Code3: A System for End-to-End Programming of Mobile Manipulator Robots for Novices and Experts

Justin Huang and Maya Cakmak
Computer Science & Engineering, University of Washington
Seattle, WA 98195
{jstn,mcakmak}@cs.washington.edu

ABSTRACT

This paper introduces *Code3*, a system for user-friendly, rapid programming of mobile manipulator robots. The system is designed to let non-roboticists and roboticists alike program end-to-end manipulation tasks. To accomplish this, *Code3* provides three integrated components for perception, manipulation, and high-level programming. The perception component helps users define a library of object and scene parts that the robot can later detect. The manipulation component lets users define actions for manipulating objects or scene parts through programming by demonstration. Finally, the high-level programming component provides a drag-and-drop interface with which users can program the logic and control flow to accomplish a task using their previously specified perception and manipulation capabilities. We present findings from an observational user study with non-roboticist programmers (N=10) that demonstrate their ability to quickly learn *Code3* and program a PR2 robot to do manipulation tasks. We also demonstrate how the system is expressive enough for an expert to rapidly program highly complex manipulation tasks like playing tic-tac-toe and reconfiguring an object to be graspable.

Keywords

Programming by demonstration; Programming tools; End-user programming; Mobile Manipulation

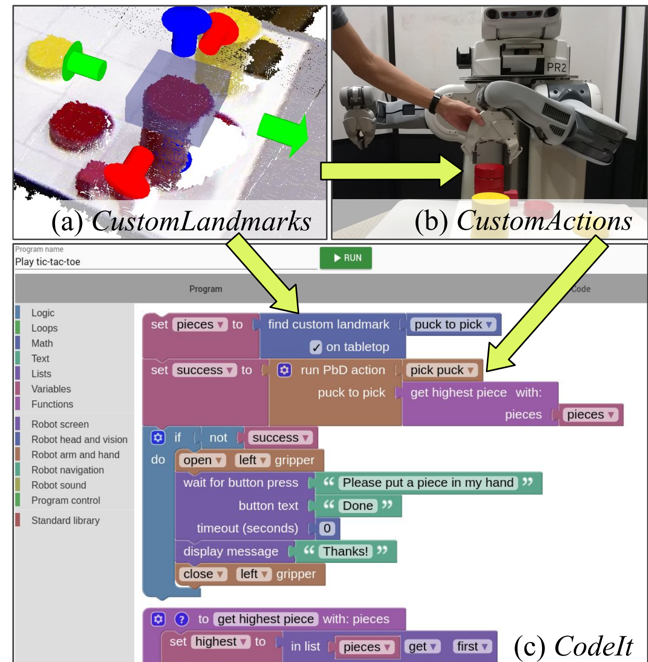

Figure 1: Overview of the *Code3* system.

1. INTRODUCTION

Programming mobile manipulator robots such as the PR2 or Fetch robots can be challenging to learn and take a long time, even for expert roboticists. Not only does it require learning a robot programming system such as ROS [33], it also requires specialized knowledge of robot perception and manipulation. These factors make robot programming inaccessible to general programmers who do not have specific skills in robotics, potentially slowing progress in the field of mobile manipulators. As an example of how long it can take, one course teaching senior computer science undergraduate students to program the PR2 with ROS required over 17 hours of lab time spread across 10 weeks [11]. Even a group of professional robotics engineers had to spend five days to program the PR2 to fetch beers from a kitchen refrigerator [1]. And, several teams of roboticists took months programming the PR2 to pick items from warehouse shelves for the Amazon Picking Challenge [15]. Our goal is to make programming mobile manipulators to do similar tasks easy for general programmers while still allowing experienced users to program more complex tasks.

This paper introduces *Code3*, a robot programming system that integrates three core components that are necessary for end-to-end programming of mobile manipulation tasks. The first, *CustomLandmarks*, lets users create a library of perceptual landmarks (objects or scene elements) by annotating the robot's sensor stream, which the robot can detect in new scenes. The second, *CustomActions*, is a kinesthetic programming by demonstration (PbD) system that lets users create a library of manipulation actions. The third component, *CodeIt*, is a drag-and-drop programming interface that lets users define control flow logic for their task

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HRI '17, March 06 - 09, 2017, Vienna, Austria

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4336-7/17/03…$15.00

DOI: http://dx.doi.org/10.1145/2909824.3020215

using the perception and action capabilities they developed with the other two components.

