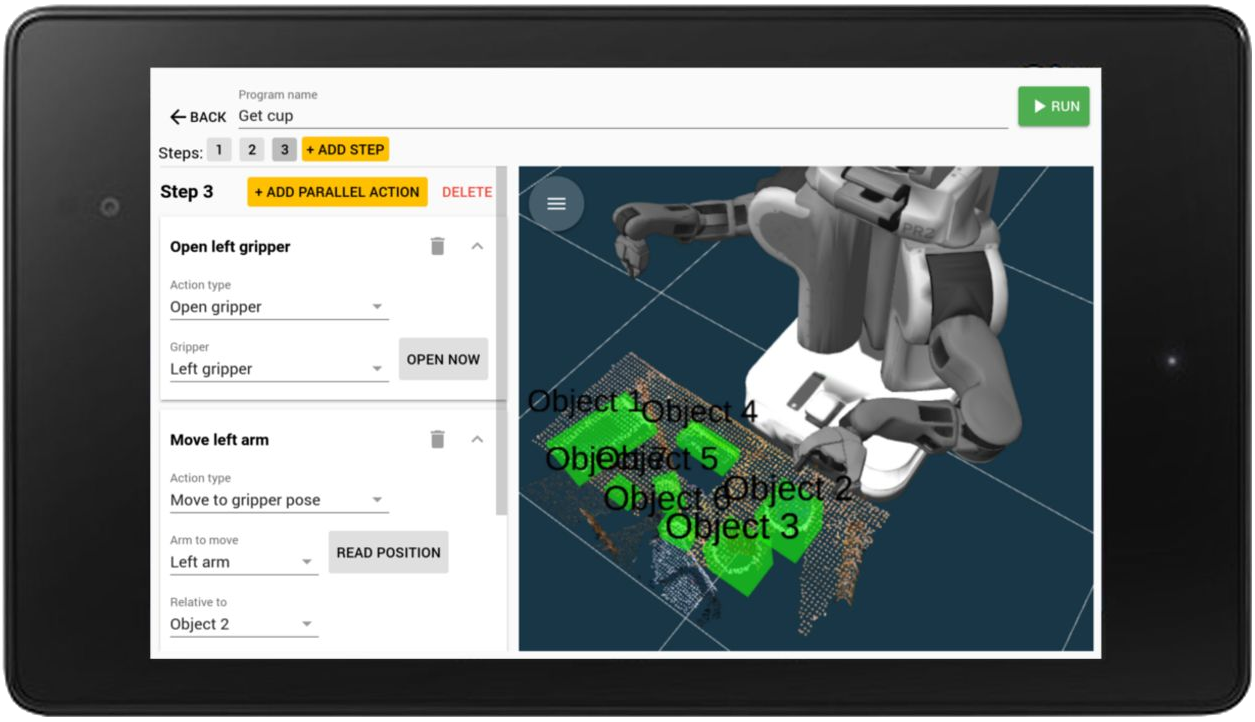


Figure 6.1: A screenshot of *RapidPbD* running in on a tablet computer.

arm. The distinction between these two landmarks is important for robots like the PR2 and Fetch that have adjustable-height torsos. Poses relative to the robot base are useful for actions that apply to objects with a fixed height in the world, regardless of the robot's height. Poses relative to the base of the arm are useful for gestural actions such as a wave, in which the robot's current height should be ignored (*e.g.*, poses programmed while the robot is at its maximum height may not be reachable when the robot is at its minimum height).

Chapter 7

TASK SCRIPTING

In this chapter, we discuss the third key component of our robot programming framework: task scripting. Programming end-to-end tasks like dropping off or fetching an item requires more than just arm motion specification. Often, these tasks will involve asking for user input, executing complex looping and branching logic, and handling errors. For this reason, developers need expressive scripting systems that give them the ability to write this logic when needed. At the same time, we also want to make it easy and quick to write these programs, so that experts and non-experts alike can rapidly prototype new programs for robots.

The primary system that we designed for this purpose is *CustomPrograms*. *CustomPrograms* is a programming system designed to help programmers rapidly prototype and customize applications for the robot. It was originally designed for use on the Savioke Relay robot, but we have continued to use it task-specific code on the PR2 and Fetch mobile manipulators. We have also used it for educational workshops on the Turtlebot robot. Our goals for *CustomPrograms* were 1) to be expressive enough for users to make new behaviors they wanted, and 2) to be easy to use for inexperienced programmers. We designed and implemented a simple API for the robot's capabilities, which we refer to as *primitives*. To make programming easier for inexperienced users, we provide a graphical interface for making programs, shown in Figure 7.1. However, users have access to standard programming language features, giving them control over the logic and flow of their programs. This interface is web-based, allowing users to create, edit, and run programs without special software.

CustomPrograms can be thought of as a compromise between simple interfaces designed for end-users and programming in a fully expressive, general-purpose programming language. In our design, we considered the fact that for commercial service robots, there are many types of

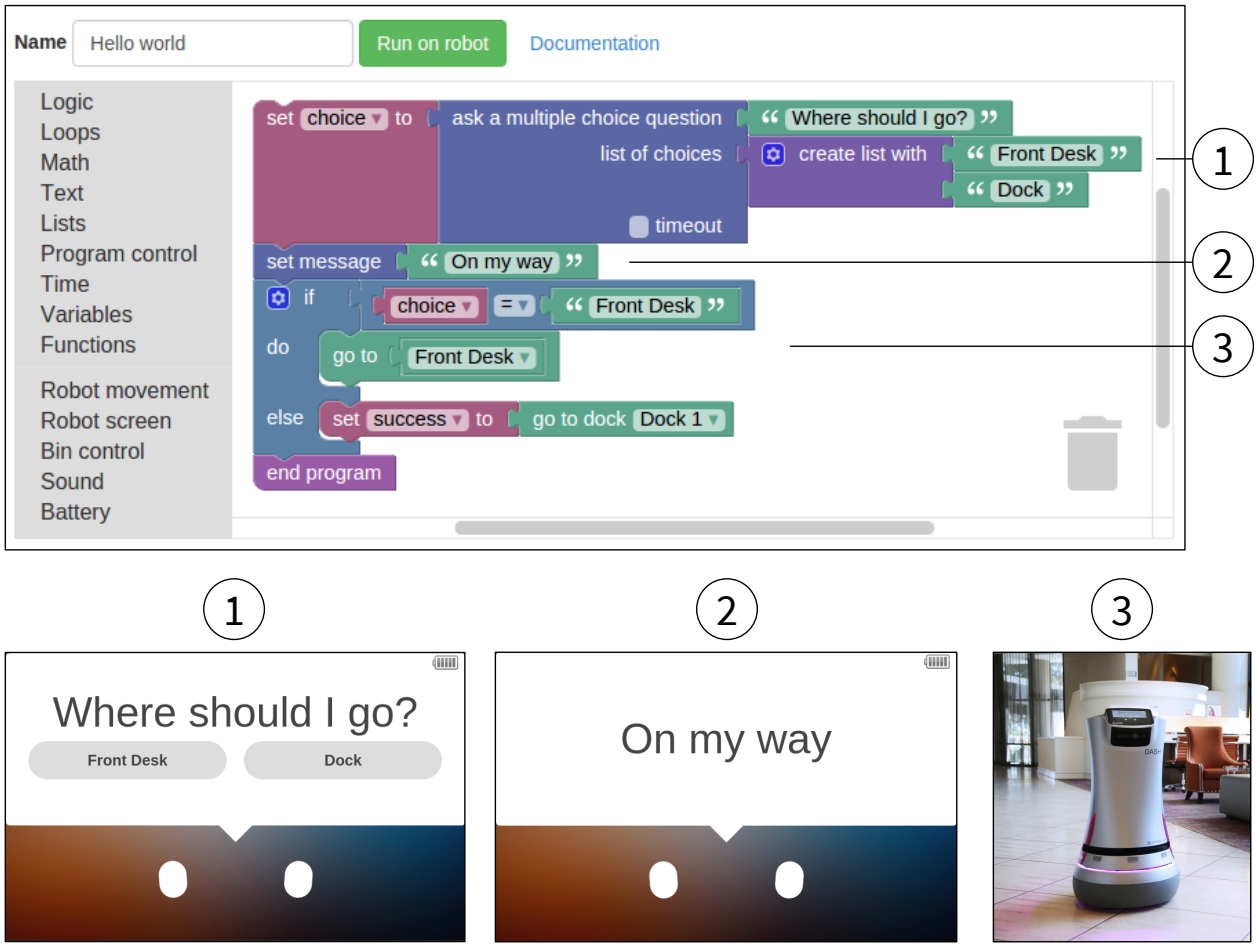


Figure 7.1: A sample execution of *CustomPrograms*. Steps 1 and 2 affect the robot's touch screen, as shown. The Relay robot is shown driving in step 3.

users, with a wide range of technical ability. On one end of the spectrum, there are end-users who might not have much programming experience, but who want an easy way to customize how the robot interacts with their guests. On the other end of the spectrum, there are experienced robot programmers. In the middle, there are users with some programming experience, such as designers, sales people, and hotel technicians. We designed our system with this diverse range of users in mind.

In the sections below, we describe the design and evaluation of *CustomPrograms*. We also discuss our choice of primitives for the Savioke Relay robot and describe our experience deploying the system on the Savioke Relay robot. We present results from interviewing Savioke employees about new use cases for their robot, both internally and for their customers. We also conducted a user study with people new to the interface and report results on the system's ease of use and expressiveness. Finally, we present two case studies on how *CustomPrograms* was used by Savioke. While we focus on how *CustomPrograms* was used with a mobile service robot in this chapter, Chapter 9 describes how we extended *CustomPrograms* for use with mobile manipulator robots.

7.1 CustomPrograms overview

7.1.1 The Relay robot

CustomPrograms was initially implemented on the Savioke Relay robot, described in Section 1.3. However, as we mentioned earlier, the system has been applied to other robots (all shown in Section 1.3). The Relay's main commercial use case is delivering items to rooms in hotels. When the hotel staff receive a delivery request from a guest, they enter a PIN on the robot. They then enter the room number to deliver to, load the bin, and send the delivery. The robot travels to the guest room, riding an elevator if needed. At the guest door, the robot calls the room phone. When the guest answers the door, the robot opens the bin. The robot also asks guests how their stay is. Finally, the robot leaves and goes back to its docking station.

7.1.2 Primitives

Each primitive robot behavior is a function call. The primitives are described below.

Robot movement

Table 7.1 shows a list of primitives related to indoor navigation, a core capability for service robots like the Relay.

Name and arguments	**Returns**
goTo(string location)	void
shimmy()	void
move(number forward, number right)	void
turn(number degrees)	void
distanceTravelled()	number

Table 7.1: Primitives related to robot movement.

The **goTo** primitive makes the robot navigate to a location named with a string ID. These IDs are human-friendly names, like “Front Desk” or “Room 201.” We assume that the robot has an existing mechanism for building maps and labelling locations with names. Users can provide the location ID manually, generate it programmatically, or use a helper block that lists all the location IDs the robot knows about.

The **shimmy** primitive is a short side-to-side swaying, which gives the robot a way to convey happiness. The **move** and **turn** primitives move the robot relative to its current position, allowing it to go to unnamed locations on the map.

Name and arguments	Returns
displayMessage(string text, number timeout=0)	void
askMultipleChoice(string text, string[] choices, number timeout=0)	string
askPasscode(string text, string passcode, number timeout=0)	bool
askNumber(string text, number timeout=0)	number
askRating(string text, number numStars, number timeout=0)	number
askScale(string text, number min, number max, string minText, string maxText, number timeout=0)	number
playSound(string sound)	void

Table 7.2: Primitives related to user interaction.

User interaction

Another important capability of the robot is the ability to interact with people. Our user interaction primitives are shown in Table 7.2. We include primitives not just for displaying messages on screen, but also for receiving user input, e.g., by asking a multiple choice question. Asking for a PIN number enables applications that require user authentication. The robot can also ask survey questions like star or scale ratings, which makes the robot capable of information-gathering applications.

The robot does not talk, but plays beeping and whistling sounds with the **playSound** primitive. This helps to manage expectations of the robot [149]. This was confirmed by Savioke designers in personal communications with the authors.

Battery and bin

The last set of primitives, shown below, relate to the robot's battery and bin.

Name and arguments	Returns
goToDock(string dock)	bool
batteryPercentage()	number
isCharging()	bool
raiseLid()	void
lowerLid()	void
isLidOpen()	bool

Table 7.3: Other primitives for the robot.

We included the **goToDock** block for making the robot go to a docking station autonomously. When combined with the battery percentage primitive, this enables programs to run the robot for a long period of time, charging its battery as needed.

7.1.3 Error handling

Error handling is an important topic when designing APIs. However, we wanted users to be able to focus on their programs, and assume that the primitives work as described. The only primitive that we offered error handling for was **goToDock**, which returns false if docking failed. For other robots, we could have more primitives return a boolean success value, depending on the robustness of the implementation.

7.1.4 Graphical interface

To facilitate programming for users without formal programming experience, we used a graphical interface called Blockly [54]. Blockly is not a programming language itself, but a framework for building visual programming languages. In Blockly, program elements such as constant values,

binary operators, while loops, or function calls are represented as blocks shaped like puzzle pieces. These blocks can be connected by stacking them, attaching values to inputs, or nesting blocks inside of other blocks. Blockly allows custom blocks to be made by defining a block's appearance, inputs, and outputs.

We designed custom blocks for each primitive, with inputs and outputs as shown in Section 7.1.2. We also provided standard programming language constructs such as loops, conditionals, variables, functions, strings, lists, math utilities, and logical operators. A full list of all the standard blocks can be seen on Blockly's website[1].

Blockly does a shallow form of type checking, which can eliminate some errors. For example, a block representing a string value can't be attached to a block expecting a number. However, it does not guarantee the correctness of the types used, and the type constraints can be circumvented.

A useful feature of Blockly is the ability to organize blocks in an arbitrary hierarchy of menus. We organized the primitives into categories as shown in Tables 7.1, 7.2, and 7.3. The blocks were color-coded so that blocks in the same category shared the same color. Additionally, snippets of code, represented as pre-assembled collections of blocks, can be added to the menus. Figure 7.2 shows how we used this. The *ask a multiple choice question* involves 4 different kinds of blocks, which we pre-assemble in a commonly used configuration.

7.1.5 Compilation and runtime

Each block generates JavaScript code, by first recursively generating code for its input blocks. The code for the block itself is then assembled along with the inputs appropriately. The code for the program as a whole is done by generating code for the blocks at the top level of the program.

When the user starts the program, the generated JavaScript code is sent to the robot. The robot runs a sandboxed JavaScript interpreter[2] on a Node.js server. The interpreter parses and executes the code, including primitives. In our implementation, the interpreter uses Robot Web Tools [126] to call ROS services and actions that execute the primitives.

---

[1] `https://blockly-demo.appspot.com/static/demos/code/index.html`

[2] `https://github.com/NeilFraser/JS-Interpreter`

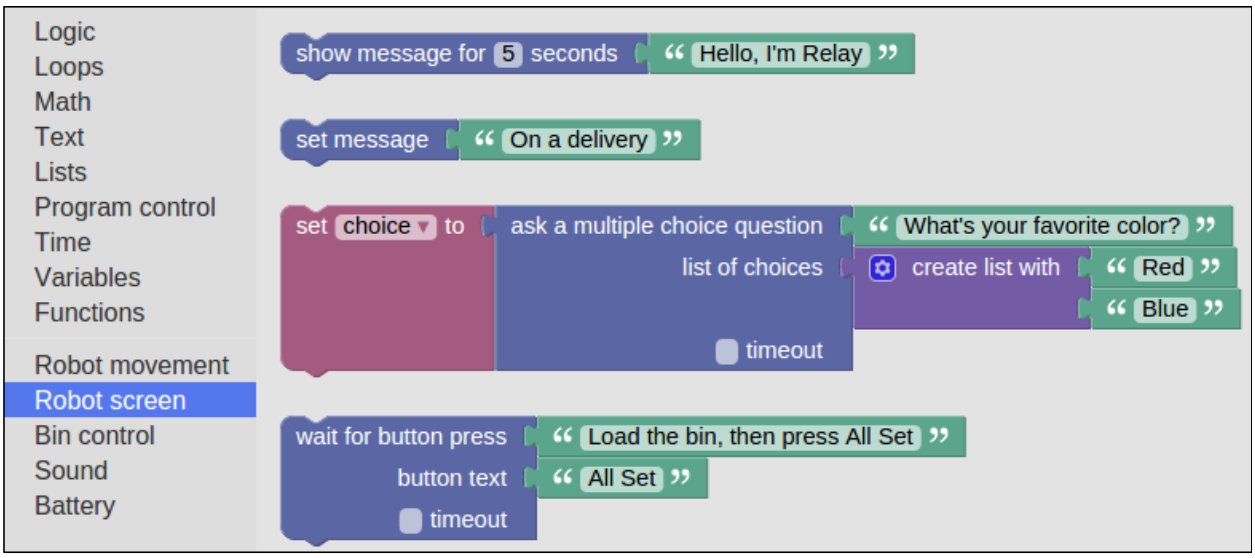


Figure 7.2: Screenshot of the interface, showing how blocks can be pre-assembled for convenience.

7.2 Use cases

Before developing *CustomPrograms*, we interviewed 2 employees at Savioke about the possible uses for the Relay robot, outside of deliveries in hotels. Additionally, we received informal feedback from employees at Savioke acting as points of contact for hotel customers. The use cases that turned up included both internal uses and customer-facing programs.

7.2.1 Employee interviews

We conducted separate semi-structured interviews with 1 designer and 1 business person at Savioke. For question 3 below, we showed the employees the default version of Blockly, from the Blockly website. The probe questions were:

1. When or where might you want to create a custom program for the robot?

2. Are there examples of programs you've wanted to make?

3. What do you think of Blockly?

4. How much time do you want to spend learning to program?

5. How much time do you want to spend making a demo program for a conference?

The business person said they would want to spend around 20 minutes making a program if they had to make it on the spot (e.g., at an off-site demo). The designer said that it would be fine to spend more time creating programs, as long as they were reusable. Both said an hour would be a reasonable time to learn to use the system.

7.2.2 Use cases

Customizing deliveries

One use case, mentioned by the designer, would be for hotel staff to customize the robot's interaction with the guest. For example, the robot could show "Happy birthday!" on screen, if applicable. Having this capability would also make it easy to pick up items like laundry from a room, instead of doing a delivery. The robot would just have to visit the room with an empty bin, and say something different at the door. The designer also mentioned that hotels have asked for multi-destination deliveries to staff members in the hotel. Instead of returning to the front desk after a single delivery, it could visit multiple floors in one trip.

Mingling with guests

Both employees mentioned mingling with hotel guests as a use case. The business person described a kiosk mode, where the robot displays information about the hotel while not in use, and has lightweight interactions with guests. One hotel had expressed interest in having the robot do something similar in the breakfast area in the morning. Another hotel asked if the robot could go to different locations in the lobby, displaying information at each location. In these examples, the robot is also advertising itself, potentially leading customers to ask for a delivery later.

Information gathering

A third type of capability that was asked for revolved around information gathering. For example, some Savioke employees discussed the idea of detecting food trays in the hallways, and reporting the locations to the hotel staff. Another example was to ask people in the breakfast area to rate their stays. Reporting bad stays would be useful for hotels, to avoid getting a bad review online.

Other uses

Savioke employees informally discussed internal programs they wanted the robot to run. These included running programs to test new software, and creating custom behaviors for sales demos. In Section 7.4.4, we give two case studies of our system being used for these purposes.

Additionally, as discussed in Section 7.5.4 below, researchers can use the system to study the use of mobile robots in other settings such as retirement communities. Such research requires a rapid prototyping system to create a large space of candidate designs [25], before the designs are narrowed down and detailed.

7.3 User study

We evaluated our system by conducting an observational study with non-software engineering employees at Savioke, and experienced programmers outside of Savioke. The goal of the study was to understand how well each group could use our system to make real-world programs. We also wanted to see whether users thought the system was expressive enough for the uses they had in mind.

7.3.1 Procedure

Participants were seated at a computer in view of the robot they would program. The participants were first introduced to the robot with a paragraph-length description and short video. Then, they completed a 45-minute tutorial on how to use *CustomPrograms*. In the tutorial, the users made

an application that sent the robot to a room, and a program that asked whether to go to the "Front Desk" or to the dock (Figure 7.1).

The participants then created three programs with *CustomPrograms*, one program at a time. These programs, shown in Table 7.4, were based on the real-world use cases discussed in Section 7.2. The robot was configured with locations including room numbers, a "Front Desk", and some poses in the nearby area representing the lobby. Some participants (those in Group 2, see Section 7.3.3) were allowed to optionally create a fourth program, of their own choosing, if time allowed. All participants were asked additional questions at the end of the study. Finally, we also collected demographics such as gender, age, and prior programming experience.

7.3.2 Measures

Ease of use

An objective measure of the system's ease of use was whether or not users were able to create the three programs correctly. For each program, we determined if the program correctly matched the program description we gave. Because some errors are more serious than others, we labeled the errors participants made, and classified them as either major or minor. Although the labels could be subjective, in practice, most of the errors made were easy to spot and distinguishable from one another. We considered minor errors to be easy to fix errors such as a missing statement or errors that could be fixed by changing a constant value. An example of a minor error might be displaying the wrong text on screen. An example of a major error might be using an if statement to repeatedly check a condition, instead of a *while* loop.

We also gathered subjective measures for ease of use. After completing each program, users were asked to rate the easiness of making the program on a 7 point Likert scale, and to describe what the most challenging part of making it was.

#	Prompt
1	**Goal**: The robot drives around the lobby, stopping at certain places to explain to guests what it does. **Program behavior**: The robot should go to "Lobby 1" and "Lobby 2" over and over again. At each place it visits, it should stop and show these three messages for 10 seconds each: "I'm Relay, a delivery robot", "Need something? Call the front desk and I'll bring it up!", "Have a great day!" While the robot is travelling, it should say, "Excuse me, on my way".
2	**Goal**: Pick up items from a guest room. **Program behavior**: When the program starts, the robot should go to the front desk, and ask which room number to go to. The robot should be able to go to any room number it knows about. The robot should go to the room and open its bin. It should say "Please load your items" and present a "Done" button. Once the guest touches the "Done" button, the robot should close its bin and go back to the front desk. After that, the program should end. When the robot is out on a pickup it should say "On a pickup" on its screen. When it's returning to the front desk, it should say "Returning home" on its screen.
3	**Goal**: Stand in the lobby, going back to dock if it needs to charge. **Program behavior**: The robot should go to "Lobby 1" and say "Welcome to the hotel!" for 60 seconds on its screen. After showing the message for 60 seconds, it should check if its battery level is at least 50%. If so, it should show the message for 60 seconds again. If the battery is below 50%, the robot should go to the dock. While in the dock, it should say "Welcome to the hotel!" for 30 seconds, and "I'm charging up" for 30 seconds, until its battery is full. Once the battery is full, it should go back to the "Lobby 1" and repeat.

Table 7.4: The prompts describing each of the 3 programs to make in the user study.

Expressiveness

We considered two measures of expressiveness for this study. After being introduced to the robot, but before being exposed to the programming interface, users were asked to think of a specific task the robot could do other than hotel delivery. They were asked to describe these programs in a step-by-step fashion. We analyzed these programs to determine whether *CustomPrograms* could be used to make them. Additionally, after making all three programs with the interface, we asked users to rate their agreement with the statement, "Any task the robot could possibly do, without any hardware modifications, can be programmed using this interface" on a 5 point Likert scale.

7.3.3 Participants

We conducted the study with two groups of participants. The first group of participants, referred to as Group 1, were employees at Savioke who were not in software engineering roles. This group represented an important group of potential users–designers, sales people, and engineers who would want to prototype programs for hotels or for internal uses. To study the system with programmers outside of Savioke, we recruited an additional group, Group 2, from amateur robotics email lists targeted at the San Francisco Bay Area. This group represents another group of early users–researchers or other programmers who might use our system outside of hotels. Participants in the second group were required to have at least 2 years of programming experience, and were offered a $50 Amazon.com gift card for their participation.

There were 18 participants, 9 in each group. We asked all participants to rate their prior programming experience on a scale of 1 to 5, with 1 being no prior experience and 5 being professional level of experience. All of the programmers in Group 2 rated their prior experience a 5, while Group 1 had less experience (M = 2.67, SD = 1), ranging from 1 to 4. We consider *experienced* programmers to be any participant who rated their prior experience a 4 or above. Based on this definition, 2 of the users in Group 1 were experienced, and 7 were not. The Group 2 programmers were asked to approximate their years of programming experience, which ranged from 6 - 50 years (M = 20.3, SD = 13.6). Group 1 included 7 males and 2 females with ages ranging from 23 to 64

(M = 36.1, SD = 13.5); the Group 2 programmers were all male with ages ranging from 24 to 73 (M = 48.7, SD = 16.4).

7.4 Results

There are three parts to the evaluation of *CustomPrograms*. First, we examine its ease of use, using the data from our user study. Second, we analyze its expressivity, based on use cases from Section 7.2 and from the user study. Finally, we present two case studies showing how the system was used in practice.

7.4.1 Ease of use

In the user study, Group 1 made 46% of the programs with only minor errors, compared to 77% by Group 2, shown in Table 7.5. The subjective ease of making each program is shown in Table 7.6.

Program	**All users**	**Group 1**	**Group 2**
All programs	0.62	0.46	0.77
Program 1	0.71	0.56	0.88
Program 2	0.5	0.33	0.67
Program 3	0.65	0.5	0.78

Table 7.5: The rate of programs made with only minor errors.

When asked, two Group 2 programmers said they would definitely prefer a graphical programming interface, and one said they would definitely prefer a text programming interface. The remaining gave a mixed response, saying that the graphical interface would be better for people with less experience, or for one-off tasks, while a text interface would be better for complex programs or day-to-day use.

Program	All users	Group 1	Group 2
All programs	5.14	4.73	5.58
Program 1	5.71	5.44	6.00
Program 2	4.76	4.22	5.38
Program 3	4.94	4.50	5.38

Table 7.6: The perceived ease of making each program, on a 7 point Likert scale.

7.4.2 Error analysis

Across all three programs, we found that Group 1, which had less experienced programmers, had a harder time making programs than Group 2. The errors they made were mostly related to programming concepts like infinite loops, string concatenation, and programming logic.

Program 1 required an infinite loop to repeat the program forever. However, 3 programmers, all from Group 1, used a loop that repeated a finite number of times. One way to alleviate this issue could be to have a *repeat forever* block. In our system, programmers had to make infinite loops with *while-true*, which is not obvious for new programmers.

In Program 2, the robot was supposed to ask for a room number, and go there using string concatenation, as in, **robot.goTo**("Room " + number). Although making the robot go to a room number was part of the tutorial, users may not have fully absorbed how to do it. Instead, users avoided that part by presenting a limited subset of rooms as a multiple choice question, or by going to lobby locations rather than rooms. Two Group 1 users who used string concatenation did so incorrectly, e.g., by omitting a space after the word "Room".

Program 3 required programmers to continuously monitor the robot's battery state using *while* loops. These loops needed to be nested inside another *while* loop, to repeat the program forever. 4 of the Group 1 users used an *if* statement at the top level to check the battery level, rather than a *while* loop (Figure 7.3). 4 users used an incorrect primitive to make the robot dock. We had a

Program 1 errors	**Type**	**G1**	**G2**
Did not loop forever	Major	3	0
Did not show message when travelling	Major	1	1
Looped 5000 times, but not forever	Major	1	0
Did not show required message the first time the robot travels	Minor	0	3
Did not show correct message when travelling	Minor	1	1

Table 7.7: # of people making each error in Program 1.

goToDock primitive specifically for this purpose; however, these users used the **goTo** primitive, and made the robot go to a location named "PreDockingPose," which was a location in front of the dock.

Overall, even the major errors made were not too serious, and could be avoided with just a small additional investment of development time or training.

7.4.3 Expressiveness

In this section, we characterize our system's ability to create applications envisioned by Savioke employees (Section 7.2) and by user study participants. We consider 4 categories of applications, based on how much effort it would take to implement: those that can be done, those that need minor software changes, those that need major new software capabilities, and those that require hardware modifications.

Savioke use cases

Our system implements all the primitives necessary to do a normal hotel delivery, so small wording changes can be made easily. Visiting multiple locations on one trip is also just a small modification

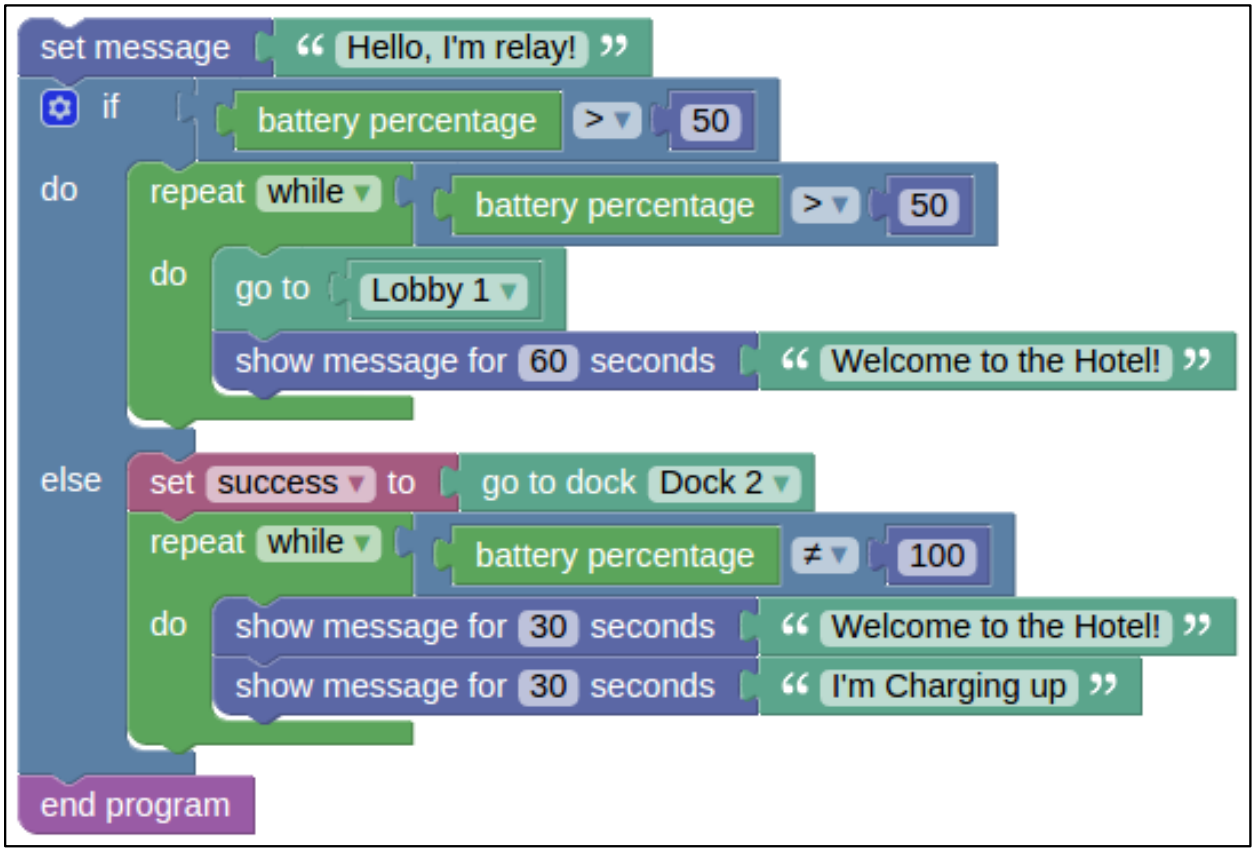


Figure 7.3: A Group 1 participant's version of Program 3. The participant used *while* loops to wait for the battery to charge and discharge, but used an *if* statement at the top level instead of an infinite loop.

Program 2 errors	**Type**	**G1**	**G2**
Could only go to a subset of possible rooms	Major	2	1
Could only go to locations other than rooms	Major	2	1
Did not concatenate `Room` with room number correctly	Major	2	0
Did not go to front desk after pickup	Major	0	1
Did not show correct message when travelling	Minor	1	2
Did not show any message when travelling	Minor	2	0

Table 7.8: # of people making each error in Program 2.

of normal delivery.

Our system can display messages to guests while mingling. It can also interact with guests and gather information by asking for star ratings, ratings on a numeric scale, or multiple choice questions. The responses can be stored in variables or lists. However, the robot can only communicate the answers by displaying the information on screen. Other ways, such as email, would need minor software development.

One behavior our system does not support is scheduling the robot to run parts of the programs at certain times of day. For example, the hotel would not be able to have the robot automatically go to the breakfast area at 6 AM every day. We chose not to implement this because we wanted to keep hotel staff in control of the robot's availability. Making this feature available would require minor software development.

Our system does not include any perception-related primitives, such as a hotel tray detector. We consider this to be major development work, as many of these perception tasks are still active areas of research. However, as perception primitives are created, they can be added to the interface as blocks.

Program 3 errors	**Type**	**G1**	**G2**
Did not use correct block to go to dock	Major	2	2
Did not continuously check battery before docking	Major	3	0
Did not continuously check battery before undocking	Major	1	0
Did correct checks, but did not loop program forever	Major	1	0
Continuously checked for wrong battery level before undocking	Minor	1	3
Displayed message for wrong amount of time	Minor	1	2
Did not display correct message(s) on screen	Minor	1	1
Continuously checked for wrong battery level before docking	Minor	0	2

Table 7.9: # of people making each error in Program 3.

User study programs

Participants in the user study came up with new use cases for the robot in 11 unique settings, before they had seen the interface. The most common setting, envisioned by 7 participants, was in an office building. Other settings included homes, retirement communities, hospitals, warehouses, and restaurants. Most programs were variations on delivery, such as delivering medication in a retirement community on a schedule, or delivering packages in an office. A few applications were more novel, such as checking if the front door was locked, or using the robot as a shopping cart.

8 of the 18 applications were doable immediately with our system. 5 applications could be done with minor software development. 2 applications required major new software capabilities the robot did not have, such as using computer vision to check if the front door of a house was locked. 4 of the applications required hardware modifications to the robot, such as readers for office badges, RFID, and credit cards.

Of the 5 applications that needed short-term development, 2 of them needed the ability to input text through a software keyboard, 2 of them required the ability to schedule tasks for the future, and 1 required the ability to call phones.

For the optional 4th program, one experienced programmer tried to make the robot drive in a Fibonacci spiral, and said that it would be interesting to use the robot to teach math.

After trying the interface, users were asked to rate, on a scale of 1 to 5, whether *CustomPrograms* could program any task the robot could do (barring hardware modifications). Users gave a slightly positive response (M = 3.28, SD = 1.27) to this question.

7.4.4 Case studies

Intel Developer Forum 2015

One of the motivating use cases for *CustomPrograms* was to customize the robot's behaviors for trade shows and sales demos. In the summer of 2015, *CustomPrograms* was deployed to 5 Relay robots at the Intel Developer Forum, a large technology conference. The day before the conference, we developed and tested a demo application in about an hour.

In our initial designs, all user input primitives waited indefinitely for input. However, we realized that even at a crowded conference, there might not always be someone around to interact with the robot. As a result, we added an optional timeout to all user input primitives, and updated our program. If a timeout was set and no user input was provided, then the primitives returned `null`, which the program handled.

At the conference, the robot's bin was loaded with snacks and water bottles. The robots then drove around to several locations around its area, and offered a snack to anyone nearby. If no one asked for a snack, the robot timed out and went to a different location. The robot also told people to visit a booth at the conference. One limitation of the program was the inability to customize the navigation behavior. Groups of people often crowded around the robot, which made the robot navigate in circles, looking for a path out.

Internal testing

When *CustomPrograms* was deployed to robots at Savioke, one technical business person wrote a program for stress testing the robot. The program drove the robot around the office continuously, stopping only to charge its battery. During the first few attempts, employees noticed that the robot behaved oddly after driving for several hours. After investigating the issue, they discovered that the robot was having difficulty storing log data after driving for a long period of time. This issue had not been encountered before in the field because the robot did not drive as frequently and as long in normal use. This experience helped guide the development of changes to fix this issue. Eventually, *CustomPrograms* was able to run on a robot for over 24 hours continuously.

7.5 Discussion

Our user study results showed that users could make real-world programs for the Relay robot, with programming errors that are easy to correct. Most of the differences between the two groups can be explained by prior programming experience. The first group had less programming experience, and made basic programming errors like not properly concatenating a number to a string. Group 2 programmers had some issues with the idiosyncrasies of our interface. For example, in Program 3, just as many Group 2 users did not use the **goToDock** primitive to go to the dock as Group 1 users.

During the study, participants had limited time (approximately 90 minutes) to learn the system and create three programs. We envision that users interested in using the interface would spend more time learning and using the interface. As shown in Section 7.4.4, our system enables new public-facing applications to be developed and tested in a matter of hours.

We believe that the interface will scale to meet increasing needs over time. Because we use a general-purpose programming language, users will have expressive ways to manage complexity. For example, some of the experienced programmers in our study organized their code with functions, although we never explained how. Additionally, we can add more primitives to our system at any time, and organize them in an ever-growing standard library.

7.5.1 Revision implications

Additional development of the system would make it more usable. In Section 7.2, we discussed missing capabilities that could be useful, like scheduling programs to run in the future, perception primitives, and sending emails. Other missing primitives included text input and the ability to call phones. We also would want to refine the blocks and user interface to address the issues we encountered in the user study.

7.5.2 Design implications

Blockly incorporates a general-purpose programming language, so it is not surprising that experienced programmers had an easier time with the interface. While it eliminates the need to remember syntax, some researchers have argued that software engineering is inherently difficult [21]. As a result, our system may never be fully intuitive for non-technical users. However, an area for future research is to understand whether we could adopt a "copy and paste literacy" model of end-user programming [105]. In this model, technically skilled users would create and distribute programs, while less experienced users could modify and customize them [85].

7.5.3 Limitations

CustomPrograms is built on top of Blockly, which was designed with educational purposes in mind. In its current form, Blockly generates global variables for all variables. This naturally limits the maximum complexity of the program. The scale of a program is also limited by the fact that code cannot be shared between two programs.

In our study, we did not incentivize users to make correct programs. This may have led to programmers not testing their code or not carefully reading the program specifications. For example, in Program 1, one of the experienced programmers in Group 1 repeated the program 5000 times instead of making an infinite loop. Most likely, the programmer was familiar with infinite loops, but they either did not read the program specification carefully enough, or they did not want to spend the time to make the loop correctly.

7.5.4 Other uses

CustomPrograms has been used for other uses as well. Below, we describe three uses of *CustomPrograms* outside of our research: socially interactive applications, elder care, and computer science outreach. Additionally, Chapter 9 discusses how *CustomPrograms* was used in the *Code3* system for mobile manipulator robots.

Socially interactive applications

Researchers affiliated with Savioke added additional primitives tailored for socially interactive use cases [35]. The first of these was **findPeople**, which returned a list of locations of people detected by the robot's camera. This enabled applications in which the robot actively sought out people, rather than waiting for a person to interact with it. The second was **goToUntil**, which made the robot navigate to a location, stopping if someone touched the touchscreen. This allowed programmers to create more natural navigation behaviors. Without this primitive, the robot could only navigate to known destinations, and it would ignore people's attempt to interact with it until reached its destination. The researchers created two socially interactive applications and conducted field trials at five hotels [35]. In 2018, Savioke deployed one of the socially interactive applications that the researchers had prototyped and experimented with using *CustomPrograms* to their hotel customers [116].

Elder care

We worked with researchers at the University of Pennsylvania who used *CustomPrograms* to study the use of robots in an elder care setting [93]. They prototyped two applications that could be used in an elder care facility: water delivery and walking encouragement. In the water delivery task, the researchers programmed the robot to visit an older adult's room, remind them of the importance of staying hydrated, and offered a bottle of water. The robot also asked to schedule another time to do another water delivery in the future. In the walking encouragement task, the robot visited the room and asked the resident to take a walk with it. At the end of both tasks, the

robot conducted a pain assessment by asking the residents to rate their level of pain. These results were then reported to observers of the experiment. The researchers implemented these interactions using *CustomPrograms* and were able to quickly change the wording of the dialogue and the flow of the interaction based on design feedback from other researchers.

There were a couple of areas where *CustomPrograms* could be made more expressive in this scenario. First, when doing the walking task, the robot needed to ensure that the older adult was following the robot, and the robot needed to adjust its pace to match that of the older adult's. We did not have primitives for performing this autonomously, so the researchers teleoperated the robot during the walking portion of the task. Additionally, for interacting with older adults, the system needed customized displays on its touchscreen interface that had larger text and buttons. The display is an important aspect of a robot's behavior, so giving programmers the ability to create their own designs for the screen is another area for future work.

7.6 Application to computer science outreach

Robots are popular tools for computer science (CS) outreach efforts that introduce programming to K-12 students. Examples include Lego Mindstorms[3], Thymio[4], and Dash and Dot[5]. Having students program robots can make computing and engineering disciplines more appealing to students because robot programming results in physical, tangible feedback [40, 60, 71, 77, 88, 100, 110, 132].

Previous CS outreach programs have been limited to using "toy" robots that are small and that cannot perform useful activities (*e.g.*, autonomous indoor delivery of everyday objects). A possibly more engaging approach is to allow students to program human-scale robots, such as the Turtlebot described in Section 1.3. The Turtlebot is capable of delivering items, interacting with end-users on a touch screen display, and autonomously navigating the environment. Allowing students to program useful robot tasks could help convey how CS can have a positive impact and

[3] `http://www.lego.com/en-us/mindstorms`

[4] `https://www.thymio.org/home-en:home`

[5] `https://www.makewonder.com/`

encourage students to pursue careers in CS. In particular, our target audience includes students with disabilities, who could be interested in programming robot tasks to assist themselves.

One reason why "toy" robots are more prevalent in CS outreach could be cost. However, the Turtlebot is, arguably, a low-cost platform designed for robotics education[6]. Another reason is that the programming barrier to entry is higher for robots like the Turtlebot. As we have discussed previously, programming robots like the Turtlebot requires significant background knowledge, including knowledge of Python, C++, and ROS, whereas "toy" robots often come equipped with simplified programming interfaces.

In this work, we explore the use of *CustomPrograms* in the context of CS outreach with human-scale, functional robots. We first describe the design of the system as applied to the Turtlebot. We then share our experiences from two offerings of a week-long introductory programming workshop. In these workshops, high-school students with various disabilities learned to program a Turtlebot using *CustomPrograms*. We found that the robot and *CustomPrograms* were effective at allowing novice programmers to realize their ideas on a physical robot, increase interest in CS, and establish confidence about their ability to pursue CS. We present key observations from the workshops, lessons learned, and suggestions for readers interested in employing a similar approach.

7.6.1 Accessible Robot Programming

In this section, we describe how we adapted the Turtlebot and *CustomPrograms* for use in our workshops with students with disabilities.

Turtlebot platform

In our research, we used the Clearpath Turtlebot 2, described in Section 1.3. The Turtlebot can be reconfigured with platforms at different heights and other components. We allowed students to specify minor hardware modifications to the robot, such as adding a basket, box, brush, or other

[6] As of 2018, a Turtlebot costs $1,900 or $1,450 without an included laptop. While more expensive than robot kits such as Lego Mindstorms ($350), it is less than the starting costs of other robotics outreach programs like the FIRST Robotics competition ($5000 - $6000).

Figure 7.4: Chester, a waiter robot programmed by students at the 2015 Robot Programming Workshop, (a) escorts guests, (b) takes orders, and (c) delivers orders at "Cyber Cafe". Robots programmed by individual students at the 2016 workshop include (d) a cleaning robot (FRED), (e) a medication reminder/delivery robot (Ultravax), (f) a language tutor robot that names objects in the room (Lærer), (g) a math tutor robot (Khan), (h) a pet robot that talks about sports (Sport), and (i) a dog walking robot (Walker).

tool onto the robot. Our Turtlebots held tablets which served to display information and receive input in the form of button presses. In addition, they were capable of speech output. Given a map of its environment, the Turtlebot can localize itself and navigate between any two points on the map while avoiding obstacles. Locations on a map can be given names for easy referencing. Prior to the workshop, we used the Turtlebot's navigation tools to create a map of the room and define names for useful locations, such as "Door" and "Table."

7.6.2 CustomPrograms modifications

To make *CustomPrograms* more accessible for students with motor skill impairments and low vision, we modified CodeIt to present a text-based interface, which shown in Figure 7.5, instead of the block-based interface described earlier in this chapter. In our first workshop in July 2015, we had students write Python code on the online code hosting site, Github[7]. An instructor manually executed the code on the robot. In the second version of the workshop, held in July 2016, we had students program in JavaScript using the browser-based interface shown in Figure 7.5, which allowed students to create, save, and execute *CustomPrograms* programs directly.

In both cases, the robot primitives were the same:

- **goTo(location):** Navigates the robot to the named location, avoiding obstacles.
- **say(text):** Uses voice synthesis to say the given text out loud.
- **displayMessage(message):** Displays the given message on the robot's touch screen.
- **askMultipleChoice(message, choices, timeout):** Displays the given message and choices on the screen. Returns the choice selected. If the optional timeout parameter is specified, the robot will stop waiting after the timeout if no user input is provided.
- **moveForward(speed), moveBack(speed):** Moves the robot forward or backward at the specified speed (m/s) for one second.

[7] `https://github.com`

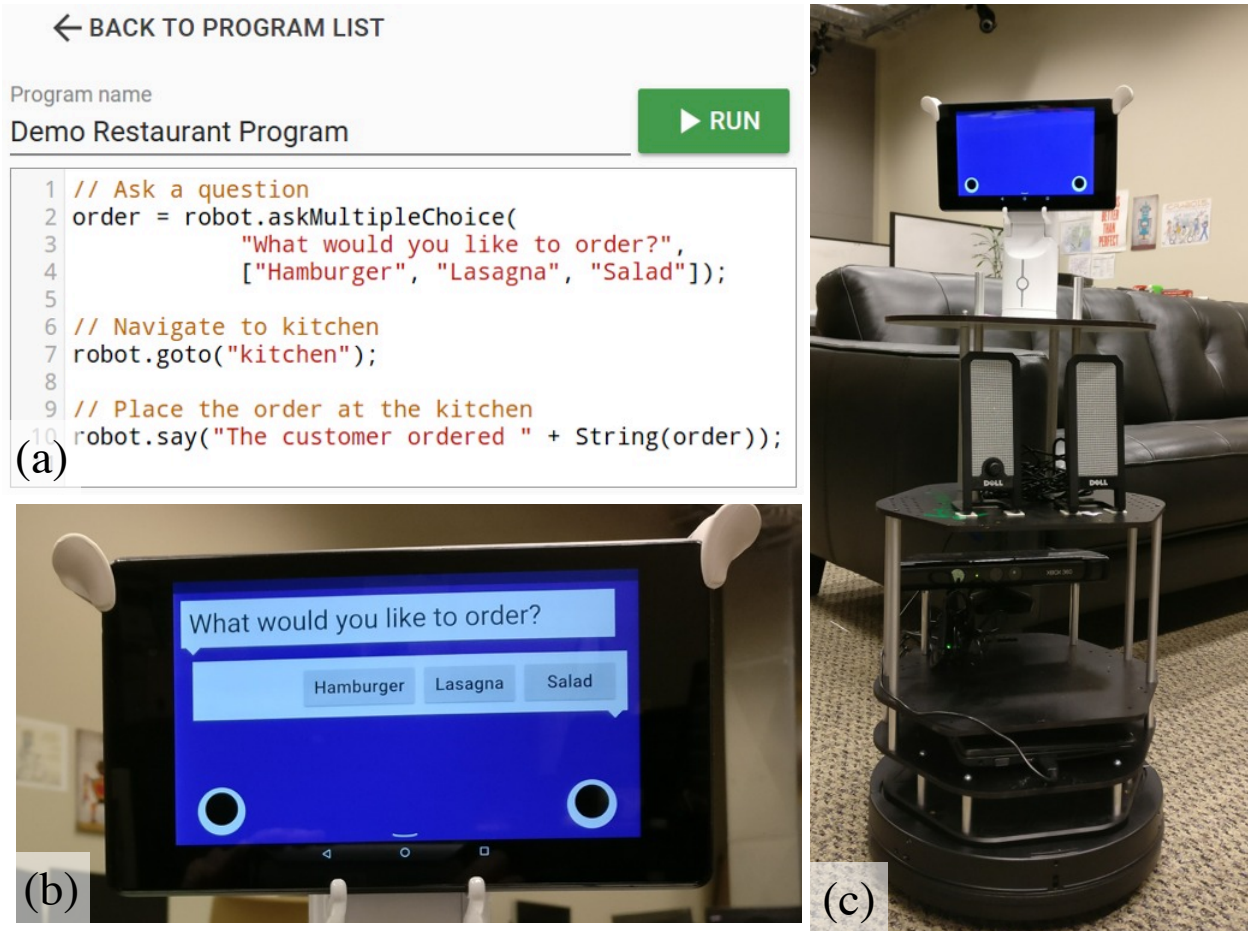


Figure 7.5: (a) The text-based *CustomPrograms* interface showing a sample program, (b) the robot's screen during the operation of the sample program, (c) the Turtlebot.

- **turnLeft(speed), turnRight(speed):** Turns the robot left or right for one second at the specified rate (rad/sec).

Accessibility and ease-of-use were key requirements for the system because we wanted it to be used by students with disabilities. Implementing our interfaces as browser-based web applications allowed students to program on their own laptop computers and ensured they could use pre-installed accessibility tools such as screen-readers, special fonts, and zoom features.

7.6.3 Robot programming workshops

The robot programming workshop was offered as part of the DO-IT Scholars summer program[8] in July 2015 and 2016. The DO-IT Scholars summer program engages students with disabilities from the Seattle area with the goal of encouraging them to go to college and facilitating their transition. Besides workshops like ours, the summer program includes activities such as self-advocacy skill building and field trips to local technology companies.

Both of our workshops took place over five days, three hours each day, for a total of fifteen hours of instruction. The 2015 workshop had five students, while the 2016 workshop had six. In 2015, all students worked together to program a single robot functionality. In 2016, each student developed their own program. Both years, students presented their programs to the entire group at the end.

The primary goal of the robot programming workshop was to interest the students in CS as a field and to develop their confidence that programming could be a fun and accessible activity. The secondary goal of the workshop was to develop introductory knowledge of CS concepts such as functions, loops, and parameters.

Curriculum

The outline of the five-day workshop was as follows.

- **Day 1:** We introduced basic concepts in robotics. Pairs of students brainstormed dozens of program ideas for what to program on the robot and began exploring CodeIt.

- **Day 2:** Students finalized project ideas and began implementation and low-fidelity hardware modifications.

- **Day 3:** Students tested and debugged their programs, and implemented improvements.

[8] http://www.washington.edu/doit/programs/do-it-scholars/overview

- **Day 4:** Students fixed final bugs and demonstrated their robots. We conducted interviews with the students.

- **Day 5:** We demonstrated other robots and performed activities. We administered a post-workshop survey.

Teaching the API

Learning from our past experience and past work by others [77], we pursued a teaching style that minimized lecturing and maximized hands-on time with the robots and code. We decided to teach the API via sample code. On the first day of the workshop, we provided a sample program which demonstrated the use of the most common robot functions. We included detailed comments explaining each function's purpose. A portion of the 2016 CodeIt JavaScript sample code is given in Figure 7.6. We asked students to make a copy of this program, modify it in any way they would like, and test it on the real robot.

Over the course of the next few days, students learned the API while developing their program. We introduced conditional statements to the students when prompted, and introduced for-loops and while-loops on Day 3 to students who required them for their programs.

Participants

Of our eleven students, three were female, and eight were male. Most were juniors and seniors in Seattle area high schools. One student had taken AP Computer Science and participated in FIRST robotics. Two had taken one or two general computing classes. One had learned HTML/CSS to create a website. The rest of the students had no prior programming experience. Our students had various disabilities or conditions including deafness, low vision, cerebral palsy, muscular dystrophy, Ollier disease, attention-deficit disorder, Asperger syndrome, and other autism spectrum disorders or learning disabilities.

Figure 7.6: A code sample from the 2016 DO-IT programming workshop.

```
// Portion of 2016 Sample Code
// go to table_southeast
robot.goTo("table_southeast");
// other places: door, home, trash, couch

// display message
robot.displayMessage("Hello!", "My Name is LoggerHead.");

// say message
robot.say("Hello, my name is Loggerhead.");

// wait for two seconds
waitForDuration(2);

// say message
robot.say("What's your favorite color?");

// Ask a question
item = robot.askMultipleChoice("What's your favorite color?",
   ["Indigo", "Jade"]);

// Display a message by adding two strings together
robot.displayMessage(String("You Selected ") + String(item));

// say message
robot.say(String("You Selected ") + String(item));
```

Projects

The 2015 workshop involved collaborative programming. The five students were split into three teams of two, with one student working with a teaching assistant. Each of those teams worked on a small part of a larger program which played out the scenario of Chester, a robot that serves people at a restaurant. The first team worked on greeting and escort to table, the second team worked on taking customers' orders and delivering food, and the third team managed the receipt and payment.

The 2016 workshop involved students working on individual projects. Two robots were shared between the 6 students. The different projects were as follows:

- **French Robo Expunging Device (FRED):** A cleaning robot that dusts tables, cleans white-boards, and sweeps objects on the ground.
- **Ultravax:** A robot with a sassy personality that delivers medicine to a caretaker at the proper time.
- **Lrer:** A robot that teaches Norwegian by navigating around a room, translating the names of various objects it comes across and administering a quiz at the end.
- **Khan:** A robot that teaches mathematics and administers quizzes.
- **Sport:** A robot that acts as a pet for people who have allergies and also knows a lot of sports facts.
- **Walker:** A robot that can take a dog for a walk, for busy people with pets.

These projects are illustrated in Figure 7.4.

7.6.4 Data, observations, and findings

Throughout both workshops, we took notes of our interactions with students as they worked on their projects, and we recorded video-blog style interviews with each of them on the last work

day of the workshop. In these brief, open-ended video interviews, we had the students describe their projects and asked them questions about their work. In addition, in 2016 we included a post-workshop questionnaire and received responses from a short interview conducted by an external supervisor on the last day. Below, we present our findings based on the compiled data.

Collaborative vs. individual programming

In 2016, as students programmed their own robots, we saw the students engaged with the robot more and felt more of a personal connection with the project, compared to 2015. Multiple students referred to the Turtlebot they worked on as "my robot" instead of "the robot" in 2016 compared to 2015. In our post-workshop survey, a student mentioned that it was better to let each student work on their own program, saying, "not everyone wants to do the same thing."

On the other hand, collaborative programming could still be useful to give students teamwork experience in a CS environment. For example, the robot waiter program developed by the teams in 2015 was larger in scope than any of the individual projects done in 2016.

Post-workshop survey

At the end of the 2016 workshop, students filled out an eight-question survey (Figure 7.7) that evaluated the effectiveness of the workshop. Next, we discuss the most interesting results of the survey which all six students responded to.

In response to question 2, four out of six students agreed that the "best" part of the workshop was being able to program whatever they wanted on the robots. In question 6, two students rated the ease of learning as 4/5 and the rest as 5/5, indicating high levels of satisfaction with how simple *CustomPrograms* was to learn. This was an expected result, since *CustomPrograms* was designed to have a low barrier of entry. One student added to their response that the reason they responded with a 4 was because at times the student needed help understanding some of the concepts.

In question 7, three students rated the ease of implementing their ideas as 4/5 and three as 5/5. One student mentioned that they did not respond with a 5 because of technical challenges with

Figure 7.7: 2016 Post-Workshop Survey Questions

1. Before participating in this workshop, what experience did you have with computer science or robotics, if any?
2. What parts of the workshop did you like the best? Why?
3. What parts of the workshop could use improvement?
4. What part of the workshop was most beneficial to your learning?
5. What part of the workshop was most distracting or hindered your learning?
6. The Web-based CodeIt programming tool was easy to learn. [1-5]
7. I found that CodeIt allowed me to easily turn my ideas into a functional robot. [1-5]
8. I would like to use CodeIt to program robots in the future. [1-5]

the infrastructure encountered occasionally during the workshop. These responses demonstrate that students perceived *CustomPrograms* to be expressive enough to program useful tasks. It also indicates that our introduction of the robot and the robot API correctly communicated the robot's capabilities.

In response to question 8, three students rated their desire to use *CustomPrograms* in the future as 5/5, two as 4/5 and one as 2/5 response. The student who responded with a 2 wrote that while *CustomPrograms* was "very easy" to learn, its commands would not be versatile enough for a more complex robot.

Perception about programming

Coming in to the workshop, many students believed that programming would be difficult, but, after the workshop, found it easier than they imagined. In a video interview, a student from the 2015 workshop reflected on her week, mentioning that at first, the sample code "looked really intimidating," but after working with it for a while, she learned "how any small change we made to the code affected how the robot behaved." This student came back as an instructor in our 2016

workshop.

A 2016 student remarked in interactions with instructors on multiple occasions, “technology doesn’t like me,” expressing her discomfort and lack of confidence when working with computers. However, at the end of the workshop, she mentioned in her video interview that learning to program “was easier than [she] thought it would be,” adding that programming “came easily to [her].” This sentiment was echoed by a 2015 student who remarked in his video interview that during the workshop he “grew to understand programming.”

Another 2016 student, who created the Norwegian-teaching robot *Lærer* remarked in his video interview that trying to “communicate your thoughts to the robot was a pretty big learning experience,” and that “being successful at these challenging things, like telling it how to move, to teach, .. was just absolutely fantastic.”

Engagement with content

We saw high levels of engagement from students using CodeIt to create complex robot behavior. This did not necessarily involve learning new CS concepts, but rather creating detailed program content using learned concepts. For example, in 2015, students spent time adding various food items to their menu which customers could order from, and discussed in detail what the robot should say and which direction it should face when it interacts with customers at the restaurant.

In 2016, one student spent hours adding new vocabulary words to his Norwegian-teaching robot, testing and iterating to ensure his robot was pronouncing words properly as it moved from object to object. The same student worked with others to develop an entirely separate program meant to make the robot emit bizarre noises and phrases and drove it down the hallway to attempt to spook onlookers. Another 2016 student created a caretaker robot that “has personality.” He said in his video interview that he had the most fun “getting the robot to be a little sassy.” These examples show that these students had progressed past the stage of simply learning the content; they moved on to enjoying it, which is important to establishing long-term interest.

Disability in the background

By allowing students to write code on their own laptops, we were able to mitigate many accessibility challenges that would normally be present when teaching robot programming to a group of students with disabilities. Providing alternative input and output modalities on the robot allowed students to make the robot they programmed as accessible as possible to the rest of the group. For instance, most students in the 2016 workshop displayed status messages on the robot's screen while also using text-to-speech to verbalize the same message, such that both low-vision and hard-of-hearing students could be aware of the robot's status. Similarly, in the 2015 workshop the robot programmed by the students could take food orders both directly from the screen or with speech input, making it accessible for students in wheelchairs with limited reach and dexterity and students with low-vision.

Given that the workshop was run as part of the DO-IT summer program, which is centered around disability, we expected that participants would program the robot to address challenges they faced due to their disabilities. Instead, participants chose to work on developing robots that addressed general problems (*e.g.*cleaning, forgetting to take/losing medications) or problems they faced separate from their disabilities (*e.g.*studying math). Many explored their personal interests (*e.g.*pets, sports, languages).

7.6.5 Computer science concepts

The workshop introduced students to basic CS concepts. In 2015, students learned variables, conditionals, loops, events, and arrays, and they learned to write simple functions. In 2016, students learned the same concepts, except for events.

Despite the differences between the two offerings of the workshop, the kinds of mistakes made by students were similar. Most common mistakes were syntactic; such as mismatched quotation marks, brackets and braces, and incorrect use of the assignment operator (=) instead of the equality operator (==), among other similar novice programmer mistakes. Many of these mistakes could be avoided with a more advanced IDE with autocompletion and linting features.

Other mistakes had more to do with semantics. For example, some students in the 2016 workshop wanted to ask a question to the user and timeout after sixty seconds. They wrote the following code:

```
// Javascript semantic mistake
answer = robot.askMultipleChoice(
         "What is your favorite color?" ,
         ["Indigo", "Jade"]);
waitForDuration(60);
```

This code actually waits indefinitely to receive a response to a multiple choice question, and then waits for sixty seconds after receiving the response. The correct way to develop this feature uses the optional final parameter for the askMultipleChoice function:

```
// Javascript corrected
answer = robot.askMultipleChoice(
         "What is your favorite color?" ,
         ["Indigo", "Jade"],
         60); // final parameter is timeout
```

Common robot-related errors encountered both years were bugs that had to be tuned by experimentation, such as determining how much to rotate the robot to face someone in a particular seat and how long to wait in between spoken sentences for natural interaction. For example, the student who was making the robot speak in Norwegian made several iterations to achieve the correct phonetic pronunciation using the robot's English text-to-speech system.

7.6.6 Discussion

Overall, we found that individual programming created a more personalized and engaging work environment compared to a team programming project. As a result, we recommend this approach. Key resources that made this approach possible were the small ratio of students to robots (3:1), the

ability to switch between two student's programs instantaneously on the robot, the small ratio of students to instructors/teaching assistants (1:1), and a large lab space that allowed us to have separate stations for testing different robots and recording video blogs. Although the number of students could be increased (as long as other resources are also scaled), we believe that larger group activities (*e.g.*discussion about robot jobs, discussion about feasibility and usefulness of brainstormed ideas) might not be as engaging with large groups. We ran this workshop with twelfth-grade students, but believe that high school students of all ages, as well as with older middle school students, could participate as well.

One way to further scale our workshop is to enable teachers, who do not necessarily have expertise in CS, to run them in their communities. We see potential for using simulated robots that can be programmed similarly to the Turtlebot, or even to mobile manipulators like the Fetch or PR2. In future work, we would like to better understand the impact of using human-scale robots for education, as opposed to smaller, toy robots or virtual agents.

In our workshops we taught the robot API by having students edit a simple example program, as shown in Figure 7.6. Due to the small number of functions in the API, it was easier to let students learn the system by experimenting with a sample program, as opposed to a lecture that explained each function. That being said, as the API grows, we would need to select a subset of the functionality to put into a sample program, so as not to overwhelm the students with information.

Overall, we found that using *CustomPrograms* with the Turtlebot made robot programming accessible to novice programmers in an educational setting. We think that pairing *CustomPrograms* with human-scale robots can be an effective way to build students' confidence in their ability to program and to develop interest in Computer Science through relatable applications. Additionally, as more students are exposed to programming tools like *CustomPrograms* and robots like the Turtlebot, we can expect that larger numbers of people in the future will feel more comfortable applying similar programming tools to even more advanced robots like the Fetch or PR2.

Chapter 8

TRIGGER-ACTION PROGRAMMING

In this chapter, we introduce a higher-level form of task scripting that could be used for robotics programming. Trigger-action programming (TAP) is a simple programming model in which the user associates a *trigger* with an *action*, such that the action is automatically executed when the trigger event occurs. Using TAP, a user could, for example, configure the robot to perform a program developed using *CustomPrograms* at a certain time each day. As we explain below, TAP has proven to be a popular model for end-user programming of smart home and Internet of Things (IoT) devices. However, despite simplicity of the TAP paradigm, there could be subtle ways in which users make programming errors, which may be harmful when robots that perform physical motions are involved. In this chapter, we briefly turn our attention away from programming mobile manipulators directly to the domain of controlling smart home devices (which some may argue are also robotic systems). Regardless of the specific domain TAP is applied to, we are interested in examining TAP's suitability as a programming interface in general.

8.1 Background on TAP

A popular TAP interface is an online service called *If this, then that* [68] (IFTTT). IFTTT allows users to create programs that can automatically perform actions like sending alerts or changing settings of a smart home, when certain triggers occur (*e.g.*, it starts raining, someone tags the user in a picture, the user arrives at home, etc.

Figure 8.1(a) shows three examples of such IFTTT programs. With its increasing support for wearables, smartphone sensors, and other connected devices (*e.g.*, Nest Thermostats, Belkin WeMo switches, or Phillips Hue lightbulbs), IFTTT has become highly relevant for ubiquitous computing. The simplicity of IFTTT has allowed millions of everyday users to create simple

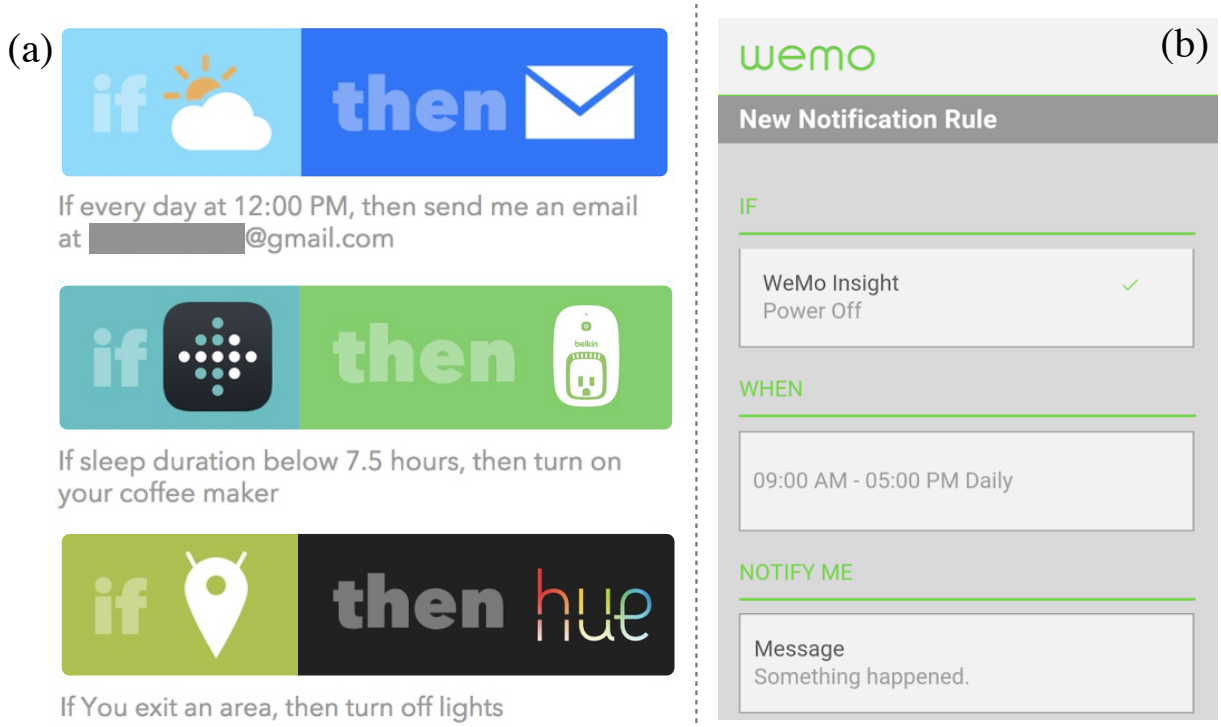


Figure 8.1: Trigger-action programming in existing products. (a) Example rules from IFTTT. (b) An example of programming a rule for a WeMo Insight Switch.

programs without requiring any specialized programming skills, hence addressing an important challenge in ubiquitous computing in the home [49].

8.1.1 Motivation

Despite its widespread and diverse use, IFTTT has an important restriction that limits its expressivity. It only allows a single event to be used as triggers in programs. As a result, it does not support rules that may require multiple triggers. For example, a user might want to receive notifications that the motion sensor in their home was activated while they are not at home; however, IFTTT does not allow this simple conjunction of triggers (user not at home and motion sensor activated) and is unable to express this rule.

Previous work on TAP has noted this limitation and proposed systems that allow conjunctions

of multiple triggers [129, 46, 59]. Although their user studies have demonstrated that people are able to use multiple triggers to create complex programs with correct behavior, none of them have investigated the accuracy of the user's mental model about how exactly those programs behave. In particular, previous work does not distinguish between conceptually different trigger types and action types. This can cause ambiguities in terms of when exactly the action would be activated and whether or not certain actions would be automatically reverted. Errors caused by such misunderstandings can have serious consequences; for instance in the context of home automation, program errors could risk home security by unlocking doors at the wrong time or cause unintended energy waste by not reverting a thermostat setting.

In the rest of this chapter, we first identify these distinct types of triggers (event and state triggers) and actions (instantaneous, extended, and sustained actions) that are used in existing systems or in previous research. Then, we present a study that reveals inconsistencies in *interpreting* the behavior of single-trigger and multiple-trigger programs involving these different types of triggers and actions. We follow up with another study that reveals errors made in *creating* similar programs. Finally, based on a characterization of these issues, we offer recommendations for improving the IFTTT interface so as to mitigate user mental model inaccuracies.

8.2 Trigger and action types

Trigger-action programs in IFTTT consist of a single rule that associates a trigger with an action. The wording "if this then that" suggests that the rules are equivalent to if-then statements found in many general purpose programming languages such as Java or Python. However, their semantics are more similar to event-driven programming [42], in which an asynchronous input signal (trigger) is handled by a callback function (action). This inaccuracy in the if-then metaphor creates some ambiguities.

One ambiguity arises from the lack of distinction between two trigger types (Figure 8.2):

- *Events* which are instantaneous signals, versus
- *States* which are boolean conditions that can be evaluated to be true or false at any time.

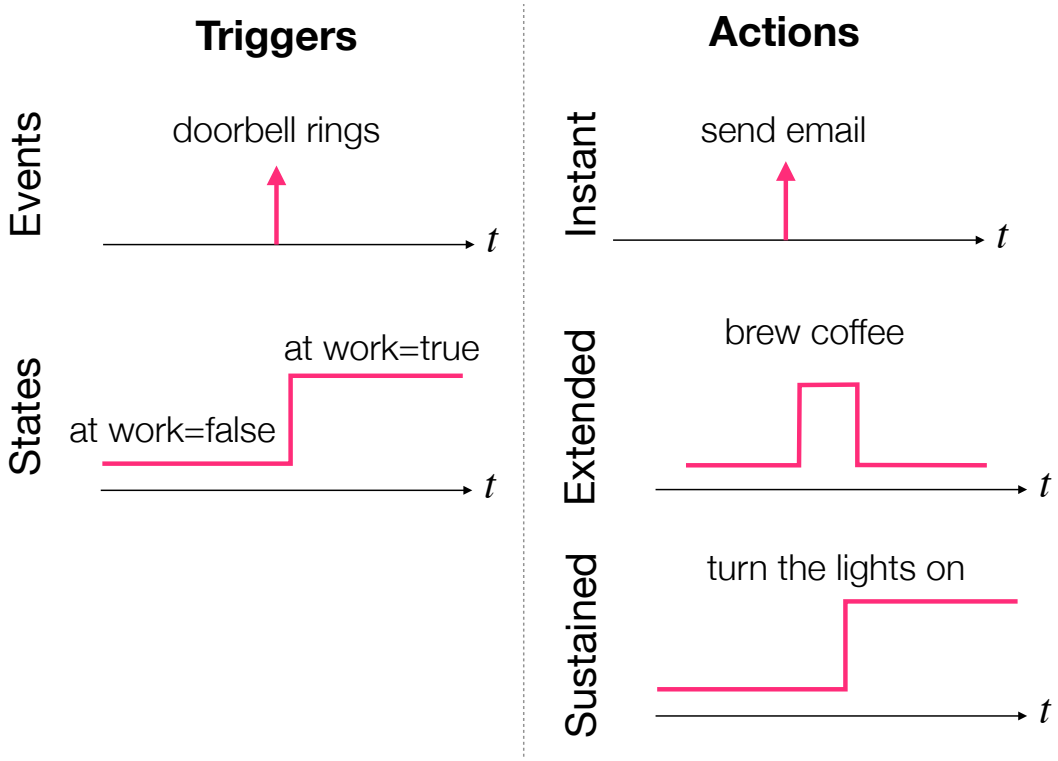


Figure 8.2: The different kinds of triggers (left) and actions (right) we distinguish between in this chapter. The horizontal axis in each diagram represents time. A raised value indicates that the trigger or action is active at that point in time.

Events indicate the occurrence of some change at a specific point in time. Examples of event triggers include "the doorbell rings," or "the temperature drops below 50° F." State triggers indicate that some condition is currently true, lasting over a period of time. Examples of state triggers could be "it is between 3:00 - 5:00 pm," or "it is raining." While regular if-then rules in general-purpose programs are meaningful with state type triggers, TAP with a single trigger is mostly meaningful with event type triggers. Indeed, IFTTT avoids state-type triggers through the use of events associated with state changes, e.g., "if the temperature *drops* below" instead of "if the temperature *is* below."

We nonetheless observe the use of state-type triggers in previous work. For example, Dey et

al.'s user study involves creating rules such as "If I am sleeping, turn the stereo off" [46]. The meaning of this rule with the state-type trigger "if I am sleeping" is ambiguous—does the rule start as soon as I fall asleep, or does the rule start any time while I am asleep? We hypothesize that most people would choose the former interpretation. The reason for this is because if I fall asleep and the rule did not activate immediately, then there would be a period of time during which the rule did not hold.

A similar distinction can be made with action types (Figure 8.2):

- Some actions are *instantaneous* and do not change the state of the system. An example is sending an alert (email or a text message). This action can be completed within one time step and the system would be ready to send another alert at the next time step.

- Other actions are *extended* in time but they are completed within a certain, ofter deterministic, amount of time. An example is brewing coffee. This action can take a few minutes during which the same action cannot be executed because the actuator is busy. However, the action completes automatically and reverts back to its original state.

- In contrast, *sustained* actions involve changing the state of an actuator, such as turning the lights on/off or setting the thermostat temperature. This new state does not revert back automatically, as with extended actions that come to a conclusion.

IFTTT allows all three types of actions (see three examples in Figure 8.1(a)) but it does not make a distinction between them. This can be problematic particularly for sustained actions, since reverting their effects requires a separate rule and users might have false expectations that they are automatically undone, particularly when paired with a state-type trigger. For instance, in the example quoted earlier "If I am sleeping, turn the stereo off" [46], the fact that there is an end to the state trigger (*i.e.*, I will wake up at some point) may imply that the action will also end—*i.e.*, the stereo will turn back on—while others would say that the stereo would not turn on unless directed to by another rule.

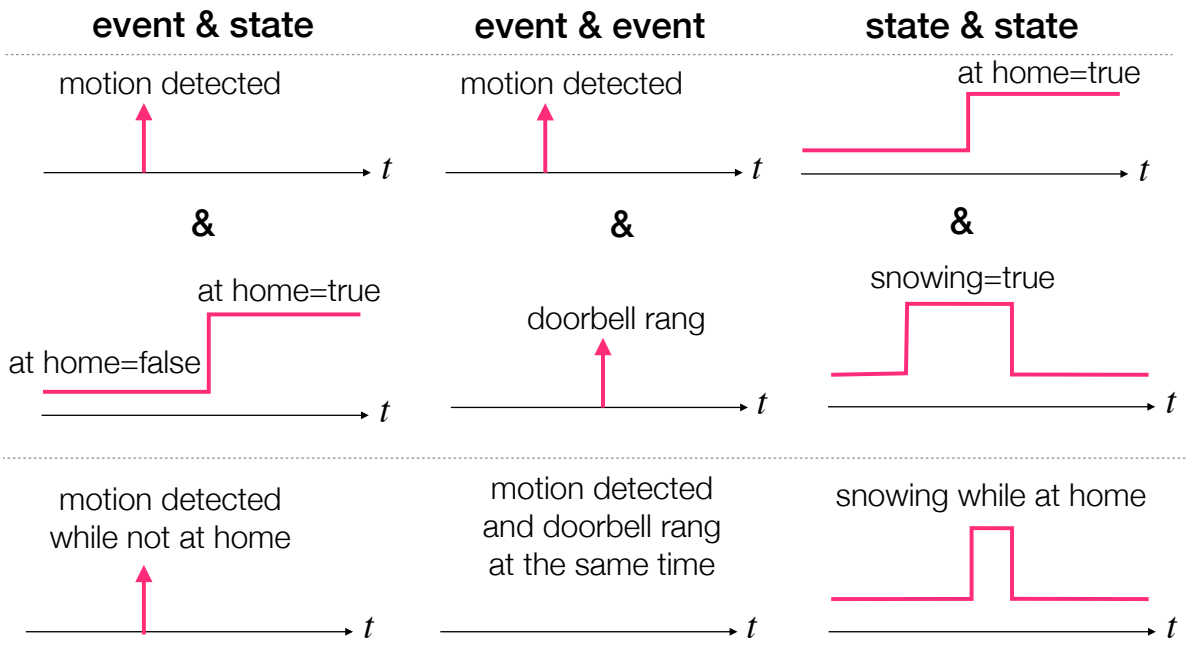


Figure 8.3: Conjunctions of different trigger types.

8.2.1 Trigger conjunctions

In IFTTT, users can only create rules with exactly one trigger. This limits the expressivity of the rules, because multiple triggers cannot be combined. For example, suppose a user wants their lights to turn on when they arrive home, but only after the sun has set. While IFTTT has both a sunset trigger and a personal location trigger, it is not possible to use them both at once to create the desired rule. This limitation has been pointed out in the literature and systems that allow conjunctions of multiple triggers have been developed [129, 46, 59]. However, the distinction between different trigger and action types in the context of multiple triggers has not been studied systematically.

Conjunctions of multiple triggers result in different types of triggers, depending on the types of constituent triggers (Figure 8.3). Conjunctions of multiple states is another state that is true when all the constituent states are true. Conjunctions of a state and an event is another event that happens at the same time as the constituent event, if the state constituent is true. For example, the rule, “If

Trigger(s)	**Logical interpretation**
Single event	Rule activates when event occurs
Single state	Rule activates once the state becomes true
One event & one or more states	Rule activates when the event occurs, but only if all the states are true
Multiple events	Unlikely that the rule will ever activate
Multiple states	Rule activates as soon as all the states are true

Table 8.1: The logical meanings we assume trigger combinations have. We expect that a majority of people will have these interpretations of trigger combinations.

the doorbell rings and it is between 3:00 - 5:00 pm," should activate when the when the doorbell rings, but only if it is between 3:00 - 5:00 pm.

A conjunction of two events is another event that happens at the same time as the constituent events, provided that all events occur exactly at the same time, which is theoretically impossible and in practice extremely unlikely. Hence a rule with a trigger like "if the doorbell rings and the sun sets" would almost never activate. Nonetheless, users might actually have an interpretation for such rules. For instance, one could interpret the example above as meaning, "If the doorbell rings *around the time* of the sunset."

8.2.2 Studying mental model ambiguities

We hypothesize that the lack of distinction between different trigger and action types is a source of ambiguity, especially in the context of trigger conjunctions. In particular, we will point out three potential points of confusion. The first is the interpretation of when exactly triggers will occur. In particular, for state-type triggers, it is unclear if people expect the action to start immediately as the state becomes true, or to happen anytime while the state is true. Second, it is unclear if conjunctions of events (which are practically invalid triggers) are actually meaningful for people. Finally, people's expectation about whether sustained actions will revert automatically is unknown.

In this work we aim to understand people's interpretations of these ambiguities when all trigger and action types are supported in the system. To that end, we performed two user studies. In the first study, participants were asked to describe their interpretation of given trigger-action programs. This study revealed a significant discrepancy in people's interpretations (low levels of agreement), where sometimes the selection of the majority did not correspond to the logical interpretation. In the second study, participants were asked to create trigger-action programs for a desired behavior and once again respond to questions related to their interpretation of a given program. This study confirmed that ambiguities are cause for errors; we observed that people created different programs given the same prompt. Furthermore, people's interpretations of given programs were still in disagreement after having created programs themselves. We describe these two studies in the following sections.

8.3 Study 1: Program Interpretation

To begin understanding how users interpret different trigger and action types, we conducted a web-based study on Amazon Mechanical Turk. In the introduction to the study, we introduced the concept of a smart home and gave a few examples of "if-then" rules which could be defined for the home. The study was split into three parts, examining different aspects of TAP.

Throughout the study, we asked participants to interpret trigger-action programs by reading a *text description* of the program and answering multiple-choice questions that assessed the partic-

Event triggers	It turns 10:00 am
	The doorbell rings
State triggers	It is raining
	It is between 3:00 - 4:00 pm
Actions	Send an email notification
	Brew a pot of coffee
	Turn the lights on

Table 8.2: The set of event triggers, state triggers, and actions used in Study 1.

ipant's understanding of the program. The descriptions we provided were similar to text descriptions automatically generated in IFTTT upon the creation of rules (see examples in Figure 8.1(a)). Event triggers were worded using the active verbs *turns* and *rings*, in line with IFTTT's way of wording events. State triggers were worded with the present tense of the verb *be*.

Throughout the study, we decided to minimize variance between the questions by using a uniform set of triggers and actions. These are shown in Table 8.2. The actions were chosen to represent 3 different kinds of actions: sending an email (instantaneous), brewing coffee (extended), and turning the lights on (sustained).

8.3.1 When actions will occur

In the first part of study, we asked users when an action would occur, given single or multiple triggers. The purpose of these questions was to evaluate whether users expected actions to occur according to the logical interpretations of the triggers (Table 8.1). There were 9 such questions: 2 single event triggers, 2 single state triggers, 3 for combinations of one event trigger and one state trigger (we excluded "If it turns 10:00 am and it is between 3:00 - 4:00 pm"), 1 which combined the two event triggers, and 1 which combined the two state triggers. In all cases, the action was

simply referred to as "[X]," *e.g.*, "If the doorbell rings, then do [X]."

For each question, the respondents were asked to choose from a multiple choice list to explain when the action would occur. For event triggers, users could say that the action would occur immediately, within 1 minute, or within 10 minutes. For state triggers, users could also say the action would occur at any time while the state was true. For questions with one event and one state trigger, the options for when the action would occur were limited to 1) when the event and state trigger became true simultaneously, or 2) when the state was already true when the event occurred.

For the question with two event triggers, the respondent could say the action occurred if the two events happened at exactly the same time, or within 1 minute of each other, or within 5 minutes of each other. For the question with two state triggers, the respondent could say the action occurred if the first state became true while the second state was already true, or vice versa, or either of those two options. They also could say that the action could occur at any time while the two states were true. For all questions, respondents could also choose to say that the action would not occur at all, or specify a free-response answer.

8.3.2 When actions will end

In the second part of the study, we asked users when an action would end, given a fully specified rule. These questions were designed to study two issues. First, did users believe that sustained actions would automatically end when paired with state triggers (*e.g.*, "If it is between 3:00 - 4:00 pm, turn the lights on")? Second, did the trigger involved in the rule lead to different expectations about when actions will end?

We gave a fully specified rule for each of the 4 triggers and 3 actions shown in Table 8.2, for a total of 12 questions. For each question, the respondents could say that the action would end within 1 minute of starting, within 10 minutes of starting, or within an hour of starting. For state triggers, respondents could say that the action would end once the trigger was no longer true. For all questions, the respondents could say that the action would never end, or they could provide a free-response answer.

An example of a question in this section is shown in Figure 8.4.

If it turns 10:00 am, then turn the lights on

Assume that the lights have turned on. If there were no other rules, when do you expect the lights to turn off?

- Within 1 minute of turning on
- Within 10 minutes of turning on
- Within an hour of turning on
- Never
- Other

Figure 8.4: An example question from the second part of Study 1.

8.3.3 Open-ended questions

Next, we asked open-ended questions. Some of the questions were designed to understand people's interpretation of the the differences between different kind of triggers. In particular, we asked respondents to list differences between "If the doorbell rings" and "If it is raining outside," as well as between "If the doorbell rings" and "If it turns 10:00 am." We also asked for different wordings of a couple of rules to see if users generally had a preferred way of wording the rule other than an if-then statement.

We asked additional open-ended questions, which were placed at the end of the study, so as not to be leading. These questions directly asked respondents for their opinions on the issues we studied through earlier questions. For example, in one question, we wrote that it would be unlikely for two event triggers to occur at exactly the same time. The users were asked if such rules should be allowed, and what their meaning should be if so. In another question, we asked when an action should occur if the trigger was a state trigger. Finally, we asked, given the rule, "If the time is

between 3:00 - 4:00 pm, then turn the lights on," if the lights should turn off automatically at 4:00 pm, or if there should be another rule to turn the lights off.

8.3.4 Demographic questions

In the last part of the study, we gathered demographic information about the respondents, including their age, gender, level of prior programming experience, and level of prior experience using IFTTT.

The study was formulated as a multi-page questionnaire using Google Forms. Participants were allowed to go back and change their answers.

8.4 Results from Study 1

8.4.1 Demographics

There were 60 respondents to the survey. We restricted the set of workers to those who had obtained the "Master Worker" distinction by Amazon and who lived in the United States. An initial set of 19 respondents were paid $0.50 to complete the survey. Because the survey took longer to complete than expected, the remaining respondents were paid $1.00 to complete the survey. 30 respondents were male and 30 respondents were female. The ages of the respondents ranged from 21 to 68 years, with an average of 39.2 and a standard deviation of 11.6. 32 respondents (53.3%) reported no prior programming experience, 19 (31.7%) said they programmed "a little," and 9 (15.0%) said they programmed "on a regular basis." 54 (90.0%) respondents said they did not know about IFTTT, 5 (8.3%) said they had heard of it, and 1 (1.7%) respondent said they had used it before.

8.4.2 Findings

Below, we summarize key findings from our first study.

Expectations about triggers depend on the specific trigger(s)

We found that respondents had different expectations for when actions should be triggered depending on whether the trigger was an event trigger or a state trigger. Across the two rules that included a single event trigger, 75.8% of responses said the rule would start as soon as the event occurred. But, across the two rules with a single state trigger, only 36.7% of responses said the rule would start as soon as the state became true. This difference can be seen by comparing the leftmost categories in Figures 8.5(a) and (b).

However, we also found that user expectations varied even between two different event triggers or two different state triggers. Figure 8.5(a) shows that when the trigger was "If it turns 10:00 am," more users thought the rule would start exactly when the event occurred, compared to when the trigger was "if the doorbell rings." Similarly, for rules with a state trigger, more users thought that the rule would start any time while the state trigger was true if the trigger was "If it is between 3:00 - 4:00 pm," compared to "If it is raining."

The free response questions provide the explanation for this apparent inconsistency. Comparing the 10:00 am trigger to the doorbell trigger, one participant wrote *"Doorbell rings are not predictable, whereas times are."* Similarly, describing rainfall as a trigger, participants wrote, *"When it's raining, the weather can vary and fade-in/fade-out,"* and *"it's more ambiguous as to when it can be called rain."* This could explain why there was more consistency for the triggers involving time.

Participants mostly agreed on the behavior for rules that combined one event trigger and one state trigger. Across the 3 questions of this type, an average of 85% of responses indicated that the rule should activate when the event occurred, as long as the state was true.

Multiple event triggers are considered to be technically valid

When asked when the action should start for the rule, "If it turns 10:00 am and the doorbell rings, do [X]," only about 7% of the respondents said that the action would not occur. A majority (55%) said that the rule should only activate if both events occurred simultaneously (Figure 8.6). The

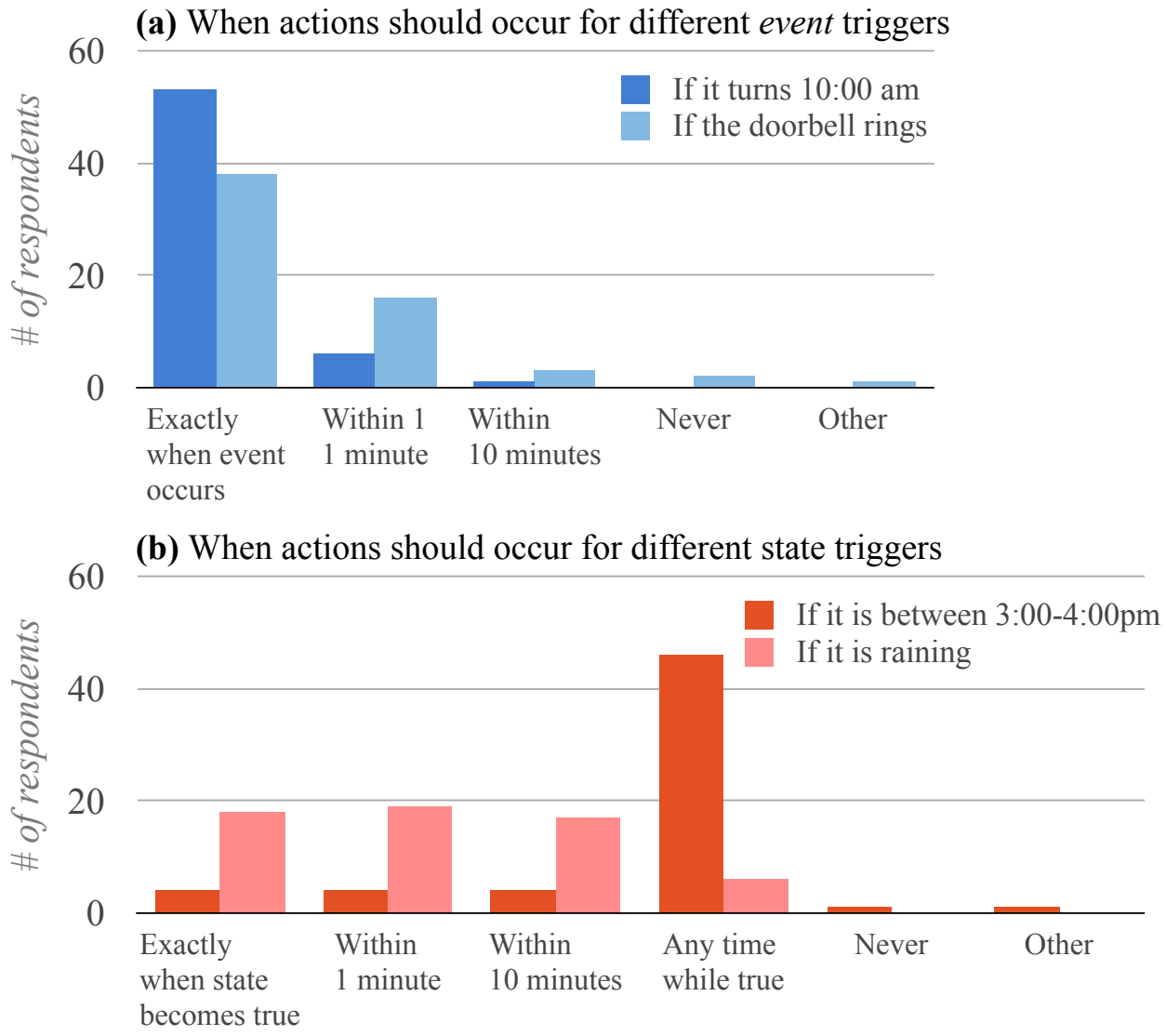


Figure 8.5: When users expect a rule to activate, for particular (a) event and (b) state triggers. The y-axis indicates the number of participants out of 60.

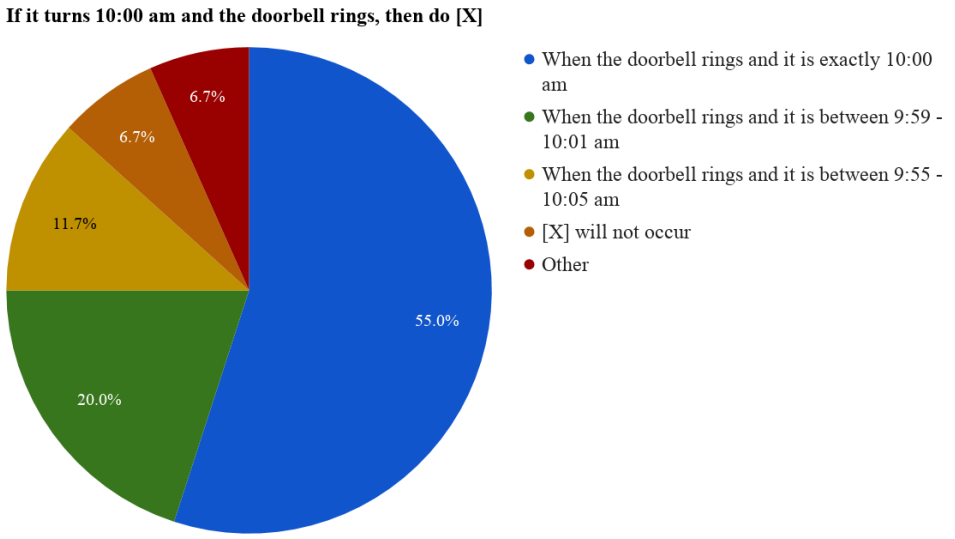


Figure 8.6: When users expect a rule with two event triggers to start.

remaining stated that the rule should activate when one event occurred within 1 or 5 minutes of the other.

In one of the open ended questions, we explained that two event triggers were unlikely to happen at exactly the same time, and asked if such rules should be allowed. Although the wording of the question possibly led them to a particular answer, many respondents gave nuanced responses. For example, one respondent said that the rule should be allowed even if it did not work in practice: *"I think they should be allowed, but I can't really think of many reasons why people would want to do that."* Another participant suggested a way of making it clearer: *"I would make the statement require a duration, so within 5 minutes of it starting..."* Some participants said that the system could automatically figure out a way to make the rules work, *e.g.*, *"Yes, they should be allowed... But their meaning should be as follows: ... So that as long as it is raining as soon as the limited event occurs (doorbell rings), that's the trigger."*

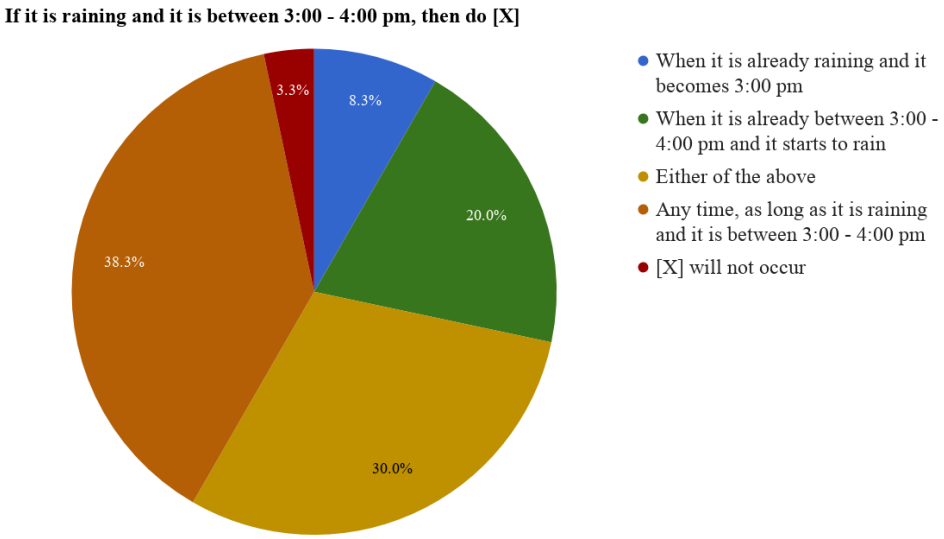


Figure 8.7: When users expect a rule with two state triggers to start.

Expectations varied widely for multiple state triggers

When asked when a rule should start for the trigger, "If it is raining and it is between 3:00 - 4:00 pm," 38% of respondents said the rule could start *any time*, as long as both states were true, while 30% said the rule would start as soon as both rules *became* true. No single answer had a majority. The results are shown in Figure 8.7.

When sustained actions end depends on the trigger

Across all the rules where the action was "send email" or "brew a pot of coffee," participants agreed that the action would finish within one or ten minutes, as expected. However, when the action was "turn the lights on," the responses differed depending on the trigger. When the trigger was an event trigger, an average of 56% of respondents said the light would not turn off. However, when the trigger was a state trigger, an average of only 33% of respondents said the light would turn off. The data is shown in Figure 8.8.

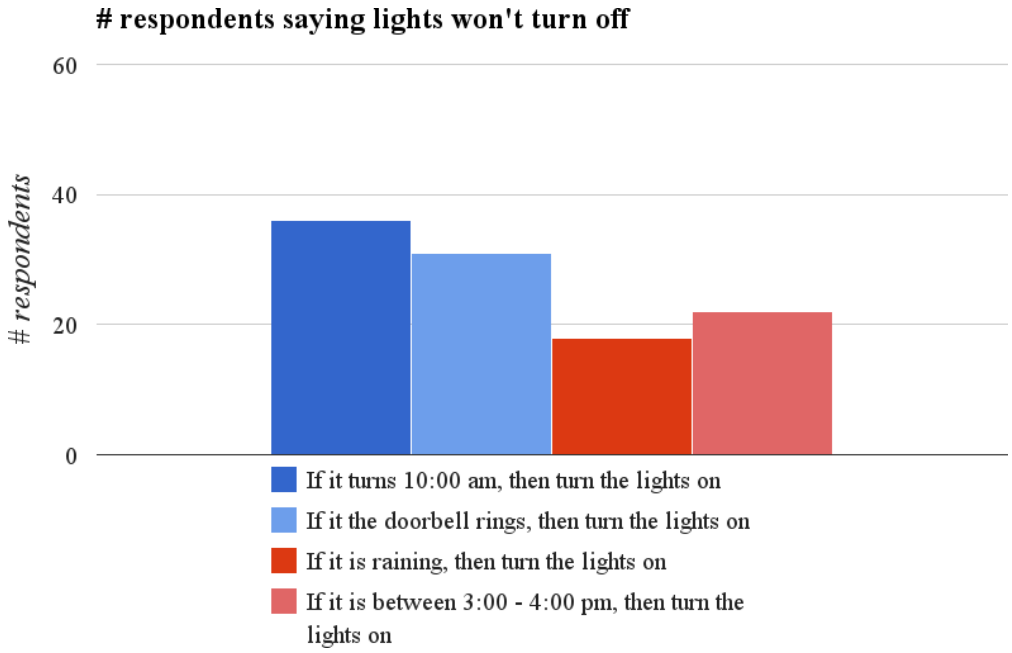


Figure 8.8: Whether users expect a sustained action to end, given different event and state triggers.

8.5 Study 2: Program Creation

Our first study showed that users' interpretations of trigger-action programs for different trigger combinations and sustained actions differed from the logical interpretation we expected. However, this study did not ask users to synthesize rules themselves. While interpretation of programs from descriptions is relevant for many IFTTT users who browse, select, and activate shared programs without modifying them, many other users create programs themselves. Seeing the rule creation process in the context of a fully implemented interface could positively impact the user's mental model of how the program should behave. To investigate whether program creation mitigates the ambiguities observed in the first study, we designed a TAP interface and conducted a second study.

8.5.1 Interface design

Our interface was designed to feature multiple triggers with different trigger and action types, while also resembling IFTTT. The interface borrows visual aspects of IFTTT, as well as the workflow

Triggers	Actions
Daily time (1 event, 1 state)	Switch lights (sustained)
Weather (1 event, 1 state)	Brew coffee (non-sustained)
Doorbell rings (event)	Doorbell rings (sustained)
My location (1 event, 1 state)	My location (non-sustained)
Motion detector (event)	Motion detector (1 sustained, 1 non-sustained)

Table 8.3: Trigger and action categories in our TAP interface.

for creating rules. This allows the results and recommendations from studying this interface to be applicable to a general set of similar interfaces.

Choice of triggers and actions

The interface supported a set of 5 trigger and 5 action categories. The triggers and actions were generic smart home capabilities and did not specify any real-world products. Some trigger categories supported both event triggers and state triggers. For example, the "My location" trigger category could be made into either "I arrive at work" or "I am currently at work." Other trigger categories only supported event triggers, such as the "Doorbell rings" trigger. Similarly, some of the action categories could be made into only sustained actions, only non-sustained actions, or both. A full list of triggers and actions is listed in Table 8.3.

Multiple triggers

Because IFTTT does not support multiple triggers, incorporating this aspect into our interface was the most open-ended design work we engaged in. One observation about the IFTTT interface is that throughout the rule creation process, it shows partially completed previews of the rule, with

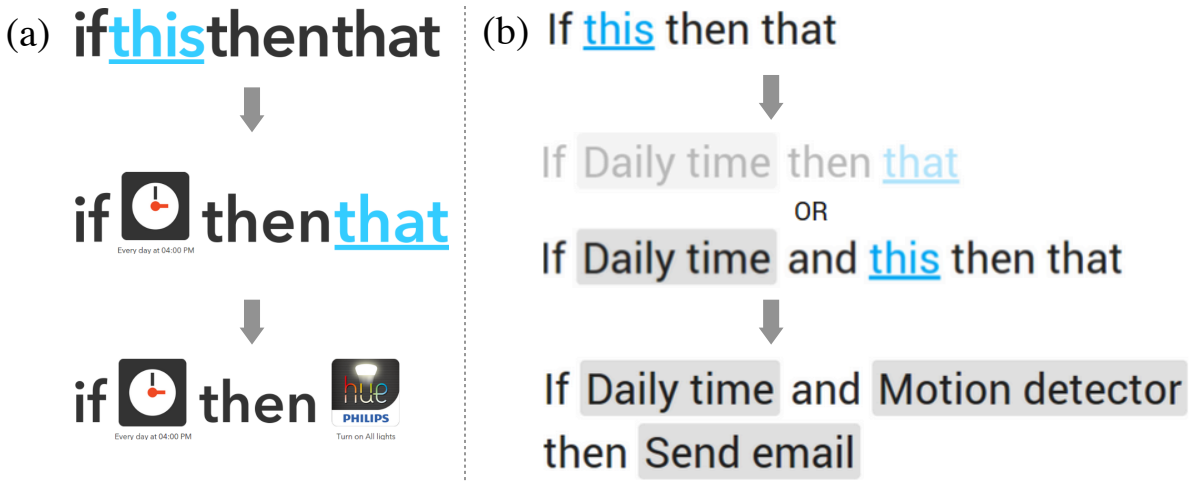


Figure 8.9: A composite screenshot of the rule creation flow in (a) IFTTT and (b) our interface.

a single clickable link to proceed. We decided to replicate this design by offering users a choice after the first trigger was added: add another trigger by clicking "and this" or select an action by click "then that." This process repeated for as many triggers the user wanted to add. Figure 8.9(a) and (b) show how this process looks in IFTTT and in our interface.

Design iterations

The interface design went through several iterations, starting with paper prototyping. We simulated the interface on paper with several participants and informally gathered feedback on the design. During the paper prototyping, we asked participants to create 3 rules, some of which required multiple triggers. No participant had major difficulties with the paper prototypes, although minor adjustments were made between prototypes based on feedback. In the next phase, we implemented a digital prototype of the interface and gathered further informal feedback. Finally, the interface was incorporated into a user study interface.

Wording of triggers and actions

Unlike IFTTT, which offers both an icon and a name for each trigger and action, our interface only included a textual name that describes the trigger and action. These were chosen to be self evident and to clearly communicate the trigger and action type. As in our first study, event triggers included active verbs (e.g., "I arrive at," "I leave") as in IFTTT, while state triggers included the verb *be* (e.g., "I am currently at"). As in IFTTT, our interface involved first choosing a trigger category and then choosing the particular trigger from a drop-down menu.

8.5.2 Questionnaire

The second study contained 5 program creation questions (P1-5, Table 8.4) and 5 multiple choice questions about the participant's interpretation of a given rule (Q1-5, Table 8.5). For program creation tasks, participants were given a description of a desired behavior and were asked to create one or more rules for the smart home to achieve that behavior. Each of these questions was worded to avoid using the words "if" and "then." We also avoided phrasing desired behaviors in ways that could have direct mappings to the interface. Program interpretation tasks were similar to Study 1; users were given textual descriptions of rules programmed in the interface, and were asked a multiple-choice question about the rule.

The program behaviors and the interpretation questions were designed to cover a variety a trigger combinations (Table 8.4, left column). Some questions involved a sustained action, while others did not. For both sets of 5 questions, a sustained action was paired at least once with an event trigger, and with a state trigger.

The 5 program interpretation questions were asked after the 5 program creation tasks. This is because the questions could call attention to issues such as whether it makes sense to have two event triggers in the same rule, which could affect the rules that the participant would create later on. Users were not allowed to change their answers to previous questions in the study. Finally, we collected the same basic demographic information as in the first study.

Trigger types	**Program behavior description**
Single event trigger	P1: You want the lights to turn on at 6:00 pm every day
One event and one state trigger	P2: You want to be notified, via email, should a person be detected in the house while everyone's at work (9:00 am - 5:00 pm every day)
A rule that could involve two event triggers	P3: Your work starts at 9:00 am. On days when you get to work on time, you want to send an email to yourself saying "I got to work on time!"
Single state trigger	P4: You want the thermostat to be off as much as possible, unless the temperature outside is below 40 degrees, in which case the thermostat should be set to 72 degrees.
Two state triggers	P5: You want a pot of coffee to be brewed when it's below 40 degrees outside, but only before 10:00 am every day.

Table 8.4: Program behaviors that users were asked to create rules for in Study 2.

Rule	**Multiple choice question**
Q1: If the time is 6:00 pm, then turn the lights on	If this is the only rule, do you expect the lights to turn off, and if so, when?
Q2: If I arrive at home and the time is between 6:00 - 11:00 pm, then turn the lights on	If this is the only rule, and you arrive home at 5:00 pm, do you expect the lights to turn on, and if so, when?
Q3: If the doorbell rings and the time is 3:00 pm, then unlock the front door for 10 seconds	You are expecting visitors to your house at 3:00 pm. If you wanted them to be let in automatically, would you use this rule? If so, when do you expect the door will unlock?
Q4: If it is snowing, then turn the thermostat to 75 degrees F	When will the thermostat be set to 75 degrees and when will it turn off?
Q5: If the time is between 7:00 am and 10:00 am and the outside temperature is below 40 degrees, then brew a pot of coffee	Suppose it was cold all night. Do you expect the coffee brewer to start, and if so, when?

Table 8.5: Multiple choice questions in Study 2.

8.6 Results from Study 2

8.6.1 Deployment

The user study was distributed through Mechanical Turk. Workers were limited to "Master Workers" who lived in the United States. Each participant was paid $1.50 to complete the survey, which took about 20 minutes to complete. We advertised the study as being about "programming rules for a smart home." Prior to beginning the study, participants were given a short introduction to the concept of smart homes, and given some examples of their capabilities. They were also asked to assume that the smart home was capable of reliably executing all the sensing and actions shown in the interface.

8.6.2 Demographics

There were 42 participants who completed the study. The ages of the respondents ranged from 20 to 66 years, with an average of 37.45 and a standard deviation of 10.9. 22 respondents (52%) were male, and 20 (48%) were female. 22 (52%) respondents said they had no programming experience, 17 (40%) said they had "a little" programming experience, and 3 (7%) said they had programming experience. 34 (81%) said they had not heard of IFTTT before, 8 (19%) said they had heard of it, and none said they had used it before.

8.6.3 Findings

Multiple event triggers were used in practice

In the creation of P3 (Table 8.4), 21 (50%) respondents created the rule, "If I arrive at work and it turns 9:00 am, then send myself an email," as opposed to "If I am currently at work and it turns 9:00 am", which only 9 (21%) respondents made (Figure 8.10). We consider the first rule to be incorrect, both because it uses two event triggers and it would not fire if the person was on time by being at work early.

One reason why this could have happened is that users do not naturally think of "arriving at

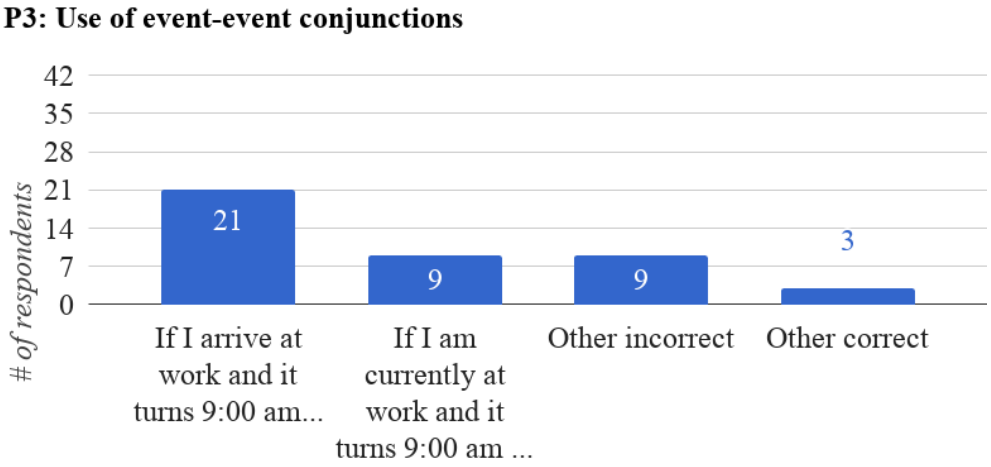


Figure 8.10: The distribution of rules that users created for P3.

work" to be an event at a specific point in time, but as a state which is true all day as soon as you arrive at work. In our interface, the "am currently at" and "arrive at" options were placed adjacent to each other, so users are likely to have seen both options. However, this raises a thematic issue with interfaces like IFTTT, which provide a natural language mapping from the interface to program behavior. Because natural language itself can often be ambiguous, the meaning of the programs they describe can become unclear as well.

Q3 (Table 8.5) also demonstrates that people are okay with multiple event triggers. 15 (36%) respondents said that they would use the rule shown, and that the door would open if the doorbell rang at exactly 3:00 pm. 13 (31%) respondents said they would use this rule, and that the door would open if the doorbell rang between 3:00 - 3:01 pm. Only 13 (31%) respondents said that they would not use the given rule. This shows that most users believed that the rule would work, either because they expected the visitors would arrive very promptly at 3:00 pm, or because the system would work even if the visitors were early or late by a few minutes.

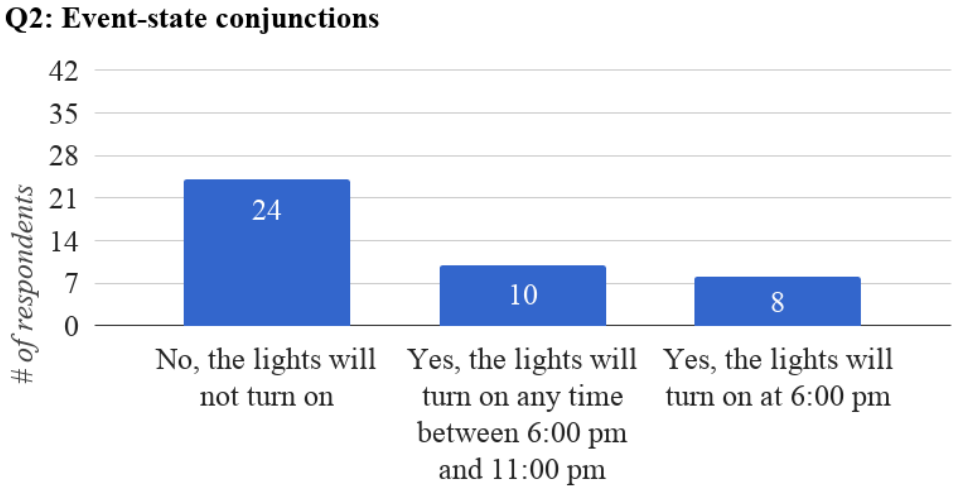


Figure 8.11: The distribution of responses that participants gave to Q2.

Event & state triggers were hard to reason about

We were surprised to find that the composition of an event trigger with a state trigger was not well understood. For P3 (Table 8.4), the ideal combination of triggers would have either been "If it turns 9:00 am and I am currently at work," or alternatively "If the time is between 8:00 am and 9:00 am and I arrive at work," both of which are combinations of an event trigger and a state trigger. However, only 9 (21%) users created the first rule, and 3 (7%) created a rule similar to the second rule (shown as "Other correct" in Figure 8.10). This could suggest that rules with a combination of an event and a state trigger are hard for users to synthesize.

Additionally, in response to Q2 (Table 8.5), only 24 (57%) participants said that the lights would not turn on, while the remainder said that the lights would turn on, either any time between 6:00 - 11:00 pm or exactly at 6:00 pm (Figure 8.11).

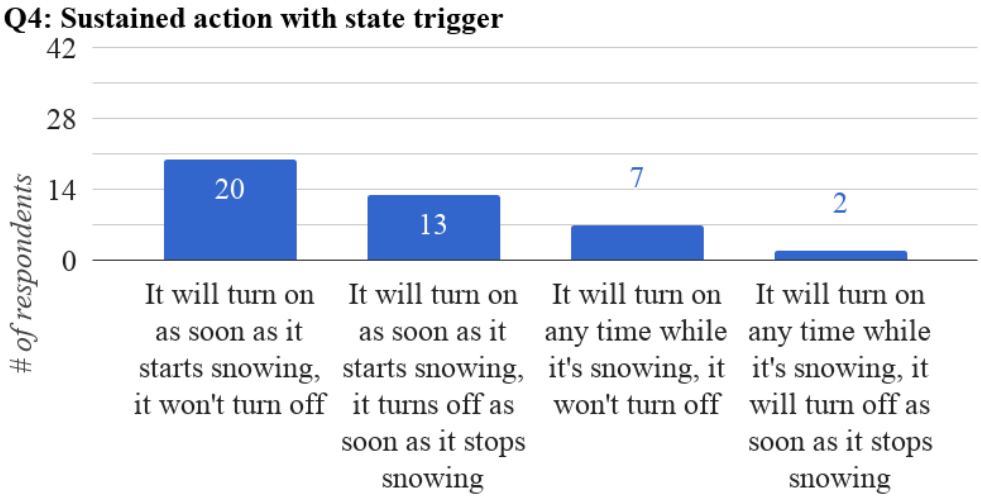


Figure 8.12: The distribution of responses that participants gave to Q4.

Users had varied mental models for state triggers

Of the two minority responses to Q2, 10 (24%) participants said that the lights would turn on any time between 6:00 - 11:00 pm, and 8 (19%) said that the lights would turn on at 6:00 pm. This shows that users are still not sure if state triggers activate as soon as the state becomes true, or at any time while the state is true.

Similarly, for Q4, 33 (79%) users said the thermostat would be set as soon as it started snowing, while the remaining 9 (21%) said that the thermostat would be set any time while it was snowing (Figure 8.12). Although a majority answered in the way we expected, in a real-world deployment, having 21% of users misunderstand this type of trigger would be a non-trivial problem.

In response to Q5, 26 (62%) participants said that the coffee brewer would start at 7:00 am, while 15 (36%) said that the coffee brewer would start any time between 7:00 - 10:00 am. These results are consistent with Study 1, showing that most users expect state triggers to activate as soon as all the states became true, but a non-trivial percentage of users had a different interpretation.

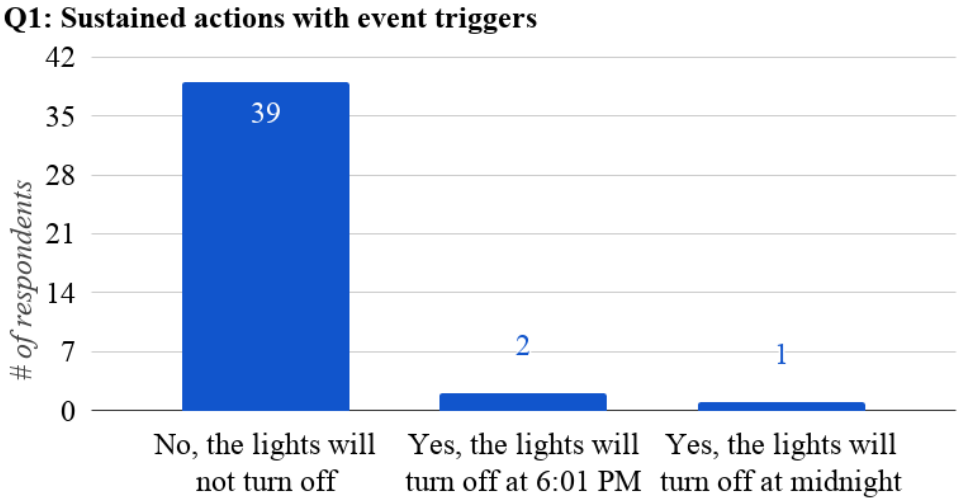


Figure 8.13: The distribution of responses that participants gave to Q1.

Users disagreed on sustained actions and forgot to undo them

While programming P4 (Table 8.4), users were asked to keep the thermostat off as much as possible, unless the temperature outside dropped below a certain level. However, most users forgot to turn the thermostat off once they had set it. 33 (79%) participants made the rule, "If the temperature outside is below 40 degrees, then set the thermostat to 72 degrees," but only 6 participants made both that rule and another rule, "If the temperature outside is above 40 degrees, then turn the thermostat off."

The multiple-choice questions revealed that more people thought that sustained actions would be undone automatically when the trigger was a state, compared to when the trigger was an event (Figures 8.13, 8.12). Q1 and Q4 asked users when a sustained action would end. For Q1 (event trigger) 39 (93%) respondents said that the lights would not turn off. However, for Q4 (state trigger) only 27 (64%) respondents said that the thermostat would not turn off. Q4 also shows that users did not universally agree whether the thermostat should turn off by itself or not, as 15 respondents (36%) said the thermostat would turn off as soon as it stopped snowing.

User interpretations may be influenced by existing products

In the creation of P2, only 30 (71%) respondents created what we considered to be the correct rule, "If it is between 9:00 am - 5:00 pm and motion is detected, then send myself an email." The most common mistake was to not limit the time the rule was active, as in, "If motion is detected, then send myself an email," This rule was made by 8 (19%) participants. We speculate that some of those participants did not feel the need to specify a time range condition themselves, since many home security systems are manually deactivated through a passcode when an occupant arrives home.

8.7 Discussion

Our studies were designed to assess the accuracy of people's mental models of trigger-action programs in the presence of different trigger and action types within a system that allows conjunctions of multiple triggers. Our emphasis, in comparison to previous user studies involving TAP, is on the distinction of different trigger and action types. While the differences between these types impact the underlying meaning of programs, previous work or existing systems do not make a distinction at the interface level. Our studies reveal that this causes ambiguities in interpreting meanings of programs and errors in creating programs with an intended behavior.

8.7.1 Problems due to mental model inaccuracies

Based on our two studies we make the following high-level characterization of mental-model problems:

- Trigger timing: Participants disagreed on whether state triggers start as soon as the state becomes true, or any time while the state is true. Furthermore, users give fuzzier triggers like rainfall or the doorbell ringing leeway in terms of start time.

- Program validity: Participants disagreed on whether multiple events should be allowed.

- Action reversal: Participants disagreed on whether sustained actions end automatically when paired with a state trigger. They did not add rules to undo sustained actions, likely as a consequence of believing that the actions would end automatically.

We observed similar problems in both of our studies, suggesting that program creation does not improve the users' mental model so as to mitigate issues that exist in program interpretation.

8.7.2 Interface improvements

Next, we describe four different interface adaptations that could address some of these problems.

Prompts

One way to mitigate some of the common issues would be to include prompts to warn users in situations that were empirically demonstrated to cause ambiguities. These prompts would inform the user about the true semantics of the programs they create. For example the system could: (1) warn users that two events are unlikely to happen exactly at the same time, after the user adds a second event trigger; (2) tell users when the action will start (e.g., immediately or within a minute) when they add a single state trigger or "fuzzy" event trigger; (3) warn users that an action will or will not automatically revert when they add a sustained action. Such prompts could also offer semi-automated ways for addressing the ambiguity. For instance, when a rule with a sustained action is created, the prompt could allow easy creation of a second rule that undoes the action.

Disallowing confusing options

Another mechanism would be to disallow the creation of programs that are invalid with respect to the true semantics of the particular TAP system. For instance, our system could disallow creation of programs that have (i) more than one event trigger, or (ii) purely state-type triggers.

Trigger duality

Conjunctions of triggers are meaningful for combining a single event trigger and multiple state triggers. To support combinations of all categories of triggers it is important for the system to provide both state and event triggers related to the same underlying concepts; e.g., include all three of "it starts raining," "it is raining," and "it stops raining." For triggers that are naturally expressed as states, the system can provide all state changes as events. For triggers that are naturally expressed as events, states could be defined by adding an adjustable time window around the event. For example, "the doorbell rings" event could become "the doorbell rang in the past 2 minutes" state. While the duality of states and events could naturally draw users' attention to the distinction between the two types, making the categorical distinction at the interface level (through grouping or actually naming trigger types as states and events) could further improve the user's mental model.

Top-level statements

One of the key ways in which TAP interfaces achieve simplicity is by mapping natural language elements (*e.g.*, "if" and "then") to program behavior. This mapping can create ambiguity, since natural language itself is inherently ambiguous. At the same time, humans are good at refining natural language statements to more accurately communicate an intended meaning. Similarly, we propose extending TAP interfaces to support alternative high-level statements that more accurately indicate the type of triggers and actions they support. Some examples for two-trigger statements are given in Table 8.6. These specific statements could make it easier for people to combine the right types of triggers or remember to undo the effect of sustained actions.

The mechanisms proposed above individually address a subset of the problems observed in our study. Therefore, mitigating all of issues will require combining these mechanism. Determining the right combination, as well as the details of each mechanism, such as which high-level statements should be included, require further design and empirical evaluation. We hope to tackle these questions in our future work.

IF *event-trigger* THEN *action*
WHEN *event-trigger* DO *action*
WHILE *state-trigger* DO *sustained-action*
AS LONG AS *state-trigger* DO *sustained-action*
IF *event-trigger* WHILE *state-trigger* THEN *action*
IF *state-trigger* WHEN *event-trigger* THEN *action*
WHILE *state-trigger* AND *state-trigger* DO *sustained-action*
AS LONG AS *state-trigger* AND *state-trigger* DO *sustained-action* OTHERWISE DO ¬*sustained-action*

Table 8.6: Alternative high-level program statements for particular trigger and action type combinations.

8.7.3 Limitations

Our work has several limitations. In both of our studies, we asked participants to create or answer questions about programs that were not their own idea, so the intent of the behaviors we described may not have been clear. The interface we tested was designed to resemble IFTTT. This limits our findings and results to interfaces which are similar to our interface or to IFTTT. Also, our subjects were anonymous users from Mechanical Turk, who may not be representative of potential users of TAP systems.

As we saw throughout the study, wording played an important role in how users interpreted rules. This suggests that our results could be dependent on the exact choice of wording we used in our questions and in our interface. For example, in question 2 of the user study, we say "while everyone's at work (9:00 am - 5:00 pm every day)," which could be interpreted as either a "My location" trigger or a "Daily time" trigger, or both. Similarly, we discussed in our user study findings how the meaning of "If I arrive at" versus "If I am currently at" could have differed between respondents.

Chapter 9

END-TO-END MOBILE MANIPULATION PROGRAMMING

In this chapter, we describe the design and evaluation of *Code3*, an end-to-end robot programming system that integrates perception, motion specification, and scripting components described in earlier chapters. In particular, it utilizes *CustomLandmarks* (Chapter 4) to let users create their own landmark detectors, programming by demonstration to specify motions (Chapter 6), and *CustomPrograms* (Chapter 7) for task scripting. With these systems combined, users can program the robot to do mobile manipulation tasks using only simplified interfaces, and without writing compiled code.

The goal of the system was to provide a way for novice users to quickly program mobile manipulator robots. However, we did not want the system to be overly simplistic. Our goal was for the system to be expressive enough that a more advanced user could program non-trivial tasks using the system. With these goals in mind, we conducted two evaluations. Our first evaluation demonstrated that 10 non-roboticist programmers, who were novices to *Code3*, could program useful manipulation tasks on the PR2 robot in under an hour, after 90 minutes of training. In our second evaluation, we showed how an expert user could use *Code3* to program complex manipulation tasks like playing tic-tac-toe with a human player and reconfiguring an object to make it graspable. Ultimately, we envision non-roboticist programmers of all skill levels using *Code3* to rapidly prototype and program task behaviors for mobile manipulators in semi-structured environments such as retail stores, warehouses, and office buildings.

We implemented *Code3* for the PR2 robot. Open-source code and documentation for using *Code3* can be found at `https://github.com/hcrlab/code3`.

Because we have previously described *CustomLandmarks* and *CustomPrograms*, we first describe the specific programming by demonstration system we used for *Code3*. Next, we provide

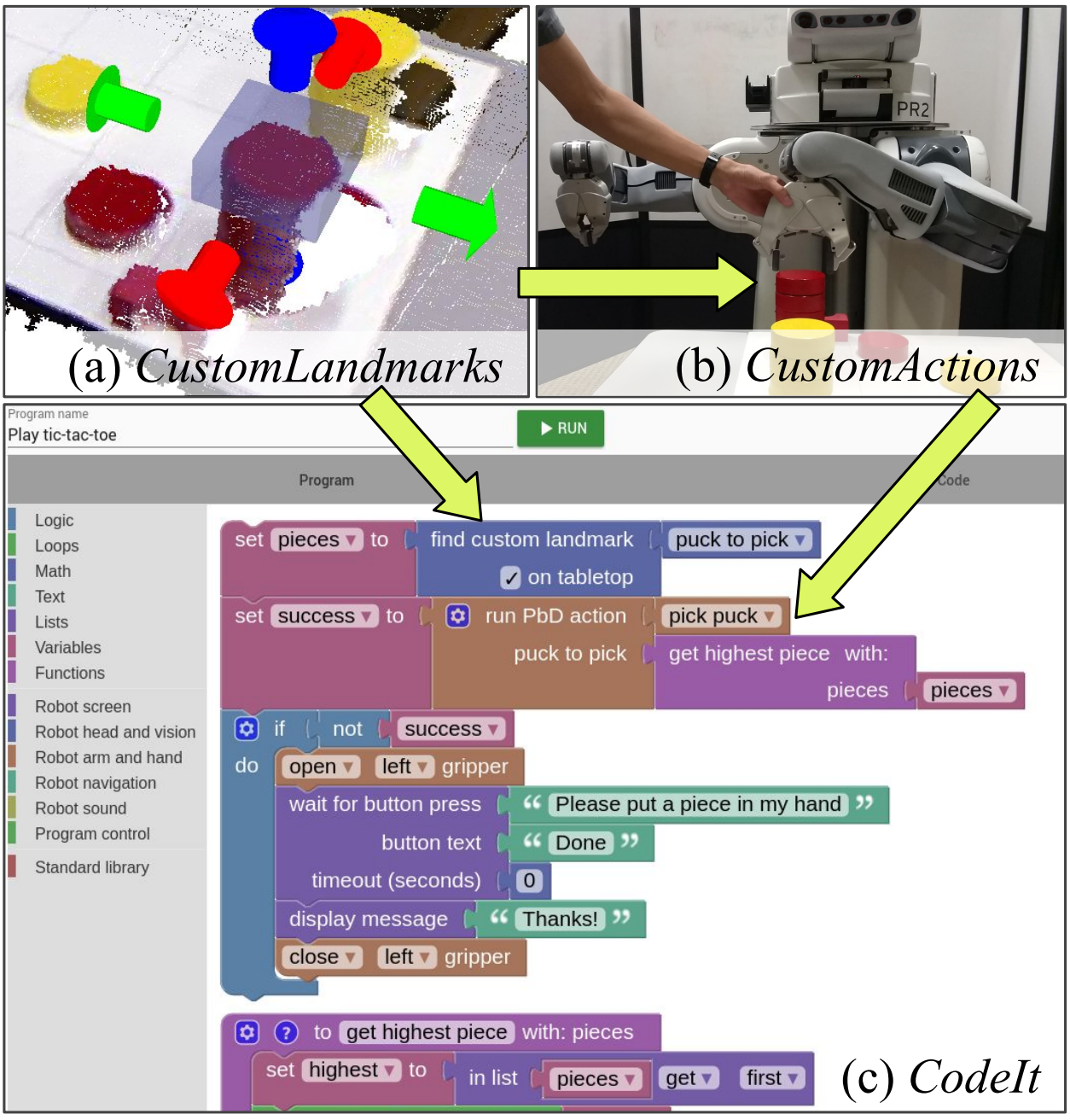


Figure 9.1: Components of the *Code3* system.

a walkthrough of how a user would use all the different components to program a task. Then, we present the design and results of a user evaluation with novices to the system. Finally, we provide examples of tasks that an expert user programmed using *Code3*, demonstrating that the system can be used to program more advanced tasks, as well.

9.1 System overview

9.1.1 Programming by demonstration for Code3

The programming by demonstration tool used in *Code3* was a modified version of the system presented in [5]. We refer to the specific variant of this software that we developed for *Code3* as *CustomActions*. Below, we describe the workflow for creating, representing, and executing Programming by Demonstration (PbD) actions for *Code3*.

PbD action representation

PbD actions are represented as a sequence of poses for the robot's end-effector(s). The system also stores whether the end-effector is open or closed at each pose (for a gripper). The locations of the poses can be defined relative to either the base of the robot or to a landmark. If the user defines an end-effector pose relative to a landmark, then the system will adjust the end-effector pose whenever it detects that the pose of the landmark has changed. A simple example of an action is to pick up an object, whose position could vary, and put it into a box at a fixed position relative to the robot. The user would first create a custom landmark of the object. Then, the user would demonstrate grasping the object, moving the end-effector over the box, and letting go of the object. The end-effector poses involved in grasping the object would be defined relative to the location of the object, while the poses to drop the object in the box would be relative to the robot's base.

Programming a PbD action

To specify an action, users first use the *CustomLandmarks* box interface to define the landmarks that will be involved in the action. They then hold the robot's arms and move them to the desired

poses. Users save the poses by using voice commands or by clicking a button in a GUI. In the GUI, pictured in Figure 9.4, users can define some poses to be relative to one of the landmarks they defined earlier. When done, users give the action a unique, human-friendly name. This name is used to invoke the PbD action from within *CustomPrograms*.

PbD action execution

To execute an action, the robot first points its head to locate each of the landmarks referenced in the action. Because the landmarks are not necessarily on a tabletop, *CustomActions* looks for each landmark near the location where it was last found. If the landmark has never been located before, the robot looks where the landmark was during the action demonstration. If the system locates more than one instance of a landmark, it uses the one that was detected with the lowest error score. Next, the system adjusts any poses that were defined relative to those landmarks. Finally, the robot moves its arms through the poses, opening or closing its grippers as programmed.

9.1.2 Integration of Code3 components

CustomPrograms provides APIs to use *CustomLandmarks* and *CustomActions*. The integration of *CustomPrograms* and *CustomLandmarks* occurs via a primitive that searches for a landmark in the current scene: `findCustomLandmark(landmarkName)`. The argument of this primitive is the name of a previously created custom landmark. When called, the robot searches for the landmark in the current scene and returns a (possibly empty) list of locations where the landmark was found. The user's program can examine the length of the list and iterate through it. *CustomPrograms* also allows users to read the x, y, and z positions of the detected landmarks, in the coordinate frame of the robot's base. The meanings of the x, y, and z values were documented for the programmers.

The integration of *CustomActions* with *CustomPrograms* occurs via a primitive that executes a previously programmed action: `runPbdAction(actionName)`. The argument of this primitive is the name of the action to run. The primitive returns true if the action executes successfully

and false otherwise. An action fails if a needed landmark cannot be found in the scene or if an arm pose relative to a landmark is unreachable.

CustomPrograms also implements a feature that we call *preregistration*. Suppose there are three identical cans in the scene and the user has created a custom action to pick up a can and put it in a box. As described earlier, *CustomActions* will execute the action with the can that was detected with the lowest error score, which can be an arbitrary choice in practice. Preregistration allows users to specify a particular landmark to execute an action with in this situation. First, users call `findCustomLandmark` in their *CustomPrograms* program, which returns a list of detected landmark locations. Then, users can iterate through the list of locations and select whichever one their task needs. Finally, the landmark location can be passed as an optional argument to the `runPbdAction` primitive. This causes the robot to execute the action using the landmark that was passed in. In the above example, the user could use preregistration to have the robot put the three cans into the box in right to left order. In Section 9.6, we describe how we used preregistration to pick a game piece from the top of a stack. Figure 9.1(c) illustrates how preregistration looks in *CustomPrograms*. The first block finds a custom landmark of a game piece. The user iterates through the list of game pieces and selects the highest one through a function called "get highest piece." Finally, that piece is passed into a custom action to pick a game piece.

9.2 Primitives for the PR2

CustomPrograms, described in Chapter 7, was originally designed for the Savioke Relay, a mobile robot with no arms. We designed and implemented a new API to support the manipulation capabilities of the PR2 robot. We also added a touchscreen tablet to the PR2, so that the robot could display messages and receive touch input. Below, we briefly list the primitives we added to the PR2.

- Tablet interaction: `displayMessage(text)`, `askMultipleChoice(question, choiceList)`
- Head control: `lookAt(upDegrees, leftDegrees)`

- Gripper control: `setGripper(leftOrRight, openOrClosed)`, `isGripper(leftOrRight, openOrClosed)`
- Pre-programmed tucking or deploying of arms: `tuckOrDeployArms(leftAction, rightAction)`
- Pausing: `waitFor(numSeconds)`
- Text-to-speech: `say(text)`

9.3 Workflow description

Next, we describe the typical workflow for using *Code3* by walking through an example, taken from one of the tasks from our user study. In this example, the user creates a program for snack delivery that allows a person to choose between one of two snacks, which the robot fetches from a snack area and delivers to the person. The programming of this task proceeds as follows:

1. The user creates an empty action in *CustomActions* and uses it to direct the robot's head to look at the snack.
2. In the *CustomLandmarks* interface, the user draws a 3D box around the first snack (*e.g.*, a chip bag) and saves it.
3. In *CustomActions*, the user saves a sequence of arm poses relative to the landmark of the bag of chips, grasping it.
4. The user gives this action a name (*e.g.*, "Pick up chips") through the GUI.
5. The above process is repeated for the other snack.
6. The user accesses *CustomPrograms* to develop a task program. The program asks which snack the person wants and then directs the robot to travel to the snack area. Next, it uses

an if-statement to decide which snack pick-up action to run. Finally, it travels back to the person and presents a touchscreen button that lets him or her open the robot's gripper.

Certain steps in this workflow must be done in order. For example, one must point the robot's head in the direction of the snack so that the landmark can be seen. And, one must create the landmark for a snack before demonstrating the action that grasps it so that the saved poses can be defined relative to the landmark. Our implementation forces users to create landmarks as part of the process of creating the actions that manipulate them. However, alternative implementations could support an independent landmark creation process and re-use of landmarks across different actions.

9.4 Novice user programming with Code3

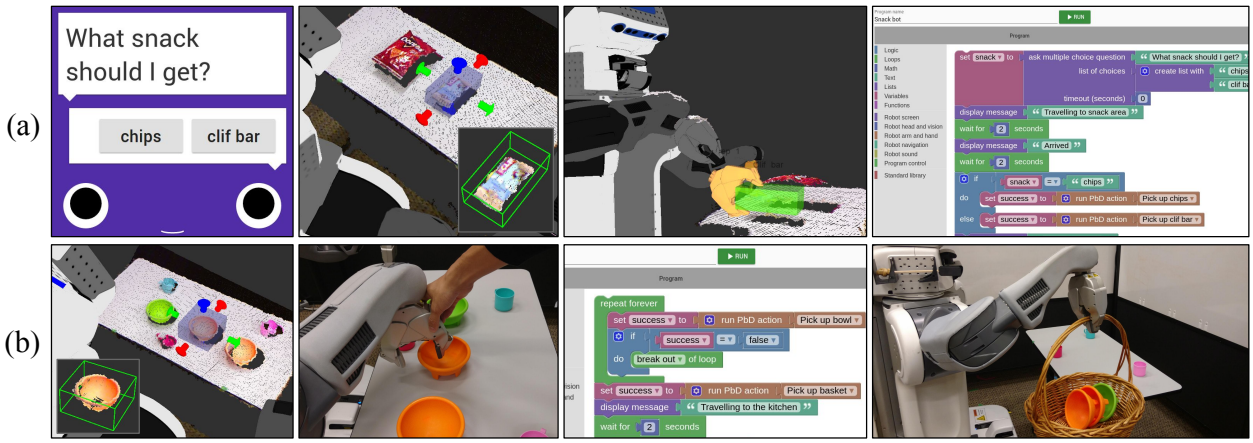


Figure 9.2: (a) In the *snack-bot* program, the robot fetches a user-requested snack. Users made landmarks using *CustomLandmarks* (second image, saved landmark shown inset), then demonstrated actions to pick the snacks. The third image shows the *CustomActions* GUI and the last image shows a user program made with *CustomPrograms*. (b) In the *waiter-bot* program, the robot places bowls into a basket and picks up the basket.

This section describes an observational user study of *Code3* that we conducted. Our goal was

to assess the system's learnability and usability for programmers with little robotics experience. To study this, we trained users to program the PR2 using *Code3* and asked them to program manipulation tasks.

9.4.1 Procedure

The study consisted of a 60-minute training session and a 90-minute programming session. The sessions could be scheduled back-to-back or up to 3 days apart. At the beginning of the training session, users were told to imagine starting a new job at a company that programmed the PR2 robot. An experimenter read from a script to train participants on: (a) how to create actions in *CustomActions*, (b) how to use *CustomLandmarks*, and (c) how to use *CustomPrograms*, including how to run custom actions from code. Users were not trained on preregistration because the user study tasks did not require it.

During the first 30 minutes of the programming session, participants worked on a familiarization task, with proactive guidance from the experimenter when needed. Next, participants received a description of the program that they would be asked to create. We assigned participants to program one of two tasks (described below in Section 9.4.2). We alternated the order in which we assigned tasks to participants. They filled out a pre-task questionnaire, which asked users to describe the actions and landmarks that would be needed and to outline the overall program. This was included to help participants organize their thoughts.

Participants were given 50 minutes to program the task. They could test their programs on the robot at any time. We considered a task finished once the robot was able to successfully execute the task. If a participant finished their task before the end of the study, they were asked to program the other task as well. Once the user finished or ran out of time, they were asked to stop working and fill out a final questionnaire.

The experiment took place in a laboratory setting with a PR2 robot and a nearby desktop computer. *Code3* interfaces were pre-loaded on the computer. Participants used a wireless microphone to send voice commands to the PR2 when using *CustomActions*. The experimenter was located in the same room near the robot and was available to answer questions. We established the following

guidelines for when the experimenter could proactively help the participant. For technical issues with the robot or the system, including issues with the voice recognition system mishearing participants, the experimenter helped right away. For other issues, when the experimenter suspected the participant was going down an unproductive path, the experimenter took note of the time and intervened after a one-minute wait. We felt these were reasonable expectations for a programmer's first day on the job. We took note of the questions asked by participants and the interventions performed by the experimenter.

Participants were recruited from computer science mailing lists at the authors' university. They could not be robotics researchers and needed to have two or more years of programming experience. They were offered a $30 gift card for participating and a $5 bonus for each completed task.

9.4.2 Tasks

Familiarization task: Grocery-bot

Users were asked to program the robot to put all cans found on a table into a box. The positions and the number of cans could vary, but the box was in a fixed location on the table. To accomplish this task, users needed to create a custom landmark of a can and program an action to pick it up and drop it into the box. They also needed to write a *CustomPrograms* program that repeated the action until the the robot put all the cans away.

Snack-bot

In this task, the robot had to ask the user which of two snacks to retrieve: an energy bar or a bag of chips. The robot would then travel[1] to a snack area and pick up the appropriate snack. The exact positions of the snacks could vary. The robot would then travel back to the user and wait for the user to press a button on the tablet before opening its gripper. This required the user to program

[1] Because our robot was immobile at the time of the study, users were told to simulate traveling by displaying a message on the touchscreen interface and waiting for a few seconds.

two actions with different custom landmarks (one for each snack) and choose which to execute. The snacks were elevated off the table with small pedestals to make them easier to grasp.

Waiter-bot

In this task, the robot started at a table with plastic bowls and cups on it. Users had to program the robot to remove all the bowls from the table and put them in a basket on the floor. Once all the bowls were in the basket, the robot had to pick up the basket and "travel" to the kitchen to deliver the bowls. As in the familiarization task, the exact number and position of the bowls could vary. This task required users to repeat an action until there were no more bowls. It also required users to program an action to pick up the basket before the robot traveled to the kitchen.

9.4.3 Measures

During the study, the experimenter recorded data on what questions participants asked and what help was given. Additionally, we recorded data on when participants used the *CustomLandmarks*, *CustomActions*, and *CustomPrograms* interfaces. We also gathered video data and screen recordings.

In a post-task questionnaire, we first administered the System Usability Scale (SUS) survey [20]. This widely used survey asks a series of ten 5-point Likert scale questions with alternating polarities, such as "I think that I would like to use this system frequently" and "I found the system unnecessarily complex." Scores from the SUS survey can range from 0 (worst) to 100 (best); a meta-study of 2,324 SUS surveys administered in published research found that the average score was 70.14 with a standard deviation of 21.71 [11]. Our questionnaire also asked users to qualitatively describe what was difficult about using *CustomLandmarks*, *CustomActions*, *CustomPrograms*, and the *Code3* system as a whole.

The post-task questionnaire next asked users to rate how difficult they felt the programming task was on a 5-point Likert scale. We then asked users to describe a task that the robot could plausibly be programmed to do using *Code3* in: a home or office environment, in a commercial

Task	M	N
Take a laundry basket to a laundry machine, load clothes, and start the machine	8	8
Take a mug to an automatic coffee maker, brew coffee, and deliver the mug back	7	9
Play a game of tic-tac-toe on a physical game board	5	6
Pick a certain set of items from a shelf	8	9
Pour ingredients into a large bowl and mix together	7	10

Table 9.1: User ratings for the feasability of hypothetical tasks. M is the number of participants (out of 10) who rated the task as either "Very easy," "Easy," or "Not easy or difficult." N is how many thought the task was achievable in 8 hours or less.

environment, and, optionally, in an industrial environment.

Users were presented with a list of hypothetical tasks and were asked to rate how difficult it would be to program them with *Code3*. Options included: "Very easy," "Easy," "Not easy or difficult," "Difficult," "Very difficult," or "Impossible." They were also asked to estimate how long it would take to program the tasks, on the following scale: less than 30 minutes, 30-60 minutes, 1-2 hours, 2-8 hours, 1-2 work days, 3+ work days, or "Impossible." Users were asked to explain their answers to both these questions.

The hypothetical tasks we asked about were: (1) load clothes into a washing machine and start the machine, (2) take a mug to an automatic coffee machine and deliver the coffee back, (3) play a game of tic-tac-toe on a large, physical game board, (4) pick a certain set of items from a shelf, and (5) pour small bowls of ingredients into a large bowl and mix them. These represented some of the most complex tasks that we believe *Code3* could be used to program.

Finally, we asked users to specify their age and gender and to rate, on a 5-point Likert scale, their prior programming experience in general and with the programming of robots.

9.5 User evaluation results

Ten users participated in the study, 6 male and 4 female. Their ages ranged between 19-37, with an average of 26.6 (SD=6.2). On a 5-point Likert scale, users gave their prior programming experience a mean rating of 3.9 (SD=0.74, min=3, max=5) but only 1.6 (SD=0.84, min=1, max=3) in terms of prior experience with programming robots.

9.5.1 Task performance

In the 50 minutes given to them, all 10 users successfully completed at least one task, and 3 users completed both tasks. Seven participants completed the *snack-bot* task, and 6 completed the *waiter-bot* task. On a 5-point Likert scale, participants rated the difficulty of the *snack-bot* task a 2.29 on average; the *waiter-bot* task was rated a 2.67. On average, participants spent 31:38 minutes programming a task (SD=13:40 minutes, min=15:40, max=50:50). They asked 1.38 questions (SD=1.56) and were proactively helped by an experimenter 1.92 times (SD=1.93) per task.

9.5.2 Perceived usability and usefulness

On the System Usability Survey, users scored the system an average of 66.75, with a low of 35, a high of 87.5, and a standard deviation of 16.95. This is below average compared to other systems evaluated with the SUS, and some researchers would interpret this to mean that the system's usability is "marginally acceptable" [11].

Table 9.1 shows how users perceived how easy it would be to use the system for certain tasks. Users generally believed that most of these tasks would not be difficult to do and would take one work day or less to program. For the tic-tac-toe game, users wrote about the difficulty of programming the strategy part of the game using the *CustomPrograms* interface. For the laundry task, one user pointed out that it would be easy for the robot to simply dump laundry in a top-loading laundry machine. Others imagined the robot using a front-loading machine and said it would be impossible to make a custom landmark representing the clothes. Two users said the shelf task might be difficult because a separate action and custom landmark would have to be created to

pick each type of object. For the other tasks, users expressed concerns that the robot might spill coffee or ingredients.

Measure	Result
# participants who completed exactly 1 task	7/10
# participants who completed 2 tasks	3/10
Mean general programming experience rating (1-5)	3.9
Mean robot programming experience rating (1-5)	1.6
Mean SUS score	66.75
Mean task completion time	31m 38s
Mean time spent in *CustomActions*	7m 56s
Mean time spent in *CustomLandmarks*	3m 41s
Mean time spent in *CustomPrograms*	5m 31s
Mean time spent testing solution	5m 29s
Mean questions asked per task	1.38
Mean instances of proactive help given	1.92
Mean task difficulty rating (1-5)	2.38

Table 9.2: High-level summary of *Code3* user study results.

9.5.3 Characterization of system usage

Across all tasks, users spent an average of 3:41 minutes using the *CustomLandmarks* interface, 7:56 minutes using *CustomActions*, and 5:31 minutes using *CustomPrograms*. Additionally, they spent an average of 5:29 minutes testing their solutions. Figure 9.3 presents a visualization of

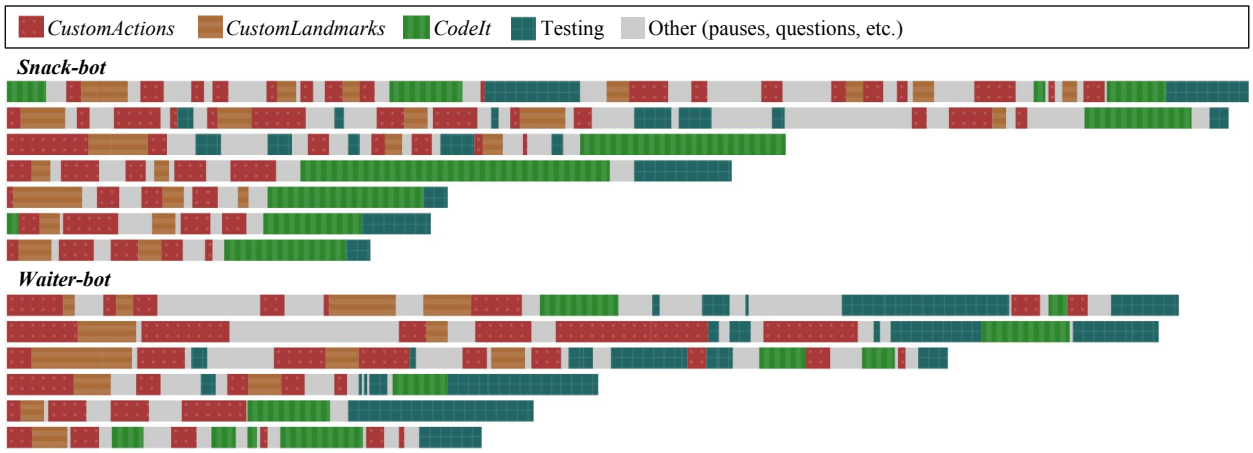


Figure 9.3: A visualization of which *Code3* components participants used while programming each of the user study tasks. For each row, the x-axis represents time that a participant spent working on a user study task. The color of the bars indicate which activity the user was doing; the length of the bars indicate the amount of time spent. The "Other" category includes time when users asked questions, were given help, or were paused and not using any particular interface. The length of the bars in the legend represent 1 minute of time.

which components were in use by the users for each task. The visualization shows that generally, users created custom landmarks and actions first, then built their *CustomPrograms* programs, and then tested their solutions last. We found that users switched between using *CustomActions* and *CustomLandmarks* frequently. This is because the workflow for creating a custom landmark is embedded in the *CustomActions* workflow. However, some users sketched out their *CustomPrograms* programs earlier or conducted small scale tests of custom actions. We also note that users used *CustomActions* many times even though both tasks only required 2 actions each. This reflects the fact that users edited or remade actions after testing or after receiving experimenter guidance.

9.5.4 Challenges in system usage

We assessed challenges participants had using the system through survey questions, questions users asked, and proactive guidance provided by the experimenter. Below, we describe the main challenges users experienced.

CustomLandmarks/CustomActions integration

Two conceptual issues came up regarding the integration of *CustomLandmarks* in the *CustomActions* system. First, many users assumed that they needed to create custom landmarks for all objects in the scene, even when the objects were in a fixed position or were not going to be manipulated. The second issue arose from the fact that creating custom landmarks was embedded in the workflow of creating a custom action. There was no way to create a landmark and use it across multiple actions or to create a landmark and retroactively apply it to an action. Users told us that the system's behavior was unexpected in this respect:

> *User 10:* I placed all the landmarks I would need for all actions at the beginning of one task, and had to place one landmark again because I needed it for a separate action.
> *User 3:* It would have been nice if I could have created a landmark after I already did the actions.

We could resolve these issues by having a separate workflow for creating a custom landmark, which could be imported into *CustomActions* through its GUI.

Voice recognition in CustomActions

The most common technical issue was with the voice recognition system in *CustomActions*. It was common for conversations between the user and the experimenter to be misheard by the robot as voice commands. The robot made various beeping sounds to acknowledge recognized voice commands; however, there was not a one-to-one correspondence between the voice commands

and the sounds. As a result, many users said they were unsure of what the robot had heard them say:

> *User 6:* I was not always confident that my voice commands had registered.
> *User 8:* I felt like there were lots of bugaboos, especially when using the voice.

Although this is a technical rather than a conceptual issue, it is important because robust voice recognition is not a fully solved problem. And, for robots like the PR2 that do not have wrist-mounted controls, voice commands may be the only modality for giving commands during PbD, since both the user's hands would be occupied while guiding the robot.

Remembering the steps of the workflow

Five users said that they had difficulty recalling the steps for programming an action. However, three users also said that it became easier to remember with greater experience:

> *User 3:* At first it was sort of difficult to understand the order that I was supposed to do stuff in, but after doing a couple of practice tasks, it got very easy.
> *User 5:* Getting accustomed to the workflow... was something to get used to, but it started coming a lot more naturally after the first couple of tasks.
> *User 9:* [What was difficult about *CustomActions* was] making sure not to skip any steps or commands.

Users also needed help with small, but important, details of the workflow. For example, one user got proactive guidance to program the arm to move to the side at the end of a custom action. This was so the arm would not block the robot's view of the workspace in subsequent tasks.

Editing and GUI interface for CustomActions

Two users said that the GUI interface for *CustomActions* was hard to interpret. The *CustomActions* GUI overlaid all the steps of an action in a single view while it was being created (see

Figure 9.4(a)). The GUI also dynamically updated to show actions that were being executed. One user suggested that there should be separate GUI modes for running vs. editing. Additionally, three users had issues with steps being accidentally added or deleted from their action due to misinterpreted voice commands.

> *User 5:* The overlapping opaque graphics made it difficult to see... what was anchored to what reference frame.
> *User 10:* I didn't realize that I had added one last action that released the basket at the end of the program.

Three users received guidance on how to use the GUI to tweak the steps of a programmed action without reprogramming from the beginning. Based on this feedback, we believe it would be valuable to have a "filmstrip" view of a PbD action, in which each step can be visualized and edited individually. Such an interface would also make it more obvious to the user if the system accidentally added or deleted a step.

CustomLandmarks box interface

In the *CustomLandmarks* interface, users had to individually specify the positions of the 6 sides of the box (Figure 9.1 and 9.2). Four of the 10 users said they had some trouble using this interface, although two of them said that it was not a major issue.

> *User 10:* Placing the box in 3D space was surprisingly difficult. I needed to move the camera around a lot before I could place it correctly. *User 9:* I wish I could just drag that box instead of having to expand the edges... Otherwise I think it was pretty simple.

9.5.5 Discussion

Overall, the study found that non-roboticist programmers could learn to use *Code3* to program useful tasks on the PR2. The user study tasks required perceiving and manipulating different

objects, and would be, in our view, non-trivial tasks even for professional roboticists. However, all of the participants, 4 of whom were undergraduate computer science students, were able to program the robot to do at least one of the tasks in under 1 hour after just 90 minutes of training.

The study also revealed various usability issues with the current interface. The workflow of our system was optimized for users to create landmarks, actions, and *CustomPrograms* programs in a linear order. However, we saw that users often deviated from this workflow and needed to edit and remake custom actions and landmarks. The feedback we received suggests a need for a specialized editing GUI for *CustomActions*. This interface would need to make it easy to preview, adjust, and delete poses, and to import previously created landmarks.

9.6 Expert User Programming

This section describes how an expert user (the author) used *Code3* to program two complex tasks. These tasks demonstrate the system's unique perception and flow control capabilities and show that *Code3* is not limited to programming simple manipulation tasks. Below, we describe the tasks, how we programmed them, and how they exercised the unique features of *Code3*.

9.6.1 Tic-tac-toe

The first task we programmed was to play a game of tic-tac-toe with a human player (Figure 9.4(a)). The robot plays the game on a board with red and yellow cylindrical pieces. The robot must pick a game piece from a stack and place it on an empty square on the board. Then, the human player makes a move and presses a button on the robot's touchscreen when done. After that, the robot must examine the board and make another move. The robot must also recognize if the game is over and ask to play again.

We created a custom landmark of the game piece and programmed a custom action for the robot to pick a piece. To pick from the stack, the program used `findCustomLandmark` to locate all the pieces, then it identified the piece with the highest z position and passed it to the picking action (an example of preregistration). We created nine separate custom actions to place the piece in each

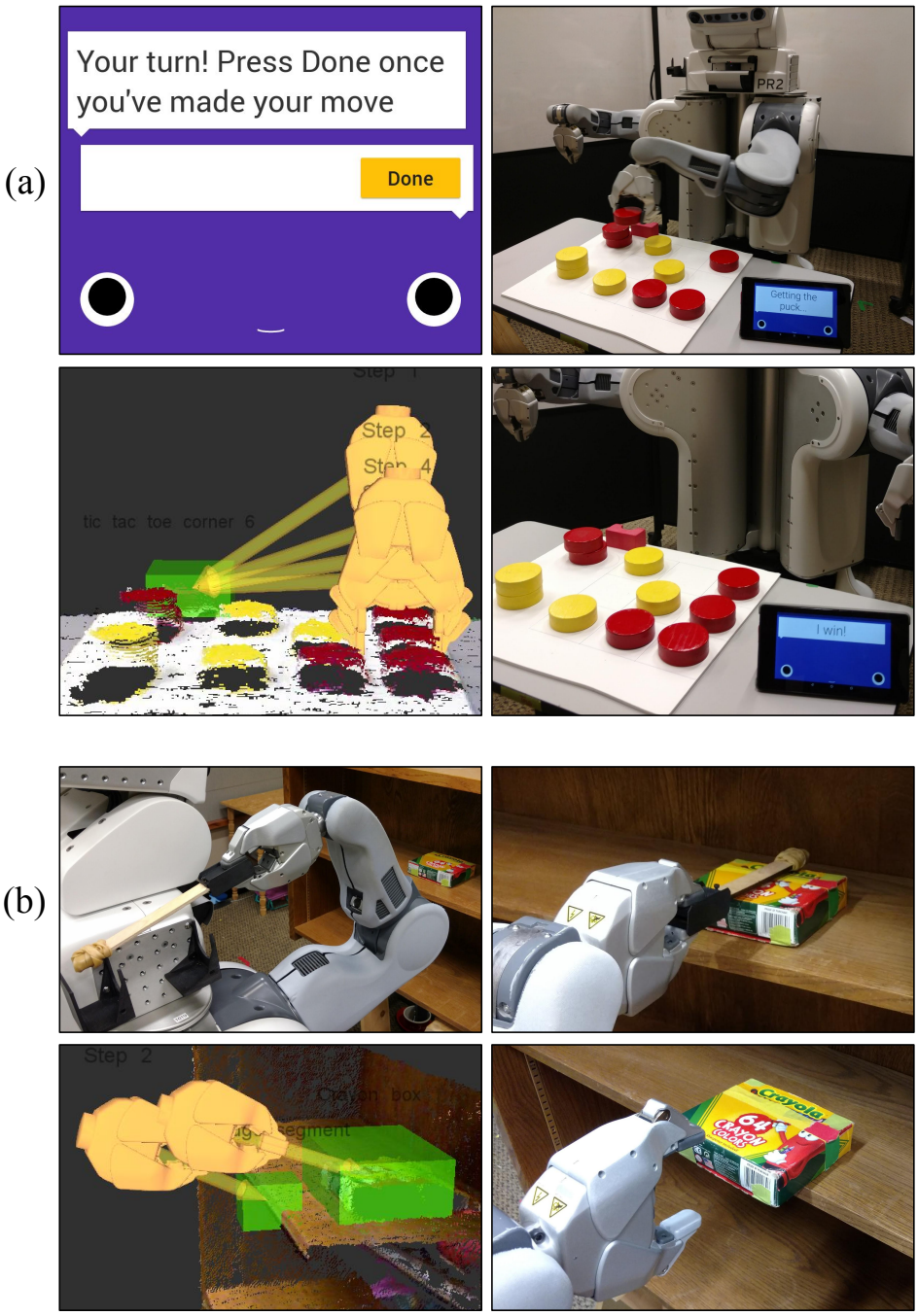


Figure 9.4: Illustrations of tasks programmed by an expert using *Code3*. (a) Tic-tac-toe. (b) A program to reconfigure an otherwise ungraspable object using a tool. See the text for more details.

of the nine squares on the board. For the robot to know the locations of the squares, we stuck a uniquely shaped foam block in the corner of the board. This foam block became a landmark; to place a piece in a particular square, the robot would position its gripper to locations relative to the block.

We took advantage of *CustomPrograms*'s expressivity to program the flow control logic to play the game. To read the board, the robot first searched for the custom landmark of the game piece and the custom landmark of the foam block. It then checked which squares were occupied by comparing the positions of the pieces with the position of the block. The robot used this procedure to infer which previously empty square the human player moved to on their turn. We represented the board with a list of nine strings, which were either "empty," "red," or "yellow." Because *CustomLandmarks* does not use color information, the robot tracked which pieces were red or yellow based on whose turn it was when the piece was placed on the board. The robot played to empty squares at random.

Figure 9.4(a) illustrates part of program. The top left image shows the touchscreen display used to facilitate turn-taking with the human player. The top right image shows the robot grasping a new game piece from a stack. The bottom left image shows the tic-tac-toe corner landmark highlighted with a green box. The yellow arrows show gripper poses offset from this landmark, which were used to place the game piece on the board. The bottom right image shows the updated state of the board.

It took the author 3 hours and 23 minutes to program and test this task, with actions, landmarks, and game-playing logic created from scratch.

9.6.2 Picking challenge

For the second task, the author programmed the robot to pick a box of crayons from a shelf (Figure 9.4(b)). The box was easy to grasp while standing upright, but impossible for the robot to directly grasp while lying flat. We used *Code3* to program a maneuver with a tool to make the box graspable.

We built a simple tool, a 25cm wooden ruler with a handle on one end and a high-friction

rubber tip on the other, to drag the box over the edge of the shelf if the box was lying flat. We used *CustomActions* to program actions to get the tool from a shoulder holster and to place it back.

For this action to work robustly, the robot had to drag the box the right distance over the shelf edge. If dragged too far, the box would fall out of the shelf, but if not dragged far enough, the box would remain ungraspable. Pulling the box a fixed distance would not work since the box's distance from the shelf edge could vary. To address this, we created two custom landmarks: one for the box and another representing a segment of the shelf edge. We used the box landmark to know where to make contact between the tool and the box. However, when pulling the box, we used the shelf edge as the landmark so that the pulling action would end at a consistent distance to the shelf edge.

This action illustrated an interesting use of the preregistration feature. We wanted to, as much as possible, drag the box perpendicular to the shelf edge. To accomplish this, we made the landmark of the shelf edge represent a small, 6 cm segment, which *CustomLandmarks* would find at multiple locations along the shelf edge. We then iterated through the list of segment locations and selected the segment that was closest to the box's location. Dragging towards the closest shelf edge segment ensured that the box was dragged perpindicular to the shelf edge.

We programmed the robot to grasp the crayon box directly if it was standing upright. If the box was lying flat, then the robot got the tool, dragged the box as described above, replaced the tool, and finally grasped the box.

Figure 9.4(b) illustrates this program. The top left image shows the robot grasping the tool. The top right image shows the robot placing the tool in contact with the crayon box. The bottom left image shows the crayon box landmark and the shelf edge landmark, both highlighted in green. The yellow arrows show gripper poses offset from these landmarks, which were used for the PbD action that dragged the crayon towards the shelf edge. The bottom right image shows the crayon box in its final position before being grasped.

This task took us 2 hours and 28 minutes to program and test. For the experimenter to finish making the program, we required that the robot successfully pick the crayon box five times in a row: twice with it standing upright and three times with it laying flat on the shelf. A first draft of

the program was finished within 90 minutes, but we spent the remainder of the time testing and refining the dragging action. It took time to refine the program because for each refinement we made, we had to test the program by running it from the beginning. An improved, step-by-step editing interface for *CustomActions* could make developers more efficient for tasks like these in the future.

Chapter 10

PBJ: PROGRAMMING MANIPULATION TASKS USING RGBD VIDEO DATA

In this chapter, we examine an alternative approach to robot programming for non-expert users: programming by imitation. Although we have developed simplified interfaces that allow non-roboticist programmers to easily program mobile manipulation tasks, some users may be less experienced, or they may not want to go through the effort of developing a program. Instead, many researchers have long envisioned developing robot systems that can learn to perform tasks simply by watching a human demonstrator. This is a difficult problem for a number of reasons. First, the robot must have accurate perceptive capabilities when watching the demonstration, including tracking the motion of the demonstrator and of the objects. Second, the robot must be able to understand the intent of the demonstration, which is a broad and open-ended problem in artificial intelligence research. Third, once the robot understands the goal of the demonstration, it must successfully adapt the human demonstration to its own morphology. For example, it must adapt the person's grasps to its own gripper design and synchronize the motions of its arms, which move at different speeds compared to a person.

In our research, we developed a system called *PBJ* that works toward this vision. In keeping with the motivation described above, the goal of the system was to allow users to program a robot to perform manipulation tasks by simply recording an RGBD video demonstration of the task. Rather than try to develop a fully autonomous system that can understand the video demonstration without making any mistakes, we try to strike a practical balance, by providing users with a semi-automated annotation interface. This interface does most of the object and person tracking automatically, but gives users the ability to make corrections and re-initialize the tracking as needed.

The output of this annotation implicitly specifies a tabletop object manipulation task for the

robot to perform, using some simple rules to convert the video into a motion specification. These rules were designed such that natural motions by the demonstrator result in reasonably similar motions by the robot. By restricting the scope of the system to short-duration tabletop manipulation tasks, we can more easily design rules that match the user's intent.

PBJ was implemented on the PR2 robot, which has two arms and parallel-jaw grippers on each arm. Given the motion specification, *PBJ* generates object grasps that are stable for the robot, while also attempting to match the grasp used by the demonstrator. Our system is capable of simultaneously tracking the motions of the person's left and right arms, and it varies the speed of its left and right arms to properly match bimanual motions by the demonstrator.

Despite its many differences from other systems described in previous chapters, *PBJ* utilizes many of the same concepts described in our conceptual framework (Chapter 3). For example, it uses the same world modeling approaches described in our framework. And, as with other systems in our research, *PBJ* allows motion specifications to be parameterized by landmark locations. We did not integrate *PBJ* into a task scripting system like *CustomPrograms*, but doing so would be straightforward. *PBJ* is available as an open-source system and can be found at `https://github.com/jstnhuang/task_perception`.

In the rest of this chapter, we describe the system in detail. First, we describe the scope of the system and the video recording procedure. Next, we describe a hand segmentation system that we developed, as well as the system for annotating the video demonstrations. After that, we describe how we generate the motion specification from the annotated video. We then discuss how *PBJ* locates objects in new scenes and generates grasps for these objects. The last part of the system description presents how the system plans to accomplish the desired motions. Finally, we present an evaluation of the system, in which we used the system to program several tabletop manipulation tasks.

10.1 Scope of the system

This section first describes some of the assumptions and limitations of our system. As we stated earlier, the system is designed to let a robot imitate a tabletop manipulation task. We assume that

the objects used in the task are known, *i.e.*, a 3D model exists for each object. The 3D models must be in mesh format, which can be obtained through CAD models or 3D scanning techniques [95, 135].

When recording a video demonstration, we require that the user stand across a table from the camera, and that the camera be placed in roughly the same pose that a robot's camera might be in when imitating the scene. However, the demonstrator does not need to wear special tracking devices or add fiducial markers to the scene.

We assume that the programmed action intends to specify the motion of objects relative to objects For example, the system can be used to program a pouring motion of a cup relative to a a bowl, but it cannot be used to program an arm-waving motion, since there are no objects involved in arm-waving. Based on these assumptions, the system assumes that if a demonstrator moves an object close to another, the goal of the action is to replicate the trajectory of the held object relative to the other object. In our cup-pouring example, if the user moves a cup close to a bowl and tips the cup over, the generated program is to move the cup close to the bowl, then to tip the cup over as demonstrated. Aside from replicating object trajectories, the system does not model non-prehensile actions or actions with non-rigid objects.

Additionally, we only model motions where both hands are moving relative to a stationary object (*e.g.*, one hand pours a cup into a skillet while the other hand stirs the skillet) or motions where one hand is holding an object still while the other hand moves relative to that object (*e.g.*, one hand stabilizes a box while another hand pushes an object against the box).

10.2 Hand segmentation

Before we can discuss the video annotation component of *PBJ*, we must first describe one of its components: hand segmentation. When analyzing the demonstration video, we need to be able to detect when the demonstrator has grasped or ungrasped an object. To accomplish this, we track the demonstrator's hands using a visual hand segmentation algorithm. Given an RGBD image, our algorithm labels each pixel in the image as either being part of a hand or not. We can then project the segmentation into 3D coordinates and determine how many hand pixels are touching an object.

Figure 10.1: A sample set of raw images for our hand segmentation dataset. The left image is the color image, the center image is the depth image, and the right image is the thermal camera image.

Pixel-wise segmentation of an image is known as *semantic segmentation* [80]. Recently, researchers have directly inferred hand poses from RGB or RGBD images [32, 101, 154]. We chose to use semantic segmentation for two reasons. First, our goal was to infer grasp events from video demonstrations, which does not require detailed information about the hand pose. Second, semantic segmentation is more a more mature technique, which made it easier to incorporate and maintain in a large, integrated system like *PBJ*. Our semantic segmentation model is based on the single-frame network described by Xiang et al. [147].

10.2.1 Data labeling

To train the model, we generated a dataset of RGBD images with corresponding label images. The label images were simple black and white images where a white pixel indicated a hand pixel.

We mounted a thermal camera next to an ASUS Xtion Pro RGBD camera and recorded 23,928 frames of video. These images all showed one of two individuals moving his hands and manipulating objects in an office setting. A sample set of raw images is shown in Figure 10.1. To label

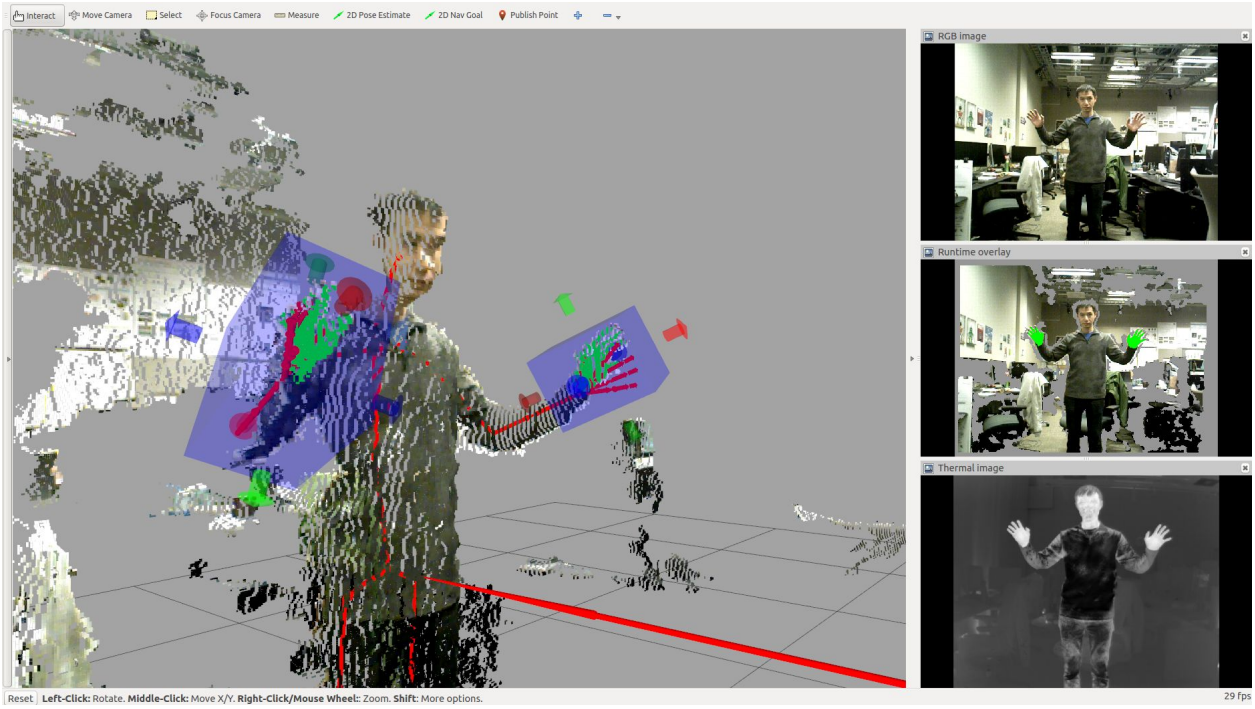


Figure 10.2: The data labeling interface for the hand segmentation model.

the images, we developed a custom data labeling interface, shown in Figure 10.2. In the thermal camera image, exposed parts of the person's body, such as the hands, arms, and face showed up as brighter pixels, while objects held in the person's hand and sitting in the background had darker pixel values. Our approach was to threshold the thermal image based on pixel brightness, while using the annotation interface to distinguish between the person's hands and the rest of his body.

Isolating hands

In the data annotation interface, we manually inspected each frame of video that we recorded. The interface used a skeleton tracker to roughly locate the person's hands. This skeleton tracker was the same tracker that we used in the *PBJ* video annotation interface (Section 10.3 below). The interface presented two bounding boxes, one around each hand. The dimensions of the bounding box could be adjusted by the annotator, which was necessary because the person's hands often

changed size as they spread their fingers or bent their wrists. The bounding boxes were used to isolate the pixels belonging to the person's hands from other bright areas of the thermal image, such as the person's forearm. It was also useful for separating the hand from other bright areas of the thermal image based on depth, such as when the person held his hand in front of his face.

Other uses of the interface

The interface was also used to inspect the quality of the alignment between the thermal camera data and the RGBD image. In some cases, the data from the thermal camera and the RGBD camera did not line up well due to imprecision in our calibration, as well as imprecise temporal alignment between the two camera streams. During the labeling process, we discarded image frames where we found the data did not align well.

10.2.2 Evaluation of the hand segmentation model

We split our dataset randomly with 80% in a training set and 20% in a validation set. Our dataset contains heavy class imbalance. Only 0.78% of pixels are hand pixels, meaning a trivial algorithm that predicts all negatives would have an accuracy of 99.22%. Instead, we measured the intersection over union (IOU) score. If tp is the number of true positive pixels, fp, false positives, and fn, false negatives, then IOU is:

$$\frac{tp}{tp + fp + fn} \tag{10.1}$$

In other words, IOU is not affected by the number of true negatives (e.g., non-hand pixels), and in our example, a trivial algorithm that predicts all negatives would have an IOU score of 0%.

Our model labeled hand pixels with an IOU score of 66.46%. However, we found that the output tended to be blobby and lose fine details like fingers. This is likely due to upsampling operations that take place in the neural network architecture. Previous work has shown that this issue can be ameliorated by adding skip layers to the network [80]. We found that by simply applying an erosion operation [19] with a 3x3 kernel, the IOU of hand segmentation improved to

72.64%.

Example outputs are shown in Figure 10.3. Code for training and running our hand segmentation model and a link to our pre-trained model can be found at `https://github.com/jstnhuang/skin_segmentation`.

10.3 Video tracking and annotation

Once the user has recorded a video demonstration of a task, our system needs to analyze the video to understand the motion of the demonstrator's hands and of the objects. Our system gives users an annotation interface that uses automated tracking tools to understand the body pose of the demonstrator, the poses of the objects, and the locations of the demonstrator's hands. However, these trackers may not function perfectly all the time, and, as soon as a tracker starts to produce incorrect results, it tends to continue to produce incorrect results. As a result, we also want to provide tools that make it possible for users to supervise the tracking tools, and provide corrections when necessary. In this section, we describe the annotation tool we built for this purpose.

The annotation tool takes in RGBD video and outputs features describing the demonstration for each frame. The features we use are:

- The poses of the demonstrator's left and right wrist. We chose to track the demonstrator's wrist because this tended to be more accurate compared to tracking the pose of the demonstrator's hand.

- The poses of all the objects in the demonstration.

- The name of the object being held in the left and right hands, if any.

The tool tracks objects using a particle filter-based tracker by Wüthrich et al. [142], and the demonstrator's pose is tracked using a body tracker by Walsman et al. [131]. The tool automates most of the tracking, but user input is required at the beginning of a demonstration. Additionally, the user can make adjustments to the skeleton or object tracking if the tracking systems drift. Screenshots of the annotation tool are shown in Figure 10.4.

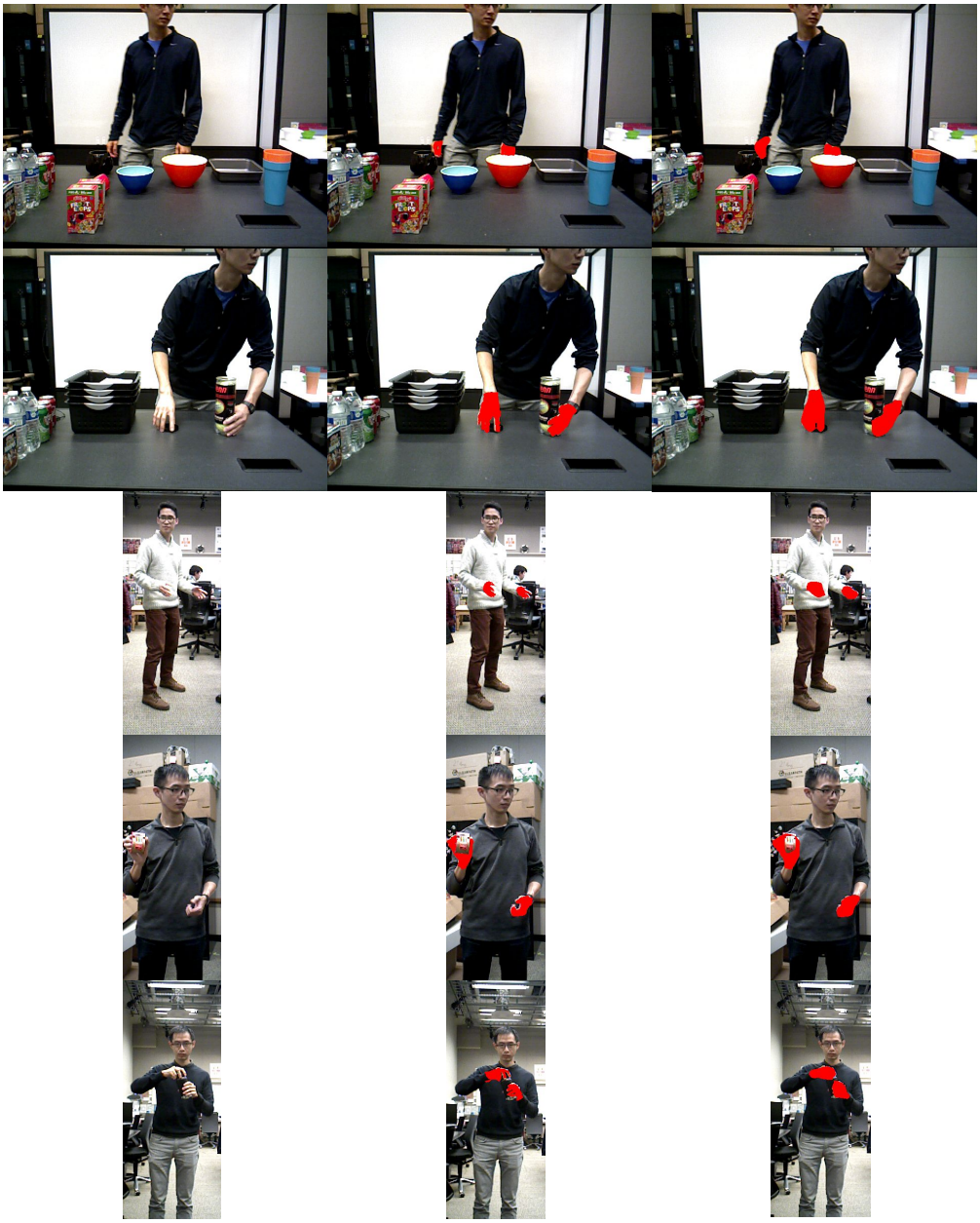
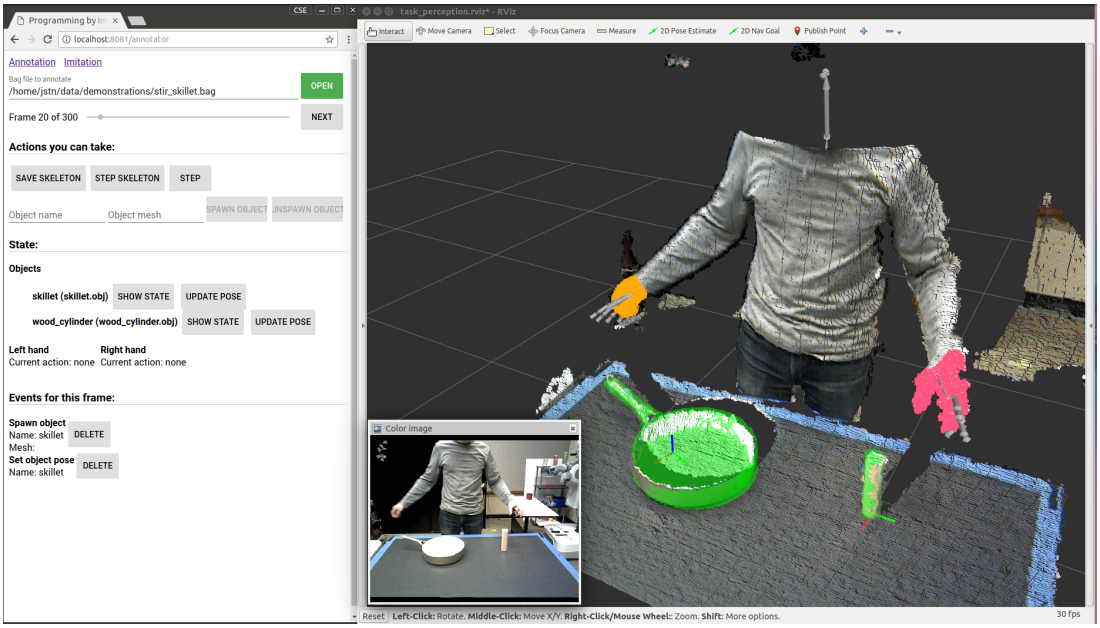
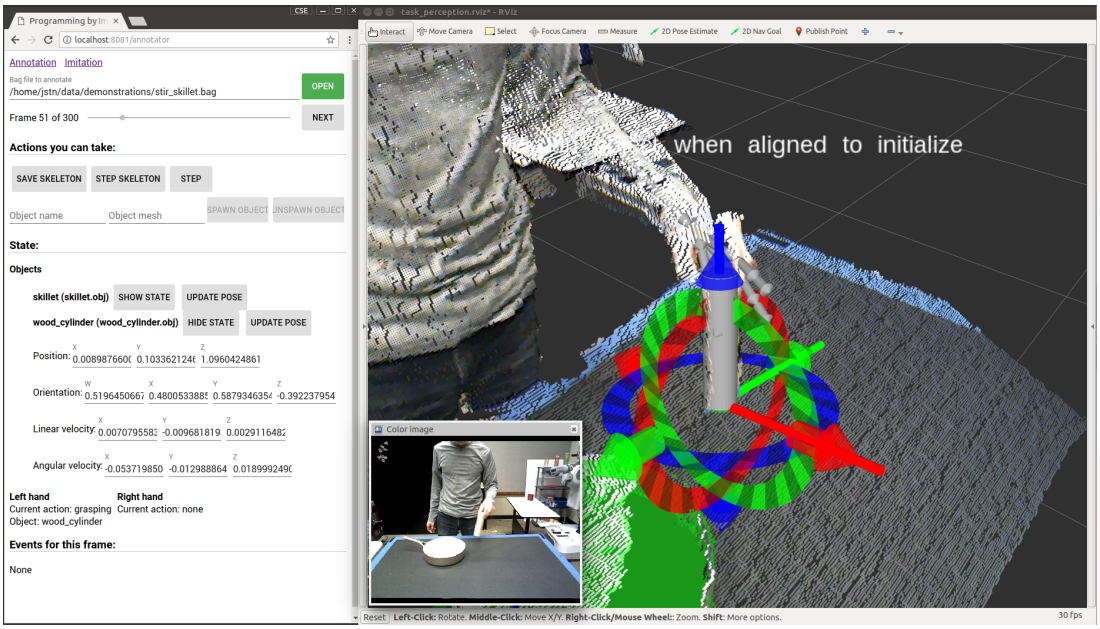


Figure 10.3: Example images from the hand segmentation dataset. The first column shows the color image recorded by the RGBD camera. The second column shows the labels with hand pixels in red. The third column shows the output of our system on the RGBD image (depth images are not shown in this figure).



(a) For each frame of the demonstration video, the annotator automatically tracks the object poses (green), the skeleton of the demonstrator, and the pixels corresponding to the left and right hands.



(b) The user can manually adjust the automatic object tracking result if needed.

Figure 10.4: Screenshots of the task annotation interface.

To initialize an annotation, the user must load an object model for each object in the scene and align the model to the object's initial pose. This is used to initialize the object tracker. Additionally, the user gives each object a unique name, which is used to identify the object throughout the system. To initialize the human body tracker, we require the demonstrator to start the demonstration by standing in a "start pose," in which the demonstrator stands with arms stretched to the side.

To annotate the video, the user steps through each video frame in the annotation tool. The tool runs the hand segmentation algorithm described above to label the hand pixels in each frame. Hand pixels that are too far away from one of the demonstrator's wrists are ignored, and pixels are assigned to the left or right hand, depending on which is closer. For each hand, if the number of hand pixels touching an object exceeds a threshold, the tool considers the demonstrator to be grasping that object. Inversely, if the number of hand pixels touching an object falls below a threshold, the tool considers that an ungrasp event.

The interface provides a visualization of all the tracking outputs. If the outputs are all correct, the user clicks a button to advance to the next frame. Otherwise, the user can manually adjust the joint poses output by the skeleton tracker or the pose of an object. If the user makes a correction, the corrected data is used to reinitialize the skeleton and/or object tracker. Once the last frame has been annotated, our tool saves the annotation data to the disk.

10.4 Motion specification from video

In this section, we describe how *PBJ* infers a motion specification from the annotated video data, a process we call *program generation*. *PBJ*'s program generator is a rule-based system that aims to produce reasonable programs in response to natural motions by the demonstrator. Its input is an annotated demonstration, and its output is what we simply refer to as a *program*.

A program is a list of *steps*. There are four types of steps: *grasp*, *ungrasp*, *move-to*, and *trajectory*. Each step has a start time and applies to either the left or the right arm. A *grasp* step specifies a grasp pose relative to an object. In a *ungrasp* step, the robot will open its gripper. A *move-to* step specifies a gripper pose relative to an object. Finally, a *trajectory* step contains a gripper trajectory, represented as a sequence of gripper poses relative to an object.

To generate the program, we iterate through the frames of the annotated demonstration video and add steps depending on changes to the state. When the demonstrator's hand state changes to a grasping state, we add a *grasp* step to the program. The representation of a grasp step includes the pose of the demonstrator's wrist relative to the pose of the object being grasped. Later, when the robot executes the program, our grasp planner will use this information to help plan a grasp. We add an ungrasp step whenever the demonstrator's hand changes from an grasping state to an empty state.

Next, we discuss how we add *move-to* and *trajectory* steps. Recall that the input to the program generator includes the trajectories of all the objects for the entire demonstration. The program generator segments the trajectories based on whether the demonstrator moves an object close to another object. When the demonstrator moves an object close to another object, we call this an *object interaction*. The object in the demonstrator's hand is the *held object* and the nearby object is the *landmark* object. When the demonstrator moves an object and the object is not close to any others, we call this a *free space movement*. When we transition from a free space movement to an object interaction, the program generator adds a *move-to* step, which specifies the pose of the held object relative to landmark object at the moment of transition. If the demonstrator moves an object and releases it without getting close to any other objects, the landmark of the *move-to* step is the held object's initial pose.

While the held object is close to the landmark object, we record the pose of the held object relative to the target object and save the poses in a *trajectory* step. An important limitation to our system is that we assume the landmark is stationary. As a result, we do not handle demonstrations where the demonstrator holds an object in each hand and moves both objects relative to each other simultaneously. However, one object can be held still while the other object moves. For example, the demonstrator can stabilize a bowl with one hand while stirring its contents with a tool held in the other hand. However, the demonstrator cannot move the bowl while simultaneously stirring it. Both hands can move objects independently relative to one or more stationary objects. For example, one hand can pour a cup into a skillet while the other hand stirs.

When the program generator encounters two held objects moving close together, this is called a

double interaction. When a double interaction is first detected, the generator looks ahead at future video frames to see which object moves the least during the double interaction. This object is considered to be the landmark object. This can be used to detect situations where one hand is used to stabilize an object while the other hand performs an action.

Once the robot processes the last frame of the video, it saves the program to a database. The end of the program generation is the dividing line between *offline* and *online* phases of the system. The offline processing phase is when the robot converts a demonstration video into a motion specification. In the runtime phase, the robot is asked to execute the motion specification in a new scene. This is described in the sections below.

10.5 Object pose initialization

Before *PBJ* can execute an action, it needs to know the initial poses of the objects used in the task, a process we refer to as *object pose initialization*. Our system attempts to perform object pose initialization autonomously as described below. However, before executing the task, we allow users to adjust the initial object poses if needed.

To locate the objects, we are given a point cloud of the scene and a model database of the objects used in the demonstration. Our goal is to find the correspondence between objects present in the given scene and the objects used in the demonstration. The system must discriminate between different types of objects used in the demonstration, *e.g.*, it might need to determine which of two objects is a cup and which is a bowl. However, it must also be able to discriminate between objects of the same type. For example, in some demonstrations, there may be two instances of a particular box in the demonstration, one on the left and one on the right. To solve this problem, we first match each object with a list of candidates that match by object type. Then, we choose from this list by seeing how well the candidates match the initial position of the object. Below, we describe these two steps in further detail.

Initially, to locate the objects, we segment the tabletop scene using the algorithm described in Chapter 5. This algorithm fits an oriented bounding box around each tabletop object, giving us its pose and the x, y, and z dimensions of the box. It also outputs the same information for the

visible portion of the tabletop surface. For each object in the demonstration, we iterate through the list of detected tabletop objects to try to find a match. We compare the dimensions of the two objects and add any object that has at least two matching dimensions, within a certain tolerance, to a list of candidates. The tolerance is needed to account for noise in the point cloud data. We only require a candidate object to have two matching dimensions instead of three to account for an error in how depth cameras perceive thin parts of objects. Thin edges of objects such as bowls may not be perceived by depth cameras, especially when the camera is pointed directly at the edge. This causes the object to not be fully constructed in the point cloud, which in turn affects the dimensions of its bounding box. We found that even for correct matches, this error could result in one of the bounding box dimensions being wrong by a large amount (*e.g.*greater than 5 cm), which is why we only require two dimensions to match closely and not three.

If the demonstration has two objects with very different dimensions (*e.g.*, a bowl and a mug), then the candidate list for either object will only have one item, in which case we proceed to the next step. If there are two or more candidates in the list, however, then we must decide which to choose. This situation ideally corresponds to the case in which there are multiple instances of the same object in the demonstration. However, it may also be the case that two different types of objects happen to have similar dimensions. To distinguish between these two cases, we try to determine the "ambiguity" between the best choices. The candidates are each given a score equal to the Euclidean distance between the candidate's bounding box dimensions and the object's bounding box dimensions. The ambiguity of a candidate is the best candidate's score divided by the current candidate's score. If the ambiguity is close to 1, then both candidates have similarly good scores. If the ambiguity is close to 0, then best candidate has a much better (*i.e.*, lower) score than the current candidate. In our system, we filter the list of candidates to those with an ambiguity of 0.15 or higher.

At this point, we have filtered the list based on the dimensions of the candidate objects, but we have not yet taken their positions into account. This is done by the final step of the matching process. Ultimately, we choose the candidate object whose position is closest to the target object's initial position in the demonstration video. One important detail is that we must properly account

for the fact that the pose of the robot's camera relative to the workspace is different from the pose of the camera used to record the demonstration. In the demonstration, the camera must be placed on the opposite side of the table, facing the demonstrator. At runtime, we want the robot to adopt the demonstrator's perspective. When we look at the demonstration video, an object on the left side of the camera image will actually be on the demonstrator's right. Therefore, when comparing the positions of objects at runtime to the positions of objects in the demonstration, we must rotate the poses from the original demonstration by 180 degrees about the center of the workspace. We also must position the cameras in roughly similar poses. When recording demonstrations, we center the camera on the demonstrator and place it at roughly eye level, looking slightly down at the table. This mimics the robot's camera pose when it sees the same scene at a later time.

As we mentioned earlier, we allow the user to make adjustments to the initial object poses if they are incorrect. To do this, they drag the object model in a visualization of the scene until the object model lines up with the data.

10.6 Grasp planning

PBJ does not plan grasps in the offline processing phase. Instead, we wait until we have found the initial poses of all the objects. This is because the initial object poses can affect generated motion, such as when it is placed in an area where the robot has limited reachability. For each grasp step in the program, we run a grasp generator to find a grasp that is likely to succeed. Our grasp planning problem differs from traditional grasp planning because, in our setting, the grasps are more constrained. We require that the grasp remain valid throughout the object's trajectory, and the same object may need to be grasped in different ways depending on how the demonstrator grasped the object. We see one such example later in Section 10.8.2, in which the robot must perform a power grasp of a cup from the side with one gripper while making a more precise pinch grasp of another cup with the other gripper.

Our grasp planner takes the following as input (collectively referred to as the *grasp planning context*):

- The object's pose relative to the robot base
- The trajectory of the object from the time it is grasped until it is ungrasped
- A model of the object (represented as a point cloud with uniform sampling over the model)
- The pose of the demonstrator's wrist relative to the object
- The location and dimensions of the tabletop surface.

Algorithm 3 provides a high-level overview of our grasp planning algorithm. The algorithm consists of multiple helper functions, which we describe in more detail below.

The algorithm essentially consists of two parts: generating grasps and ranking grasps. The grasp generation happens on lines 1 and 2 and lines 5 through 8. On line 1, the algorithm computes an initial grasp based on the position of the demonstrator's wrist at the time of the grasp (explained further below). On line 2, the algorithm samples random points on the object. For each sampled point, lines 6 through 8 compute a grasp that grasps the object at that specific point, using helper functions defined below. The planner then scores the grasp on lines 9 and 10 and outputs the best grasp on line 16.

Next, we describe each of the helper functions used in the grasp planning algorithm. `ComputeInitialGras` is computed once at the beginning of the algorithm to estimate the grasp based on where the demonstrator's wrist was at the time of the grasp event. The function simply places a gripper with the same pose as the demonstrator's wrist, then translates the gripper such that the closest object point is in the center of the gripper's grasp region (the region between the gripper's two fingers). This gives us a good approximation of where the gripper should be, since we want the system to imitate the person's grasp. However, it is generally not a suitable grasp, because the robot's gripper has a different morphology compared to the demonstrator's hand, and because the given wrist pose for the demonstrator may not be completely correct. It is the job of the other helper functions to find a more suitable grasp for the situation.

Algorithm 3: Grasp planning algorithm for *PBJ*

Input : Grasp planning context

Output: Grasp

```
1  initial grasp = ComputeInitialGrasp (context);
2  sample points on the object using a voxel grid;
3  best grasp = null;
4  best score = null;
5  for (sample in samples) {
6      grasp = CenterAndAlignOnSample (initial grasp, sample, context);
7      grasp = OptimizePlacement (grasp, context);
8      grasp = ShiftGrasp (grasp, context);
9      grasp, score = OptimizePitch (grasp, context);
10     grasp, score = EvaluateFuturePoses (grasp, score, context);
11     if (score > best score){
12         best score = score;
13         best grasp = grasp;
14     }
15 }
16 return best grasp;
```

`CenterAndAlignOnSample` shifts the gripper's pose and orientation such that the sampled grasp point is in the center of the gripper's grasp region and the normal vector of the grasp point is aligned with the normal vectors of the gripper fingertips. In Algorithm 3, the gripper's x axis points forward away from the gripper, the y axis points to the left, normal to the gripper's fingertips, and the z axis points up. Aligning the gripper to the normal of the sampled object point helps ensure that the grasp is stable and natural-looking.

`OptimizePlacement` is an iterative algorithm that translates the gripper side to side (*i.e.*, in the y direction) to maximize the number of object points in the grasp region and minimize the number of collisions. To do this, we transform all the points of the object model into the gripper frame. All of the object points in the gripper's grasp region exert a translational "pull" on the gripper that would put that point in the center of the grasp region. When all the translations are averaged, the overall translation vector centers the grasp on the object points. Object points in collision with the gripper are treated similarly, except their translation vectors are given double the weight. The final translation vector is an average of the translation vector for each point. For greater precision, we repeat this process a certain number of times, reducing the magnitude of the translation each time. This is shown formally in Algorithm 4.

This function has the effect of centering the gripper on an object and avoiding collisions between the object and the gripper. For example, if the robot is grasping a cup from the side, then the `CenterAndAlignOnSample` function would center the gripper on one of the side points, meaning that the gripper would penetrate the cup. After running `OptimizePlacement`, the grasp will be centered on the cup as a whole.

`ShiftGrasp` helps the planner distinguish between pinch grasps with the gripper's fingertips and power grasps with the object held in the entire gripper. This function uses the following heuristic: if the object inside the gripper's grasp region is narrow, it uses a pinch grasp, but if the object is wide, it uses a power grasp. For narrow parts of an object like the edge of a bowl or the rim of a cup, we generally do not want the robot to stick its gripper deep into the object, which is why we want to use a pinch grasp. Pinch grasps are also preferable in general because they keep the robot's grippers to the periphery of the workspace, making them less likely to collide with

Algorithm 4: OptimizePlacement

Input : Current grasp, Grasp planning context

Output: Grasp

```
translate object model into grasp frame;
weight = 1;
repeat{
    average translation = 0;
    num points = 0;
    for (point in object model point) {
        if (point is in grasp region){
            average translation += point.y;
            num points += 1;
        }
        else if (point is in collision with gripper){
            average translation += 2 · point.y;
            num points += 1;
        }
    }
    average translation /= num points;
    average translation *= weight;
    translate grasp in y direction by average translation;
    weight *= 0.9;
}
until (convergence or maximum number of iterations);
return grasp;
```

objects. On the other hand, if we are grasping a wide part of the object, a pinch grasp is less likely to be stable, meaning we should use a power grasp.

For a parallel-jaw gripper like that of the PR2 and the Fetch, the difference between the two kinds of grasp is determined by how far inside the gripper we place the grasp point. We define one point near the center of the grasp region to be the power grasp point, where the robot would grasp an object if it were almost as wide as the open gripper, and we defined another point to be the pinch grasp point, where the robot would grasp an object if it were extremely thin. `ShiftGrasp` measures the width of the object points inside the gripper, then linearly interpolates between the power grasp point and the pinch grasp point according to the width of the object.

Formally, let w_{max} be the maximum width of the robot's gripper, and let w be the width of the object inside the gripper's grasp region. Let o be the power grasp point and and i be the pinch grasp point. Then, the grasp point p is:

$$p = \frac{w}{w_{max}}(o - i) + i \tag{10.2}$$

Finally, `ShiftGrasp` moves the gripper such that the sampled object point is at the point p in the gripper.

`OptimizePitch` samples angles for the gripper to pitch about the grasp center. We allow pitch angles to vary, as opposed to roll or yaw angles, because pitching the gripper maintains the alignment between the object normals and the fingertip normals found by the `Align` procedure. The best pitch angle is selected according to the `Score` function described below.

`Score` computes a measure of grasp quality as a weighted linear combination of several features. Four of the features relate points on the object and their normals to the candidate gripper pose. If the normal of the object point is roughly aligned with the normal of the fingertip, it is an *aligned* point. Keeping track of aligned points causes the grasp planner to rate *antipodal* grasps—those in which the normal vectors of the grasp points are collinear and in the opposite direction—more highly. The full list of features is:

- # of aligned object points in the grasp region

- # of unaligned points in the grasp region
- # of aligned object points in collision with the gripper
- # of unaligned object points in collision with the gripper
- Distance between the grasp's wrist pose and the demonstrator's wrist pose
- Whether the gripper is colliding with an obstacle or not

Because the goal of the system is to imitate a demonstration, we put a large weight on the distance between the gripper's wrist pose and the wrist pose of the demonstrator. In essence, the demonstration provides the grasp planner a strong hint as to how to grasp the object. The score function also checks if the candidate grasp collides with obstacles (*i.e.*, the tabletop or another object). Poses in collision are given a very large negative score. The weights for the score function were set by hand, and can be found in our open-source implementation.

After optimizing the pitch of the grasp, the planner then runs `EvaluateFuturePoses`, which evaluates the grasp against the object trajectory for which the grasp will be used. We sample poses from the object trajectory and, for each pose, compute whether an inverse kinematics (IK) solution can be found given the candidate grasp pose. For radially symmetric objects like cans or bowls, we can relax the evaluation. Radially symmetric objects look identical when rotated their z axis (a yaw rotation). So, if an IK solution cannot be found for a radially symmetric object, we make additional attempts to find IK solutions by rotating the grasp pose about the z axis of the object. We append the percentage of poses in the trajectory that have IK solutions as another feature to the grasp's score.

Figure 10.5 illustrates some grasps generated using our grasp planning method.

10.7 From steps to motion plans

As we described earlier, there are four kinds of *steps* in a *PBJ* program: *grasp*, *ungrasp*, *move-to*, and *trajectory*. These steps specify the trajectories of objects in a task. To convert these object

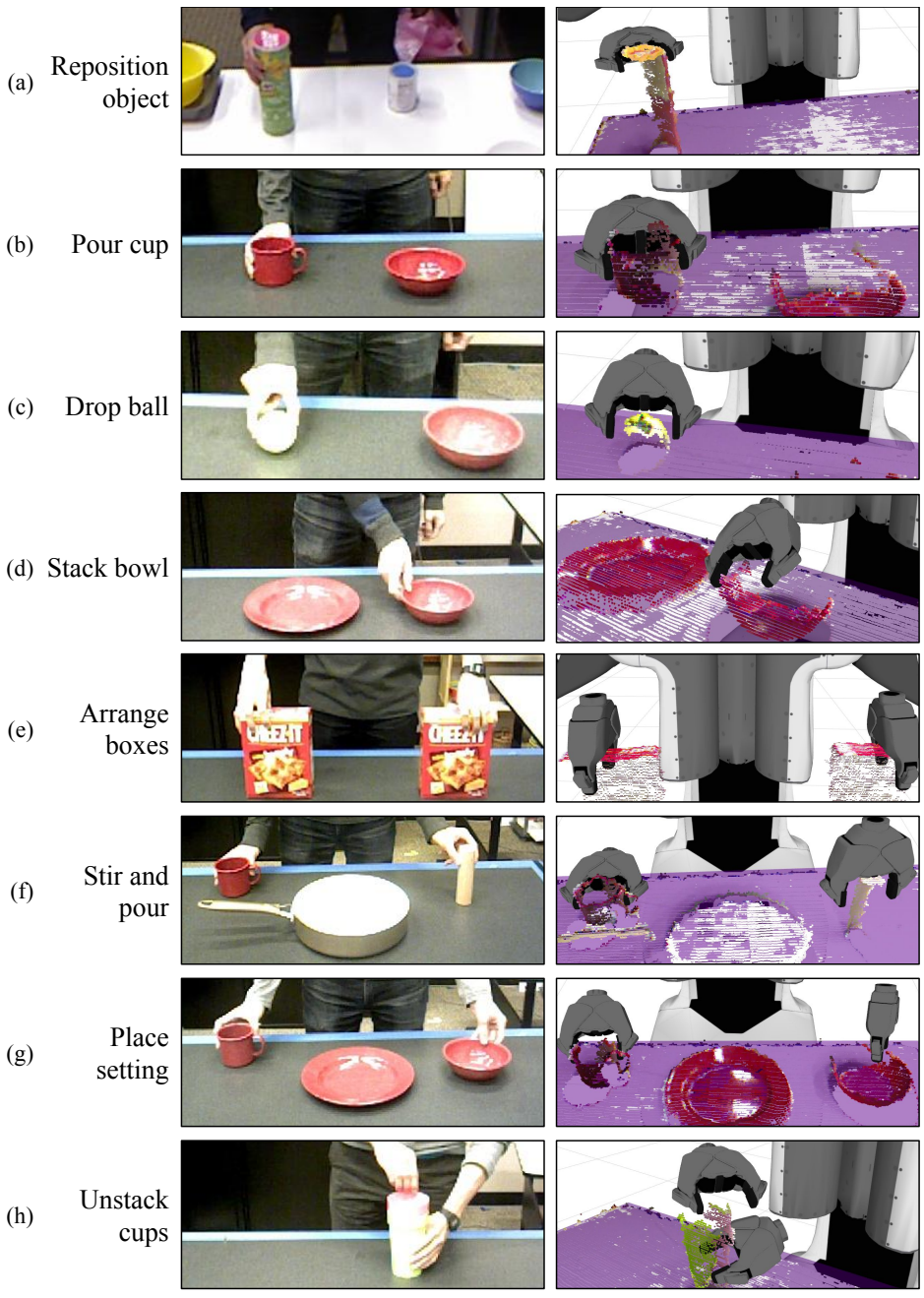


Figure 10.5: Examples of grasps adapted from human demonstrations. Each row comes from a task described in Section 10.8.2. The images on the left show frames from the demonstration videos at the moment the demonstrator grasps the object(s). On the right, we show the grasps planned by our system, which are adapted from the human grasp.

trajectories into robot end-effector trajectories, we simply apply the grasp poses found by the grasp generator described above. We then convert the steps into motion plans that will actually be executed on the robot. In *PBJ*, we treat the general motion planning problem of finding collision-free paths between two robot arm configurations as a black box. To find motion plans to a given end-effector pose, we use the open-source *MoveIt* library.[1]

For *grasp* steps, we take the grasp planner's output and automatically inject a pre-grasp pose before it. The pre-grasp pose is the same as the grasp pose, but translated backward by 10 cm. This value was chosen specifically for the PR2 based on the size of its gripper. Similarly, for the *ungrasp* step, we automatically inject a post-grasp pose 10 cm backward after ungrasping the object. The *move-to* step is straightforward, we simply apply the planned grasp to the specified pose of the object. In all three cases, we first try planning a straight-line motion for the gripper. If this plan is not feasible, then we try using an RRT-Connect motion planning algorithm. If this does not work, then we declare a failure to execute the program.

10.7.1 Trajectory planning

Transforming *trajectory* steps into motion plans is different from the other steps. A trajectory step encodes an end-effector trajectory for the robot to follow. As we did during the grasp planning phase, we check for IK solutions for each gripper pose in the trajectory. For radially symmetric objects, we rotate the grasp about the object's z axis such that the robot's wrist is as far from the center of the scene as possible (e.g., more to the right for the right arm), subject to the constraint that an IK solution exists for the pose. For a robots like the PR2 whose arms resemble those of a human, this makes the trajectory look more natural and human-like. It also helps prevent collisions between the two robot grippers during bimanual motions. Once the trajectory has been determined, we convert the trajectory from gripper poses to joint angles by solving IK for each gripper pose.

---

[1] https://moveit.ros.org

10.7.2 Bimanual trajectory synchronization

Throughout the planning process, the timing of the trajectories came directly from the demonstration. However, each robot will have different velocity limits for its joints, which make the maximum speed of its arms different from that of a human demonstrator. In our particular implementation for the PR2, the robot moves more slowly than a person does. We can retime a trajectory to obey the robot's velocity limits. However, for motions involving both arms, we cannot independently retime each arm's motion, because they will fall out of sync.

As an example, consider a task in which one arm holds a bowl still while the other arm stirs its contents. The arm holding the bowl does not move and can trivially finish the "trajectory" of staying still in a short amount of time. The arm doing the stirring, however, must move more and will take longer to complete. Our system will additionally slow down the trajectory in order to stay within the robot's velocity limits. The result is that the arm holding the bowl will let go after a short amount of time, while the other arm continues to stir.

To address this issue, we describe a procedure called *slicing* below. Slicing synchronizes the trajectories of the two arms before we retime the motions to stay within velocity limits. Each step has a start time and takes a certain amount of time to execute. As a result, we can think of the steps as intervals on one of two timelines, one for the left arm and one for the right. To synchronize the steps of the two arms, we define the beginning and end of each interval to be *synchronization points*. We then walk through the two timelines to merge them into a single timeline. Each interval on the merged timeline begins at one of the synchronization points and ends with the next synchronization point in either timeline. We call these intervals *slices* because a single step may be cut into multiple pieces during this procedure. Each slice stores the trajectories for both the left and right arms for the time period covered by the slice. This is illustrated in Figure 10.6.

Once we have generated the slices, we independently retime the trajectories within each slice so that the arms conform to the robot's joint velocity limits. To execute the program, we send the trajectories in each slice to the robot's low-level controllers, which moves the arms. At the beginning of grasp or ungrasp steps, we open or close the gripper, respectively.

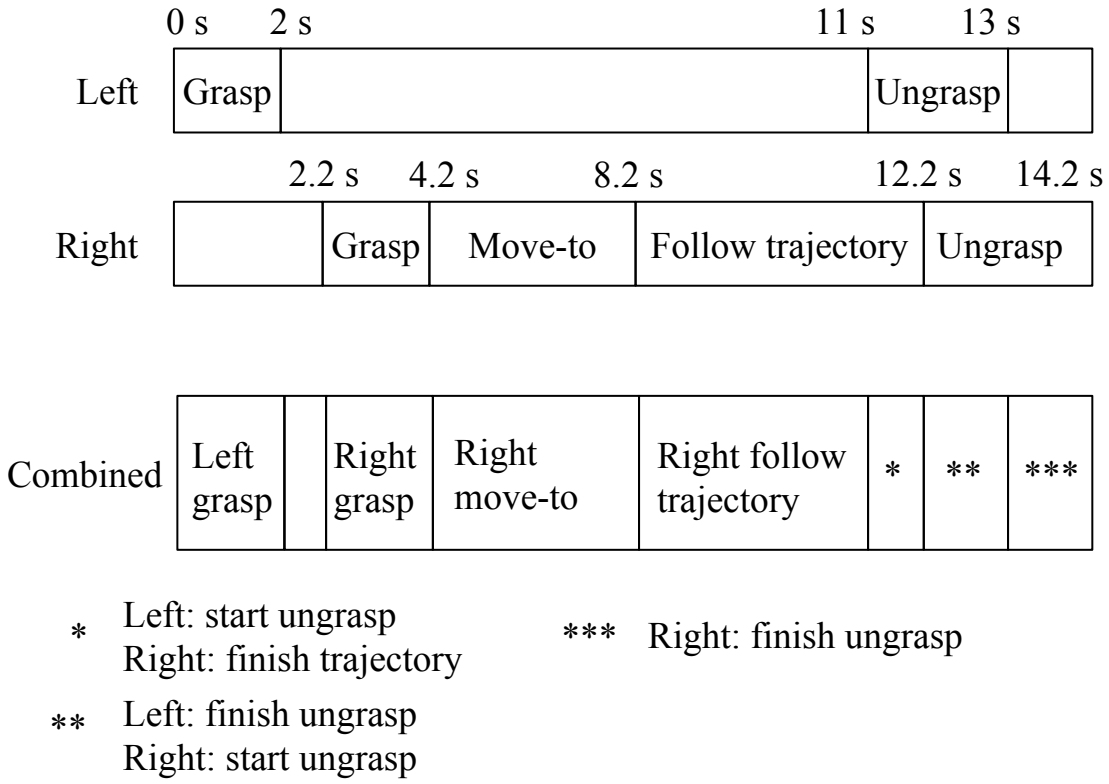


Figure 10.6: An example of synchronizing two-arm trajectories by "slicing." The figure shows three parallel timelines: one for the left arm, one for the right arm, and, at the bottom, the merged timeline. The steps each arm will execute are shown horizontally in the timeline. For example, the left arm executes a grasp step from 0 to 2 seconds and an ungrasp step from 11 to 13 seconds. In the combined timeline, the rightmost three slices are truncated due to space limitations. The asterisks point to the contents of the truncated slices: * represents the slice from 11 to 12.2 seconds, ** represents the slice from 12.2 to 13 seconds, and *** represents the slice from 13 to 14.2 seconds.

10.8 Evaluation

In this section, we present an evaluation of the system's ability to program a variety of tasks and successfully execute them. In our evaluation, we programmed a set of 8 manipulation tasks with different kinds of motions and recorded the success rate of the actions across multiple trials. We chose the tasks to exhibit a variety of motions, including bimanual motions. We also ran five trials of each task with objects in different starting configurations, and counted the number of successful executions.

10.8.1 Procedure

The experiments took place in an indoor laboratory environment with the Willow Garage PR2 robot. A demonstrator recorded video demonstrations using an ASUS Xtion PRO Live RGBD camera. The author annotated the demonstrations. We recorded the number of times we adjusted the pose of an object and the number of times we adjusted the skeleton tracker output. To evaluate the success rate of the programs, we created paper templates and drew in outlines for the starting positions of the objects. We defined one object configuration to be a "natural" configuration, in which the placement of the objects was similar to that of the original demonstration. For the other four trials, we translated the starting positions of the objects. The minimum amount of translation across all objects and trials was 1.8 cm, the maximum was 26.7 cm. Data for all objects and trials are given in Table 10.1.

10.8.2 Task descriptions

We programmed and executed 8 manipulation tasks using the system, four of which used just a single arm and four of which used two arms at the same time. Below, we describe each of the tasks, how they are unique, and how we defined success for each task. Figure 10.7 depicts the tasks we evaluated.

Task	Object	Trial 2	Trial 3	Trial 4	Trial 5
Reposition object	Can	11.6	8.8	9.3	2.2
Pour cup	Bowl	6.8	8.6	9.0	9.2
	Cup	8.8	12.6	8.9	9.7
Drop ball	Ball	16.4	18.2	16.0	16.0
	Bowl	12.5	24.4	18.8	13.2
Stack bowl	Bowl	10.8	11.9	12.2	11.4
	Plate	5.1	3.2	9.0	14.1
Arrange boxes	Box 1	12.0	16.2	21.5	26.7
	Box 2	14.0	11.1	12.8	14.1
Stir and pour	Skillet	2.2	1.8	4.9	3.8
	Cup	5.3	6.0	9.3	4.0
	Wood cylinder	4.0	10.0	11.3	6.9
Place setting	Bowl	3.3	4.7	5.2	6.0
	Cup	8.0	7.2	16.5	17.6
	Plate	5.9	6.8	4.3	3.5
Unstack cups	Cups 1 & 2	10.0	14.6	13.2	10.7

Table 10.1: Object offsets between trials of the *PBJ* task evaluation. Each task was run 5 times with different starting positions of the objects. This table shows, for Trials 2-4, how far an object's starting position moved compared to its starting position in Trial 1. Units are in centimeters.

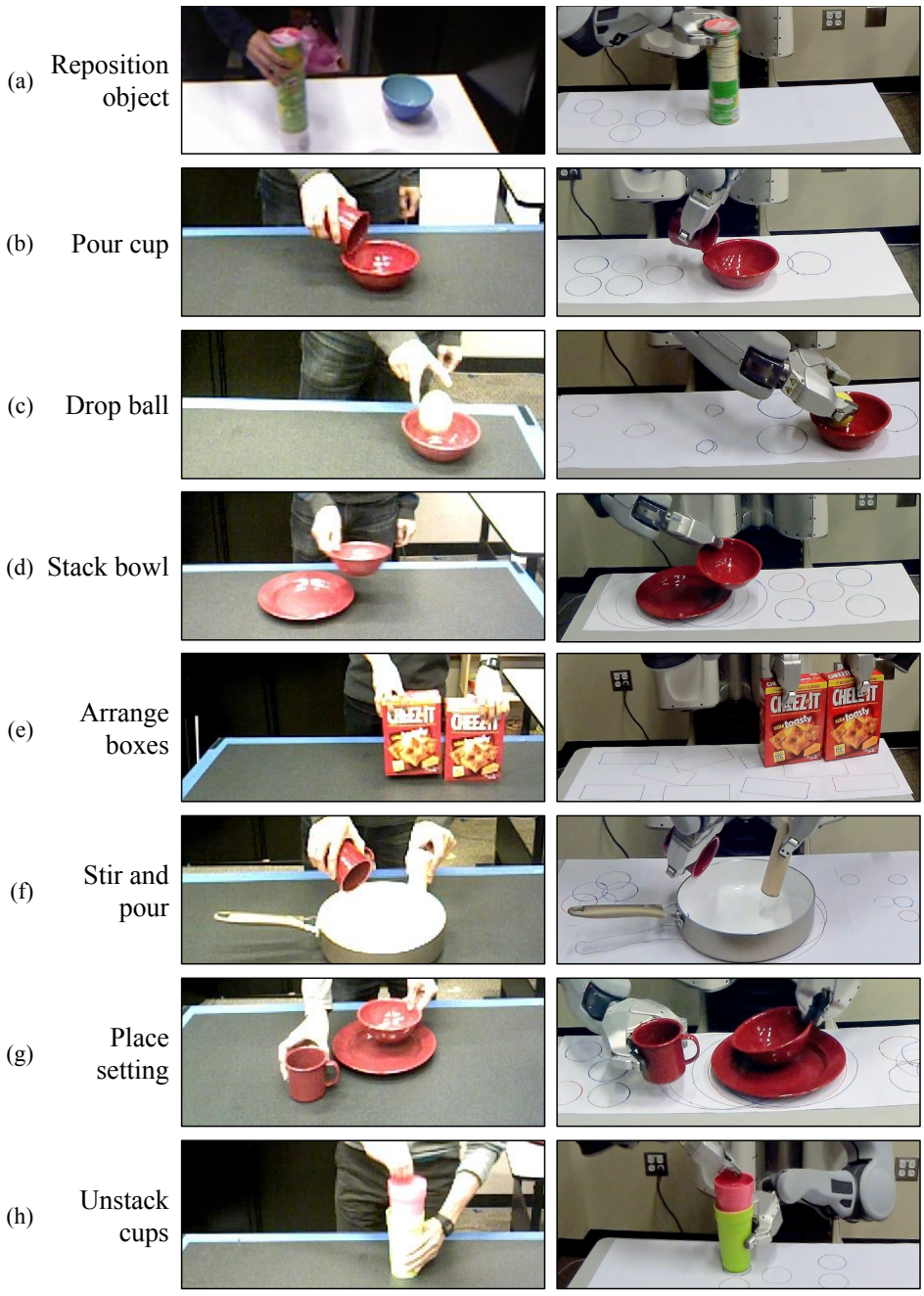


Figure 10.7: Tasks we programmed and evaluated with the *PBJ* system. The left image of each row shows a still frame from the demonstration video, and the right image shows the robot executing the task at a similar point in time. We describe the tasks in detail in Section 10.8.2.

Reposition object

In this task, the robot picks up an object and places it in a new location. Although simple, this task tests a special case in our system. As described in Section 10.4, *PBJ* generally assumes that object movements are specified relative to some other object. However, if there are no nearby objects to be the landmark, we consider the landmark to be the object's initial pose. In other words, the generated program simply shifts the object's pose relative to its initial pose. In this task, the object was a canister of chips. A trial was successful if the robot successfully grasped the object and moved it from one side of the table to the other.

Pour a cup into a bowl

In this task, the robot picks up a cup and pours it into a bowl. This task demonstrates a trajectory-following action. Once the robot moves the cup near the bowl, it follows the human-demonstrated trajectory of tipping the cup into the bowl. Additionally, it shows how our system can choose a grasp based on human demonstration. In the absence of task-specific information about the cup, a simple way to grasp the cup is from the top, sticking one gripper finger inside the cup. However, because the demonstrator grasped the cup from the side, the grasp planner should choose a similar side grasp of the cup. A trial of this task was successful if the robot successfully grasped the cup from the side, executed the pouring motion, and placed the cup back down on the table.

Drop a ball

In this task, the robot picks up a tennis ball and drops it into a bowl. There are two unique features to this task. First, because the ball is spherical, the automatic object tracker cannot correctly determine its orientation throughout the demonstration. In this case, we made use of the task annotator tool to prevent the object from pitching or rolling. This task is also an example of an ungrasping action that is detected by dropping an object. To detect the ungrasp event, the annotator must track the ball as it falls and realize that the demonstrator's hand is no longer grasping it. A trial of this action was successful if the robot successfully picked up the ball and dropped it into

the bowl.

Stack a bowl

For this task, the robot picks up a bowl and places it on a plate. One unique feature of this task is that the demonstrator showed a cross-body motion. In the demonstration, the bowl was on the demonstrator's left, and the demonstrator grasped it using his right hand. Another feature of this task is that we wanted the robot to grasp just the bowl's rim rather than place its fingers deep inside the bowl. A trial of this task was successful if the robot successfully placed the bowl on the plate.

Place an object next to another

This task was similar to the *reposition object* task, in which the robot moves a box to place it in contact with another box. The unique feature of this task is that it requires the robot to hold one box steady with one arm while the other arm pushes the other box against it. A trial of this task was successful if the robot successfully pushed one box against the other without moving the steadied box.

Stir and pour

This task simulated a cooking action in which one arm pours a cup into a skillet while the other arm stirs the skillet with a wood cylinder. This task is unique because it shows the arm executing two trajectory-following actions simultaneously. The two arms also come in close proximity over the skillet in this action. Another unique feature of this task is that the robot can avoid collisions by holding the wood cylinder differently. Because the wood cylinder is radially symmetric, the robot can hold it in such a way that prevents its arms from colliding. A trial of this task was successful if the robot successfully grasped the two objects, executed the two motions simultaneously without its arms colliding, and placed the objects back on the table.

Place setting

In this task, the robot must simultaneously arrange a bowl on a plate and a cup next to the plate. The unique feature of this task is that one of the arms is doing a pick-and-place action (placing the bowl on the plate), while the other is doing a trajectory-following action (placing the cup next to the plate). A trial of this task was successful if the robot successfully grasped both the cup and the bowl and placed them in the correct locations.

Unstack cups

In this task, there is a stack of two plastic cups, and the robot needs to separate them. Because there is a small amount of suction between the two cups when they are stacked, the robot must use one arm to grasp the bottom cup while picking up the top cup with the other arm. This task shows how the robot imitates grasps from the demonstration. The bottom cup must be grasped using a power grasp from the side. However, the cup widens at the top and cannot be grasped from the side. Instead, the robot must grasp the rim of the top cup. Additionally, the robot must not grasp too far down the cup, because then it would grasp the bottom cup as well. Another unique feature of this task is that the two objects start close together, and the robot separates them, while the other tasks listed here are the converse. Lastly, while the cups are stacked, the object tracker has a hard time correctly determining the location of the bottom cup, so we used the task annotator to correct the location of the bottom cup. A trial of this task was considered a success if the robot successfully separated the two cups and placed the top cup on the table.

10.8.3 Results

In this section, we describe *PBJ*'s performance during the annotation and execution phases. During the annotation procedure, we had to make corrections to the output of the skeleton tracker and the object tracker. This required effort by the annotator. The ideal annotation system would require no adjustments at all, and such a system would be able to annotate a demonstration video automatically. During the execution phase, we would like for the robot to be able to generate a plan

to perform the task even as the objects move to different starting locations. Additionally, it should execute those plans without missing grasps, colliding with obstacles, or dropping objects.

Annotation results

Table 10.2 shows the number of times we made adjustments to the output of the skeleton tracker or object tracker for each task. The skeleton tracker needed almost no adjustments. When we did make adjustments, it was because the skeleton tracker placed the hands on the tabletop, which sometimes interfered with the interpretation of grasp events.

Task	**# frames**	**# skeleton pose changes**	**# obj pose changes**
Reposition object	173	0	0
Pour cup	164	1	19
Drop ball	144	2	31
Stack bowl	135	0	0
Arrange boxes	271	0	0
Stir and pour	325	4	102
Place setting	158	0	4
Unstack cups	116	0	74

Table 10.2: Number of annotation adjustments for tasks used in the *PBJ* evaluation. The third column gives the number of frames in which we adjusted at least one joint of the skeleton pose. The last column gives the number of times we adjusted any object's pose during a given demonstration.

The number of adjustments we made to the object tracker's output depended on the task. In general, we noticed that the object tracker had a harder time tracking small objects that were

mostly occluded by the demonstrator's hand. These objects included the mug, the wood cylinder, and the tennis ball. This explains why tasks that used those objects required adjustments to the object tracker's output. The *unstack cups* task required the most adjustments of all, with 74 pose adjustments in 111 video frames. This was because the object tracker could not distinguish between the top and bottom cups while the two cups were nested. As a result, all of the video frames in which the two cups were close together required annotation.

Execution results

Table 10.3 shows the results of running *PBJ* on five trials of each task. We measured the number of successful trials and noted the reasons for any failures. Overall, 36 out of the 40 executions were successful. The robot failed one trial of the *pour cup* task because it was unable to generate a plan to reach a bowl placed farther away than it was in the demonstration. The remaining failures were due to errors in the motion execution. In the *stack bowl* task, the robot failed one trial by missing a grasp, which may have been due to a calibration error. In the *stir and pour* task, the robot placed the wood cylinder back on the table. However, the location that was specified was past the edge of the table, and the wood cylinder fell to the ground. Although our system detects the tabletop and knows its location and dimensions, it only uses this information to avoid colliding with the table. In the future, the system could use this information to avoid dropping objects past the edge of the table. The final failed trial was in the *unstack cups* task. The robot did not get a strong enough grip on the top cup, and the top cup slipped out of its grasp.

10.8.4 Discussion

In this chapter, we described the *PBJ* system and showed how it could be used to derive motion specifications from video demonstrations of manipulations tasks. We successfully programmed and execute these tasks in real-world experiments. However, there remain many areas of improvement. Below, we discuss some of the system's limitations and areas for future work.

An intriguing possibility for future work is to make the annotation process fully automated, or

Task	# of successful trials
Reposition object	5
Pour cup	4
Drop ball	5
Stack bowl	4
Arrange boxes	5
Stir and pour	4
Place setting	5
Unstack cups	4

Table 10.3: Number of successful trials for each task of the *PBJ* task evaluation, out of 5. Details about the failed trials are given in the text.

at least to automate more of the process. For example, instead of requiring users to provide the initial poses of the objects in the demonstration video, we could use the object pose initialization system that the system uses at runtime (Section 10.5). This step could also be improved by using other state of the art systems for finding the 6D poses of objects, which is still an active area of research.

The main challenge with fully automating the annotation process is developing highly trackers that are highly robust—enough to not require human supervision. For example, when tracking two instances of the same object, placed closely together, we found that the object tracking system we used tended to get the two objects confused. Until then, we believe it is important to give users the option to make corrections as necessary.

Grasp planning was one of the most computationally expensive parts of the system. As we saw earlier, we used an uninformed sampling-based approach that generated and scored many candidate grasps. Because our system assumes that users have models of all the objects, a simple optimization is to pre-compute feasible candidate grasps for each object offline. At runtime, the system could score just the set of candidate grasps as described in earlier.

Our system generated open-loop plans that the robot executed without any feedback. As we saw in the results section above, this led to program execution failures that the robot did not detect, such as when it failed to properly grasp an object. In the future, program execution could be made more robust by incorporating feedback into the execution. For example, the robot could visually track its gripper to get a more accurate estimate of its pose, which would help if the robot was not well-calibrated. Another idea is to detect failed grasps and attempt to re-execute them.

Another interesting possibility is to give users even more tools for specifying motions. For example, a user may want to demonstrate a circular motion or a straight-line motion. While a human demonstrator cannot make perfectly circular motions, they could annotate a portion of the demonstrated motion to be a perfect cirle using a "circle tool." This would be similar to the use of "virtual fixtures" in other robotics applications citedellin2014guidedmp.

Finally, another area for future work is to evaluate the system's ease of use, both for the demonstrator and for the annotator. As described earlier, our system has restrictions on the kinds of

motions it can generate. However, those restrictions are not tangible for a demonstrator creating a video recording of a task. More work needs to be done to see how well demonstrators understand the system's programming model and its limitations. It would also be interesting to explore whether providing feedback to the demonstrator would help them create demonstrations. Additional work is also needed to determine the intuitiveness of the system's annotation interface and how it could be improved.

Chapter 11

CONCLUSION

This document described our research on robot programming systems for non-expert users. We discussed two main approaches, direct programming (embodied by the *Code3* system in Chapter 9) and programming by imitation (embodied by *PBJ* in Chapter 10). We presented our research on technologies that support these systems, including *CustomLandmarks*, shelf segmentation, *RapidPbD*, and *CustomPrograms* Throughout this document, we found that non-expert and non-roboticist programmers could use simplified interfaces to customize the behaviors of their robots and develop useful mobile manipulation programs. We also pointed out numerous areas where these systems could be improved, and our conceptual framework provides an easy way to modify and build upon our work. We also investigated the potential of trigger-action programming as a high-level programming interface for end-users. Overall, we are optimistic that a diverse group of users will be empowered to rapidly program mobile manipulator robots for a variety of useful tasks in the future.

11.1 Future work

11.1.1 Extending CustomLandmarks to 2D

One limitation of *CustomLandmarks* is that it relies solely on depth information, and it does not use color information. This makes it incapable of locating flat objects such as elevator buttons, flat light switches, or printed patterns in the workspace. However, we could extend the concepts of *CustomLandmarks* to work with 2D color images. This would work as follows:

1. The user draws a box in the 2D image, specifying a 2D template of a landmark.

2. The robot uses a 2D template matching algorithm to locate instances of the template in a new

color image. This algorithm may incorporate common steps such as applying a confidence threshold and deduplicating nearby matches using non-max suppression.

3. Assuming the color image comes from an RGBD camera (or has otherwise been registered to depth data), identify the 3D locations of each pixel in the matched template. If the data is from an RGBD camera, this can be done simply by reading the depth channel of each matched pixel. If the depth and color images come from separate sources, one approach to perform this step is to project each 3D point into the image plane of the 2D camera, then check if the pixel is in the bounding box of a template match.

4. Fit a bounding box around the matched 3D data and attach a local coordinate frame to the center of this box.

This approach assumes that the landmarks are flat and roughly parallel to the image plane of the 2D camera.

2D landmark prototype

We developed a prototype of a 2D landmark system as described above, called *CustomLandmarks2D*. This prototype uses a standard template matching algorithm using the open-source OpenCV library [18]. In our implementation, we used an RGBD camera, which allowed us to easily associate pixels in the 2D image to depth data.

We used this prototype for two test scenarios. First, we used the system to detect whether grocery items on a shelf were facing outward or not. This could be useful for a larger program in which the robot locates items that are not facing out properly, and rotates them appropriately if not. This is shown in Figure 11.1. This example also shows that *CustomLandmarks2D* can be used, to some extent, on non-flat objects, as well as on flat objects that are not parallel to the image plane. However, we have not conducted experiments to find the limitations of the system in these respects.

(a)

(b)

Figure 11.1: Results of the prototype *CustomLandmarks2D* system. The upper right image shows the detections in 2D space. The bottom image shows the detections projected onto the 3D data, highlighted with red pixels.

As a second proof of concept, we integrated *CustomLandmarks2D* with the *RapidPbD* motion specification system (Section 6.3). This is illustrated in Figure 11.2. In this experiment, the robot used a tabletop object detection system (Chapter 5) to locate the box and *CustomLandmarks2D* to locate the fiducial. We then specified a motion to pick up the box and place it on the fiducial marker. The robot successfully executed this action.

This experiment shows that we could detect a 2D pattern, find its 3D location, and program manipulation actions using the given 3D pose as a landmark. It is important to note that *CustomLandmarks2D* can only detect a fiducial based on general similarity to a template image of the same fiducial. We do not intend for *CustomLandmarks2D* to be a replacement for a specialized fiducial tracking system. However, *CustomLandmarks2D* could be used to locate non-fiducial markers, such as brightly colored stickers that are constructed for a specific task.

As with the 3D case, we expect that a system like *CustomLandmarks2D* is primarily useful for letting users create detectors for patterns that are not detected by other, more specialized systems.

We have not formally evaluated *CustomLandmarks2D*, but we have released open-source code and documentation on its usage at `https://wiki.ros.org/custom_landmark_2d`.

11.1.2 Other improvements to CustomLandmarks

Empty space representation

As currently implemented, *CustomLandmarks* limits users' ability to model landmarks. For simplicity of implementation, for example, the box that is used to represent empty space is an axis-aligned box, aligned with the robot's base frame. However, users may need greater flexibility in how they model empty space. In the future, we can imagine enhancing the interface for creating landmarks to support arbitrary orientations. Additionally, the empty space region need not be box-shaped, and it may be more appropriate to use other kinds of shapes that provide a better fit. For example, a cylinder shape may be better for radially symmetric objects.

Figure 11.2: The *CustomLandmarks2D* prototype running on the Fetch robot.

Rapid iteration

The *CustomLandmarks* user evaluation showed that our system requires trial and error by the users. After specifying a landmark, they saw what the detection outputs were in a new scene, and refined the landmark specification multiple times. This process could be slowed down if there are several scenes that the user wants to use a landmark for. To support rapid iteration when creating landmarks, our interface could be extended to allow the user to import several scenes of interest. After drawing a box to specify a landmark in one of the scenes, the interface could then run the landmark detection algorithm in all of the scenes at once. This would give the user quick feedback as to how well the landmark they specified would work across many scenes.

11.1.3 Concurrency in task scripting

In our work, task scripting systems such as *CustomPrograms* run sequentially, waiting for one action to finish before starting the next. This makes the programming process simpler, and it works in many cases. However, this can be unnatural-looking and inefficient if say, the robot waits to finish pointing its head at a workspace before moving its arm to a "ready" position, when the robot could have done both actions simulataneously. To address this, users must be able to program the robot to do more than one thing at a time. This is especially important when we want the robot to interact with the user while performing a task. A common case is when we want the robot to display a stop button on its touchscreen while doing something else (*e.g.*, driving to a location). This not possible in our current system—the robot can display the button on its touchscreen and wait for it to be pushed, or it can drive, but not both at the same time.

We have experimented with a prototype system for concurrent programming, inspired by the ROS Actionlib[1] library. In our prototype, robot actions (move the arm, drive to a given location, etc.) are broken down into 4 methods: `Start`, `IsDone`, `Cancel`, and `GetResult`. The robot implements controllers for different actuators (head, torso, arm, wheels, touchscreen, etc.) that run concurrently with the main program. In the `Start` method, the programmer passes in arguments

[1] `https://wiki.ros.org/actionlib`

that specify the goal of the action. The developer then calls `IsDone` repeatedly using a loop to wait for the action to finish. When the action is done, calling `GetResult` yields status information or other action-specific data. Calling `Cancel` on an action at any time will tell the controller to stop performing the action.

This system allows users to execute actions concurrently in a number of ways. Two common patterns are shown in Algorithms 5 and 6. The first is *wait for all*, where the programmer starts two actions simultaneously and waits for them both to finish. The second is *wait for one*, where the programmer starts two actions simultaneously and waits for one of them to finish, canceling the other one.

Algorithm 5: "Wait for all" concurrency pattern

```
   /* Concurrently move head and arm into ready pose          */
1  Start (PanTiltHead, 0, 45);
2  Start (MoveArm, "Ready");
3  while (not IsDone (PanTiltHead) or not IsDone (MoveArm)){
4  }
   /* Both head and arm are ready                             */
```

We can imagine a variety of actions and more complex sequences being built in this way. For example, one long-running action could be started concurrently while multiple shorter actions are run in sequence at the same time. A timer action could be used in conjunction with the "wait for one" pattern to implement a timeout check for a long-running action. We could also imagine developing perceptual actions, such as a slip detector for the gripper. This action could activated immediately after successfully grasping an object. The action's `IsDone` method would return false if the robot's gripper position remained unchanged, but it would return true if the gripper unexpectedly closed all the way (*i.e.*, the object fell out of the grasp). This action could also be used with the "wait for one" pattern to quickly detect when the robot has failed at a task.

In addition to thinking about what kinds of new actions we could develop with this system,

Algorithm 6: "Wait for one" concurrency pattern

```
1  Start (AskMultipleChoice, "Press the button to pause", ["Pause"]);
2  Start (Navigation, "Place 1");
3  while (not IsDone (PanTiltHead) or not IsDone (MoveArm)){
4      if (IsDone (AskMultipleChoice)){
           /* Button was pressed, cancel navigation                */
5          Cancel (Navigation);
6          break out of loop;
7      }
8      if (IsDone (Navigation)){
           /* Arrived at destination, stop showing pause button    */
9          Cancel (AskMultipleChoice);
10         break out of loop;
11     }
12 }
```

as well as what concurrency patterns might arise, we also must study the usability of this system. It is not clear if non-expert programmers will understand this system and be able to use it effectively. User studies could also help us develop templates or other abstractions for the most-used concurrency patterns.

11.1.4 Other areas for future work

In the remainder of this chapter, we will discuss some higher-level areas for future work that we believe will further advance research in this area.

User segments

Animated films and video games represent two highly technical and commercially successful industries. These industries are also similar to robotics in that they heavily utilize technologies for world modeling, physical simulations, character motions, and real-time operation. Despite the highly technical core of these industries, the final product is a combination of efforts from many different kinds of people, including artists, designers, animators, and coders. These groups also all rely heavily on tools that make it easy to generate intermediate content as well as the final product. An interesting question for us is if we can push the field of robotics in a similar direction, in which content production (*i.e.*, robot task programs) are generated by a wide variety of users using a collection of useful tools.

In our research, we developed systems for different audiences and conducted user studies with different groups, including novice coders, experienced software engineers, and university students. Although we have touched on the idea that different kinds of users want different kinds of ways to program or customize their robots, we have not concretely explored what these user segments are. As robot programming tools such as those described in this dissertation are increasingly developed and deployed, it should be possible to prototype the use of mobile manipulators for specific tasks and groups of people. For example, in an elder care setting, we could prototype tasks and ask what kinds of programmability is desired by older adults, their families, caretakers, and robot

technicians. By doing this kind of investigation, we can better discover applications for mobile manipulators to help people, and productively direct research efforts towards technologies that are necessary to support them, as well as new or improved features needed for robot programming tools.

Web technologies for robotics

Our research made use of web technologies in several instances. Deploying tools and other robot applications to the web is an economical way to make these applications immediately available to a wide range of platforms, including mobile devices. Recently, new web technologies have been introduced that could accelerate the use of the web for robotics, including tools we have developed [67], WebGL, and WebAssembly, a binary format that can be compiled to from C++ and other high-level languages [140]. At the same time, web security remains a concern for developers, especially when allowing remote access to a robot [45]. Following in the footsteps of web security research in general, this will likely be an important area of research for some time.

Simulations and operations

We focused on three core capabilities for our robot programming systems: perception, motion specification, and task scripting. We also examined how our system could be extended to support concurrency. However, simulation and operational support systems are another major area of robot programming that our research largely overlooks. Users would be much more efficient and confident if they could run lightweight simulations at every phase of programming the robot. In Chapter 4, we alluded to an area of future work that would allow users to quickly visualize the outcome of defining certain custom landmarks. However, there should be simulation support for robot motion specification as well as task scripting. This is especially important for robotics because otherwise, users can only test their programs by running their programs on the real robot hardware. This might lead to unsafe situations if the robot performs an unexpected behavior.

In commercial settings, robot operations and remote monitoring is a critical component of the

overall business. However, our work does not provide users with any tools for monitoring the performance of the robot as it executes a program. We also need to provide users with ways to perform other operational work, such as performing ad hoc recovery behaviors when the robot is lost or stuck or changing its programmed behavior mid-program.

As of the writing of this dissertation, mobile manipulator robots are not yet widely in use, especially not in the lives of end-users. However, we believe that the potential of mobile manipulator robots has not be fully realized yet, whether it is making a logistics operation more efficient, helping in a disaster relief scenario, or assisting people with disabilities with household tasks. Our hope is that our research can help a wide variety of people explore and develop new robot applications, and that these applications can make a positive impact on people's lives.

BIBLIOGRAPHY

[1] Baris Akgun, Maya Cakmak, Jae Wook Yoo, and Andrea Lockerd Thomaz. Trajectories and keyframes for kinesthetic teaching: A human-robot interaction perspective. In *Proceedings of the seventh annual ACM/IEEE international conference on Human-Robot Interaction*, pages 391–398. ACM, 2012.

[2] Baris Akgun and Andrea Thomaz. Simultaneously learning actions and goals from demonstration. *Autonomous Robots*, 40(2):211–227, 2016.

[3] Jacopo Aleotti, Stefano Caselli, and Monica Reggiani. Leveraging on a virtual environment for robot programming by demonstration. *Robotics and Autonomous Systems*, 47(2):153–161, 2004.

[4] Sonya Alexandrova, Maya Cakmak, Kaijen Hsiao, and Leila Takayama. Robot programming by demonstration with interactive action visualizations. In *Robotics: Science and Systems (RSS)*, pages 48–56, 2014.

[5] Sonya Alexandrova, Maya Cakmak, Kaijen Hsiao, and Leila Takayama. Robot programming by demonstration with interactive action visualizations. In *Robotics: Science and Systems (R:SS)*, pages 48–56, 2014.

[6] Sonya Alexandrova, Zachary Tatlock, and Maya Cakmak. RoboFlow: A flow-based visual programming language for mobile manipulation tasks. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5537–5544. IEEE, 2015.

[7] Amazon Robotics, Inc. Amazon Picking Challenge. In `https://amazonpickingchallenge.org/`, 2015.

[8] Gary Anthes. Deep learning comes of age. *Communications of the ACM*, 56(6):13–15, 2013.

[9] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009.

[10] Christopher G. Atkeson and Stefan Schaal. Robot learning from demonstration. In *ICML*, volume 97, pages 12–20, 1997.

[11] Aaron Bangor, Philip T Kortum, and James T Miller. An empirical evaluation of the system usability scale. *Intl. Journal of Human–Computer Interaction*, 24(6):574–594, 2008.

[12] Paul J. Besl and Neil D. McKay. A method for registration of 3-D shapes. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14:239–256, 1992.

[13] Geoffrey Biggs and Bruce MacDonald. A survey of robot programming systems. In *Proceedings of the Australasian Conference on Robotics and Automation*, pages 1–3, 2003.

[14] Aude Billard, Sylvain Calinon, Ruediger Dillmann, and Stefan Schaal. Robot programming by demonstration. In *Springer Handbook of Robotics*, pages 1371–1394. Springer, 2008.

[15] Rainer Bischoff, Arif Kazi, and Markus Seyfarth. The MORPHA style guide for icon-based programming. In *Robot and Human Interactive Communication, 2002. Proceedings. 11th IEEE International Workshop on*, pages 482–487. IEEE, 2002.

[16] Andrew Blake, Carsten Rother, Matthew Brown, Patrick Perez, and Philip Torr. Interactive image segmentation using an adaptive GMMRF model. In *European Conference on Computer Vision*, pages 428–441. Springer, 2004.

[17] Jonathan Bohren, Radu Bogdan Rusu, E. Gil Jones, Eitan Marder-Eppstein, Caroline Pantofaru, Melonee Wise, Lorenz Mösenlechner, Wim Meeussen, and Stefan Holzer. Towards autonomous robotic butlers: Lessons learned with the PR2. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 5568–5575. IEEE, 2011.

[18] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[19] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc., 2008.

[20] John Brooke et al. SUS-a quick and dirty usability scale. *Usability Evaluation in Industry*, 189(194):4–7, 1996.

[21] FP Brooks. *No silver bullet*. April, 1987.

[22] Frederick P. Brooks, Jr. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[23] Anders Glent Buch, Dirk Kraft, Joni-Kristian Kamarainen, Henrik Gordon Petersen, and Norbert Krüger. Pose estimation using local structure-specific shape and appearance context. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 2080–2087, 2013.

[24] Daniel J. Butler, Sarah Elliot, and Maya Cakmak. Interactive scene segmentation for efficient human-in-the-loop robot manipulation. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2572–2579, Sept 2017.

[25] Bill Buxton. *Sketching user experiences: getting the design right and the right design*. Morgan Kaufmann, 2010.

[26] Maya Cakmak. CSE 481C autumn 2013 labs. `https://sites.google.com/site/cse481a/labs`, 2013.

[27] Maya Cakmak. CSE 481C winter 2018 course calendar. `https://sites.google.com/view/cse481wi18/course-calendar`, 2018.

[28] Maya Cakmak and Andrea L Thomaz. Eliciting good teaching from humans for machine learners. *Artificial Intelligence*, 217:198–215, 2014.

[29] Sylvain Calinon and Aude Billard. Stochastic gesture production and recognition model for a humanoid robot. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2769–2774. IEEE, 2004.

[30] Sylvain Calinon and Aude Billard. Learning of gestures by imitation in a humanoid robot. Technical report, Cambridge University Press, 2007.

[31] Sylvain Calinon and Aude Billard. Statistical learning by imitation of competing constraints in joint space and task space. *Advanced Robotics*, 23(15):2059–2076, 2009.

[32] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Realtime multi-person 2D pose estimation using part affinity fields. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7291–7299, 2017.

[33] Tiffany L. Chen, Matei Ciocarlie, Steve Cousins, Phillip M. Grice, Kelsey Hawkins, Kaijen Hsiao, Charles C. Kemp, Chih-Hung King, Daniel A. Lazewatsky, Adam E. Leeper, et al. Robots for humanity: Using assistive robotics to empower people with disabilities. *IEEE Robotics & Automation Magazine*, 20(1):30–39, 2013.

[34] Sonia Chernova and Andrea L. Thomaz. Robot learning from human teachers. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(3):1–121, 2014.

[35] Michael Jae-Yoon Chung, Justin Huang, Leila Takayama, Tessa Lau, and Maya Cakmak. Iterative design of a system for programming socially interactive service robots. In *International Conference on Social Robotics*, pages 919–929. Springer, 2016.

[36] Daniel C Cliburn. Experiences with the LEGO Mindstorms throughout the undergraduate computer science curriculum. In *Frontiers in Education Conference, 36th Annual*, pages 1–6. IEEE, 2006.

[37] Alvaro Collet, Manuel Martinez, and Siddhartha Srinivasa. The MOPED framework: Object recognition and pose estimation for manipulation. *The International Journal of Robotics Research*, 30(10):1284–1306, 2011.

[38] Toby HJ Collett, Bruce A MacDonald, and Brian P Gerkey. Player 2.0: Toward a practical robot programming framework. In *Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2005)*, page 145, 2005.

[39] Nikolaus Correll, Kostas E Bekris, Dmitry Berenson, Oliver Brock, Albert Causo, Kris Hauser, Kei Okada, Alberto Rodriguez, Joseph M Romano, and Peter R Wurman. Lessons from the Amazon Picking Challenge. *arXiv preprint arXiv:1601.05484*, 2016.

[40] P.T. Cox, C.C. Risley, and T.J. Smedley. Toward concrete representation in visual languages for robot control. *Journal of Visual Languages & Computing*, 9(2):211–239, 1998.

[41] John J. Craig. *Introduction to robotics: Mechanics and control*, volume 3. Pearson/Prentice Hall Upper Saddle River, NJ, USA:, 2005.

[42] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, pages 186–189. ACM, 2002.

[43] Chandan Datta, Chandimal Jayawardena, I Han Kuo, and Bruce A MacDonald. RoboStudio: A visual programming environment for rapid authoring and customization of complex services on a personal service robot. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2352–2357. IEEE, 2012.

[44] Christopher M. Dellin, Kyle Strabala, G. Clark Haynes, David Stager, and Siddhartha S. Srinivasa. Guided manipulation planning at the DARPA Robotics Challenge trials. In *ISER*, 2014.

[45] Nicholas DeMarinis, Stefanie Tellex, Vasileios Kemerlis, George Konidaris, and Rodrigo Fonseca. Scanning the Internet for ROS: A view of security in robotics research. *arXiv preprint arXiv:1808.03322*, 2018.

[46] Anind K Dey, Timothy Sohn, Sara Streng, and Justin Kodama. iCAP: Interactive prototyping of context-aware applications. In *Pervasive Computing*, pages 254–271. Springer, 2006.

[47] Mehmet Dogar and Siddhartha Srinivasa. A framework for push-grasping in clutter. *Robotics: Science and systems VII*, 1, 2011.

[48] Bertram Drost, Markus Ulrich, Nassir Navab, and Slobodan Ilic. Model globally, match locally: Efficient and robust 3D object recognition. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 998–1005. IEEE, 2010.

[49] W Keith Edwards and Rebecca E Grinter. At home with ubiquitous computing: Seven challenges. In *Ubicomp 2001: Ubiquitous Computing*, pages 256–272. Springer, 2001.

[50] Staffan Ekvall and Danica Kragic. Robot learning from demonstration: A task-level planning approach. *International Journal of Advanced Robotic Systems*, 5(3):223–234, 2008.

[51] Kerstin Fischer, Franziska Kirstein, Lars Christian Jensen, Norbert Krüger, Kamil Kukliński, Thiusius Rajeeth Savarimuthu, et al. A comparison of types of robot control for programming by demonstration. In *The Eleventh ACM/IEEE International Conference on Human Robot Interaction*, pages 213–220. IEEE Press, 2016.

[52] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[53] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

[54] N Fraser et al. Blockly: A visual programming editor. `https://developers.google.com/blockly/`, 2013.

[55] D Glas, Satoru Satake, Takayuki Kanda, and Norihiro Hagita. An interaction design framework for social robots. In *Robotics: Science and Systems*, volume 7, page 89, 2012.

[56] Dylan F Glas, Takayuki Kanda, and Hiroshi Ishiguro. Human-robot interaction design using Interaction Composer: Eight years of lessons learned. In *The Eleventh ACM/IEEE International Conference on Human Robot Interaction*, pages 303–310. IEEE Press, 2016.

[57] Rebecca A. Grier. How high is high? A meta-analysis of NASA-TLX global workload scores. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 59, pages 1727–1731. SAGE Publications Sage CA: Los Angeles, CA, 2015.

[58] Marcus Gualtieri, Andreas ten Pas, Kate Saenko, and Robert Platt. High precision grasp pose detection in dense clutter. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, pages 598–605. IEEE, 2016.

[59] Jonna Häkkilä, Panu Korpipää, Sami Ronkainen, and Urpo Tuomela. Interaction and end-user programming with a context-aware mobile application. In *Human-Computer Interaction-INTERACT 2005*, pages 927–937. Springer, 2005.

[60] Emily Hamner, Tom Lauwers, Debra Bernstein, Illah Nourbakhsh , and Carl Francis DiSalvo. Robot diaries: Broadening participation in the computer science pipeline through social technical exploration. In *AAAI Spring Symposium on Using AI to Motivate Greater Participation in Computer Science*, March 2008.

[61] Sandra G Hart. NASA-task load index (NASA-TLX): 20 years later. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 50, pages 904–908. Sage Publications Sage CA: Los Angeles, CA, 2006.

[62] Carlos Hernandez, Mukunda Bharatheesha, Wilson Ko, Hans Gaiser, Jethro Tan, Kanter van Deurzen, Maarten de Vries, Bas Van Mil, Jeff van Egmond, Ruben Burger, et al. Team Delfts robot winner of the Amazon Picking Challenge 2016. In *Robot World Cup*, pages 613–624. Springer, 2016.

[63] Alexander Herzog, Peter Pastor, Mrinal Kalakrishnan, Ludovic Righetti, Tamim Asfour, and Stefan Schaal. Template-based learning of grasp selection. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 2379–2384. IEEE, 2012.

[64] Daniel D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, 3(1):69–101, 1992.

[65] Kaijen Hsiao, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Grasping POMDPs. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 4685–4692. IEEE, 2007.

[66] Kaijen Hsiao and Tomás Lozano-Pérez. Imitation learning of whole-body grasps. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 5657–5662. IEEE, 2006.

[67] Justin Huang and Maya Cakmak. Reactive web interfaces with Polymer and ROS. *ROSCon 2017*, 2017.

[68] IFTTT. Ifttt.com. `https://www.ifttt.com`, 2015. [Online; accessed 29-September-2015].

[69] Caitlin Kelleher, Randy Pausch, and Sara Kiesler. Storytelling Alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1455–1464. ACM, 2007.

[70] Elizabeth S Kim, Dan Leyzberg, Katherine M Tsui, and Brian Scassellati. How people talk when teaching a robot. In *Human-Robot Interaction (HRI), 2009 4th ACM/IEEE International Conference on*, pages 23–30. IEEE, 2009.

[71] S.H. Kim and J.W. Jeon. Programming LEGO Mindstorms NXT with visual programming. In *Control, Automation and Systems, 2007. ICCAS'07. International Conference on*, pages 2468–2472. IEEE, 2007.

[72] A.J. Ko, B.A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 199–206. IEEE, 2004.

[73] Kory Kraft and William D. Smart. Seeing is comforting: Effects of teleoperator visibility in robot-mediated health care. *2016 11th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 11–18, 2016.

[74] James J Kuffner and Steven M LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 995–1001. IEEE, 2000.

[75] Pamela B Lawhead, Michaele E Duncan, Constance G Bland, Michael Goldweber, Madeleine Schep, David J Barnes, and Ralph G Hollingsworth. A road map for teaching introductory programming using LEGO Mindstorms robots. *ACM SIGCSE Bulletin*, 35(2):191–201, 2003.

[76] Ian Lenz, Honglak Lee, and Ashutosh Saxena. Deep learning for detecting robotic grasps. *The International Journal of Robotics Research*, 34(4-5):705–724, 2015.

[77] Benjamin Lester. Robots' allure: Can it remedy what ails computer science? *Science*, 318(5853):1086–1087, 2007.

[78] H. Lieberman, F. Paterno, M. Klann, and V. Wulf. *End-user development: An emerging paradigm*. Springer, 2006.

[79] YuXuan Liu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Imitation from observation: Learning to imitate behaviors from raw video via context translation. *arXiv preprint arXiv:1707.03374*, 2017.

[80] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

[81] Tino Lourens. TiViPE-Tino's visual programming environment. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, pages 10–15. IEEE, 2004.

[82] Tino Lourens and Emilia Barakova. User-friendly robot environment for creation of social scenarios. In *International Work-Conference on the Interplay between Natural and Artificial Computation*, pages 212–221. Springer, 2011.

[83] Tomás Lozano-Pérez. Robot programming. *Proceedings of the IEEE*, 71(7):821–841, 1983.

[84] Tomás Lozano-Pérez, Joseph Jones, Emmanuel Mazer, Patrick O'Donnell, W Grimson, Pierre Tournassoud, and Alain Lanusse. Handey: A robot system that recognizes, plans, and manipulates. In *Robotics and Automation. Proceedings. 1987 IEEE International Conference on*, volume 4, pages 843–849. IEEE, 1987.

[85] Wendy E Mackay. *Users and customizable software: A co-adaptive phenomenon*. PhD thesis, Citeseer, 1990.

[86] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The Scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):16, 2010.

[87] David Martinez and Danica Kragic. Modeling and recognition of actions through motor primitives. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 1704–1709. IEEE, 2008.

[88] Maja J. Matarić, Nathan Koenig, and David Feil-Seifer. Materials for enabling hands-on robotics and stem education. In *In AAAI Spring Symposium on Robots and Robot Venues: Resources for AI Education*, 2007.

[89] Daniel Maturana and Sebastian Scherer. VoxNet: A 3D convolutional neural network for real-time object recognition. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 922–928, 2015.

[90] M. Mernik, J. Heering, and A.M. Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[91] Anahita Mohseni-Kabir, Charles Rich, Sonia Chernova, Candace L Sidner, and Daniel Miller. Interactive hierarchical task learning from a single demonstration. In *Proceedings of the Tenth Annual ACM/IEEE International Conference on Human-Robot Interaction*, pages 205–212. ACM, 2015.

[92] Michael Montemerlo, Nicholas Roy, and Sebastian Thrun. Perspectives on standardization in mobile robot programming: The Carnegie Mellon navigation (CARMEN) toolkit. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 3, pages 2436–2441. IEEE, 2003.

[93] Caio Mucchiani, Suneet Sharma, Megan Johnson, Justine Sefcik, Nicholas Vivio, Justin Huang, Pamela Cacchione, Michelle Johnson, Roshan Rai, Adrian Canoso, et al. Evaluating older adults' interaction with a mobile assistive robot. In *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pages 840–847. IEEE, 2017.

[94] B.A. Myers. Visual programming, programming by example, and program visualization: a taxonomy. In *ACM SIGCHI Bulletin*, volume 17, pages 59–66. ACM, 1986.

[95] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. In *Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on*, pages 127–136. IEEE, 2011.

[96] Hai Nguyen, Matei Ciocarlie, Kaijen Hsiao, and Charles C. Kemp. ROS Commander (ROSCo): Behavior creation for home robots. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 467–474. IEEE, 2013.

[97] Scott Niekum, Sachin Chitta, Andrew G. Barto, Bhaskara Marthi, and Sarah Osentoski. Incremental semantically grounded learning from demonstration. In *Robotics: Science and Systems*, 2013.

[98] Scott Niekum, Sarah Osentoski, George Konidaris, and Andrew G. Barto. Learning and generalization of complex tasks from unstructured demonstrations. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5239–5246. IEEE, 2012.

[99] Scott Niekum and Isaac I. Y. Saito. The ar_track_alvar ROS package. `http://wiki.ros.org/ar_track_alvar`.

[100] R. Brook Osborne, Antony J. Thomas, and Jeffrey R.N. Forbes. Teaching with robots: A service-learning approach to mentor training. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 172–176, 2010.

[101] Paschalis Panteleris, Iason Oikonomidis, and Antonis Argyros. Using a single RGB frame for real time 3D hand pose estimation in the wild. *arXiv preprint arXiv:1712.03866*, 2017.

[102] Chavdar Papazov, Sami Haddadin, Sven Parusel, Kai Krieger, and Darius Burschka. Rigid 3D geometry matching for grasping of known objects in cluttered scenes. *The International Journal of Robotics Research*, 31(4):538–553, 2012.

[103] Michael Pardowitz, Steffen Knoop, Ruediger Dillmann, and RD Zollner. Incremental learning of tasks from user demonstrations, past experiences, and vocal comments. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 37(2):322–332, 2007.

[104] Peter Pastor, Heiko Hoffmann, Tamim Asfour, and Stefan Schaal. Learning and generalization of motor skills by learning from demonstration. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 763–768, 2009.

[105] Dan Perkel. Copy and paste literacy? Literacy practices in the production of a MySpace profile. *xxxix. Informal Learning and Digital Media: Constructions, Contexts, Consequences, eds. Kirsten Drotner, Hans Siggard Jensen, and Kim Schroeder (Newcastle, UK: Cambridge Scholars Press, 2008)*, 2006.

[106] Emmanuel Pot, Jérôme Monceaux, Rodolphe Gelin, and Bruno Maisonnier. Choregraphe: A graphical tool for humanoid robot programming. In *RO-MAN 2009-The 18th IEEE International Symposium on Robot and Human Interactive Communication*, pages 46–51. IEEE, 2009.

[107] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, volume 3, page 5, 2009.

[108] Joseph Redmon and Anelia Angelova. Real-time grasp detection using convolutional neural networks. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 1316–1322. IEEE, 2015.

[109] A. Repenning and A. Ioannidou. What makes end-user development tick? 13 design guidelines. In *End User Development*, pages 51–85. Springer, 2006.

[110] F. Riedo, M. Chevalier, S. Magnenat, and F. Mondada. Thymio II, a robot that grows wiser with children. In *IEEE Workshop on Advanced Robotics and its Social Impacts (ARSO)*, pages 187–193, 2013.

[111] Rethink Robotics. Baxter user guide. `http://mfg.rethinkrobotics.com/wiki/Support_Resources`.

[112] Radu Bogdan Rusu. *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments*. PhD thesis, Computer Science department, Technical University of Munich, Germany, October 2009.

[113] Radu Bogdan Rusu, Nico Blodow, and Michael Beetz. Fast point feature histograms (FPFH) for 3D registration. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 3212–3217, 2009.

[114] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point cloud library (PCL). In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1–4, 2011.

[115] Allison Sauppé and Bilge Mutlu. Design patterns for exploring and prototyping human-robot interactions. In *Proceedings of the 32nd Annual ACM Conference on Human Factors in Computing Systems*, pages 1439–1448. ACM, 2014.

[116] Savioke, Inc. Savioke announces first Las Vegas installation and new Relay features. `http://www.savioke.com/blog/2018/1/8/gpwkugbcn9xev881b1wpko1p63ekai`, 2018.

[117] Ashutosh Saxena, Justin Driemeyer, and Andrew Y Ng. Robotic grasping of novel objects using vision. *The International Journal of Robotics Research*, 27(2):157–173, 2008.

[118] Stefan Schaal, Jan Peters, Jun Nakanishi, and Auke Ijspeert. Learning movement primitives. In *International Symposium on Robotics Research.*, pages 561–572, 2005.

[119] Pierre Sermanet, Corey Lynch, Jasmine Hsu, and Sergey Levine. Time-contrastive networks: Self-supervised learning from multi-view observation. *arXiv preprint arXiv:1704.06888*, 2017.

[120] Siddhartha S. Srinivasa, Dave Ferguson, Casey J. Helfrich, Dmitry Berenson, Alvaro Collet, Rosen Diankov, Garratt Gallagher, Geoffrey Hollinger, James Kuffner, and Michael Vande Weghe. HERB: a home exploring robotic butler. *Autonomous Robots*, 28(1):5, 2010.

[121] Mike Stilman and James Kuffner. Planning among movable obstacles with artificial constraints. *The International Journal of Robotics Research*, 27(11-12):1295–1307, 2008.

[122] Halit Bener Suay, Russell Toris, and Sonia Chernova. A practical comparison of three robot learning from demonstration algorithm. *International Journal of Social Robotics*, 4(4):319–330, 2012.

[123] Zhiqiang Sui, Lingzhu Xiang, Odest C Jenkins, and Karthik Desingh. Goal-directed robot manipulation through axiomatic scene estimation. *The International Journal of Robotics Research*, 36(1):86–104, 2017.

[124] Zhiqiang Sui, Zheming Zhou, Zhen Zeng, and Odest Chadwicke Jenkins. SUM: Sequential scene understanding and manipulation. In *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pages 3281–3288. IEEE, 2017.

[125] Andrea L. Thomaz and Cynthia Breazeal. Experiments in socially guided exploration: Lessons learned in building robots that learn with and without human teachers. *Connection Science*, 20(2-3):91–110, 2008.

[126] Russell Toris, Julius Kammerl, David V. Lu, Jihoon Lee, Odest Chadwicke Jenkins, Sarah Osentoski, Mitchell Wills, and Sonia Chernova. Robot Web Tools: Efficient messaging for cloud robotics. In *IROS*, pages 4530–4537, 2015.

[127] Alexander J.B. Trevor, Suat Gedikli, Radu B. Rusu, and Henrik I. Christensen. Efficient organized point cloud segmentation with connected components. *Semantic Perception Mapping and Exploration (SPME)*, 2013.

[128] Khai N Truong, Elaine M Huang, and Gregory D Abowd. CAMP: A magnetic poetry interface for end-user programming of capture applications for the home. In *UbiComp 2004: Ubiquitous Computing*, pages 143–160. Springer, 2004.

[129] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L Littman. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 803–812. ACM, 2014.

[130] Mirko Wächter, Sebastian Schulz, Tamim Asfour, Eren Aksoy, Florentin Wörgötter, and Rüdiger Dillmann. Action sequence reproduction based on automatic segmentation and object-action complexes. In *2013 13th IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, pages 189–195. IEEE, 2013.

[131] Aaron Walsman, Weilin Wan, Tanner Schmidt, and Dieter Fox. Dynamic high resolution deformable articulated tracking. *3D Vision*, 2017.

[132] J.B. Weinberg and X. Yu. Robotics in education: Low-cost platforms for teaching integrated systems. *Robotics & Automation Magazine, IEEE*, 10(2):4–6, 2003.

[133] Evan Welbourne, Magdalena Balazinska, Gaetano Borriello, and James Fogarty. Specification and verification of complex location events with panoramic. In *Pervasive Computing*, pages 57–75. Springer, 2010.

[134] Tim Welschehold, Christian Dornhege, and Wolfram Burgard. Learning manipulation actions from human demonstrations. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, pages 3772–3777. IEEE, 2016.

[135] Thomas Whelan, Renato F. Salas-Moreno, Ben Glocker, Andrew J. Davison, and Stefan Leutenegger. ElasticFusion: Real-time dense SLAM and light source estimation. *The International Journal of Robotics Research*, 35(14):1697–1716, 2016.

[136] Willow Garage, Inc. Beer me, robot. Willow Garage, Inc. Blog. http://www.willowgarage.com/blog/2010/07/06/beer-me-robot, 2010.

[137] Willow Garage, Inc. The PR2 plays pool. Willow Garage, Inc. Blog. http://www.willowgarage.com/blog/2010/06/15/pr2-plays-pool, 2010.

[138] Melonee Wise, Michael Ferguson, Derek King, Eric Diehr, and David Dymesich. Fetch and freight: Standard platforms for service robot applications. In *Workshop on Autonomous Mobile Service Robots*, 2016.

[139] Wonder Workshop. Blockly, 2015. [Online; accessed 23-September-2015].

[140] World Wide Web Consortium. WebAssembly. https://webassembly.org, 2018.

[141] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3D ShapeNets: A deep representation for volumetric shapes. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1912–1920, 2015.

[142] M. Wüthrich, P. Pastor, M. Kalakrishnan, J. Bohg, and S. Schaal. Probabilistic object tracking using a range camera. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3195–3202. IEEE, November 2013.

[143] Keenan A. Wyrobek, Eric H. Berger, H.F. Machiel Van der Loos, and J. Kenneth Salisbury. Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 2165–2170. IEEE, 2008.

[144] Keenan Wyrokbek. Robot brings a beer. YouTube. https://youtu.be/N66olkjLmq0, 2010.

[145] Keenan Wyrokbek. Robot cleans a room (8x speed up). YouTube. https://www.youtube.com/watch?v=oyHWkQcin7I, 2010.

[146] Keenan Wyrokbek. Robot unloads a dishwasher. YouTube. https://youtu.be/9J9kxb_7dUg, 2010.

[147] Yu Xiang and Dieter Fox. DA-RNN: Semantic mapping with data associated recurrent neural networks. *arXiv preprint arXiv:1703.03098*, 2017.

[148] Zhe Xu and Maya Cakmak. Enhanced robotic cleaning with a low-cost tool attachment. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 2595–2601, 2014.

[149] Seiji Yamada and Takanori Komatsu. Designing simple and effective expression of robot's primitive minds to a human. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 2614–2619. IEEE, 2006.

[150] Wenxian Yang, Jianfei Cai, Jianmin Zheng, and Jiebo Luo. User-friendly interactive image segmentation through unified combinatorial user inputs. *IEEE Transactions on Image Processing*, 19(9):2470–2479, 2010.

[151] Andy Zeng, Shuran Song, Matthias Nießner, Matthew Fisher, Jianxiong Xiao, and Thomas Funkhouser. 3DMatch: Learning local geometric descriptors from RGB-D reconstructions. *arXiv preprint arXiv:1603.08182*, 2016.

[152] Hui Zhang and Michael J Boyles. Visual exploration and analysis of human-robot interaction rules. In *IS&T/SPIE Electronic Imaging*, pages 86540E–86540E. International Society for Optics and Photonics, 2013.

[153] Tianhao Zhang, Zoe McCarthy, Owen Jow, Dennis Lee, Ken Goldberg, and Pieter Abbeel. Deep imitation learning for complex manipulation tasks from virtual reality teleoperation. *arXiv preprint arXiv:1710.04615*, 2017.

[154] Christian Zimmermann and Thomas Brox. Learning to estimate 3D hand pose from single RGB images. In *International Conference on Computer Vision*, 2017.