Through a user study with 10 participants, we demonstrate that non-roboticist programmers, who were novices to *Code3*, were able to program useful manipulation tasks on the PR2 robot in under an hour, after only 90 minutes of training. To achieve this rapid training and implementation time, we did not have to trade off expressivity by making *Code3* overly simplistic. In contrast, we illustrate how we can 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.

2. RELATED WORK

Code3 builds upon previous research in PbD and end-user programming for robotics, which we discuss below.

Programming by Demonstration: PbD is a commonly-used technique for programming robots by guiding the robot through an example of a manipulation action [5, 9, 14, 27]. Researchers have studied many different aspects of PbD, including how to represent actions [2, 13, 30, 36], learn high-level task structures [17, 28, 31], use PbD to train policies in a reinforcement learning framework [6, 35], and resolve design issues when interacting with human teachers [12, 24, 37]. User studies have shown that end-users can learn and use PbD systems to program various manipulation actions [2, 3]. Our system adds more flexible perception capabilities compared to previous PbD systems and integrates with a coding interface that enables logic and control flow.

User programming for robots: *Code3* is designed to let a broader group of people (those with general programming experience) program robots. Other systems have been developed with similar goals. The code interface for *Code3* is based on the *CustomPrograms* [23] system. *CustomPrograms* enables programmers to rapidly write programs for a mobile delivery robot using a visual programming interface. However, their system does not address the challenge of programming perception capabilities or manipulation actions.

Two related systems include ROS Commander [29] and RoboFlow [4], which integrate robot actions, including those programmed by demonstration, with a visual programming language using a data flow model [21]. Interaction Composer [19, 20] is a system that also combines robot actions (coded in C++) with a flow-based interface. The authors of [4, 20] 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 [4] 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 [25, 26], Choregraphe [32], and Interaction Blocks [34] interfaces for programming the Nao robot. They combine flow-based programming interfaces with support for timing, social dialogue, and libraries of gestures. RoboStudio [16] is a system for authoring UI and control flow for healthcare robots. Although *Code3* has some human interaction features, it focuses on programming manipulation tasks, such as fetching and carrying, delivering objects, and manipulating the environment.

3. SYSTEM DESCRIPTION

Code3 allows end-to-end programming of mobile manipulation tasks through three components: (1) *CustomLandmarks* for developing perceptual capabilities, (2) *CustomActions* for programming manipulation actions, and (3) *CodeIt* for combining the two in a high-level program that captures task requirements. This section describes these components and how they are integrated together. Our system was implemented on a PR2 robot; however, all components are easily portable to a different robot.

3.1 CustomLandmarks

Many robot programs involve detecting and locating objects. However, developing these capabilities requires expertise in robot perception or requires using automatic systems that recognize a limited set of objects. *CustomLandmarks* is a system that lets users create partial 3D models of task-relevant objects or scene parts, called *landmarks*, which the robot can locate in new scenes. This section describes how landmarks are created, represented, and searched for in new scenes. Internal evaluations, pseudocode, and other details of *CustomLandmarks* are provided in [22].

3.1.1 Landmark representation

To create a custom landmark, the user positions the robot in the task scene and points its RGBD sensor (*e.g.,* a Kinect) at the landmark of interest. The user views a 3D visualization of the data and segments the landmark by setting a 3D box around the landmark, shown in Figure 1(a). The user then gives the landmark a name, and the system records the RGBD point cloud inside the box as well as the box's position and dimensions to a database.

Custom landmarks are represented with the point cloud and the box used to segment it from the scene. The system uses the box to explictly model regions of empty space around the landmark. This adds specificity to the representation. For example, the point cloud of the corner of a tabletop does not look different from an empty patch in the middle of the table: both landmarks are flat and rectangular. By modeling empty space, the user can create a landmark that explicitly represents the corner of the table as opposed to any part of the table. Such a landmark could be used to place or align items with the corner.

This representation is flexible and allows users to create a variety of landmarks. Examples include simple objects (*e.g.,* a cup on a table), parts of objects (*e.g.,* the handle of a Tide bottle), or arbitrary parts of a task scene (*e.g.,* the front left corner of a laundry machine).

3.1.2 Search algorithm

CustomLandmarks implements an algorithm to locate a landmark in a scene. The algorithm takes a custom landmark and a point cloud representation of a scene as input, and it returns a list of locations where the landmark was found. If the algorithm does not find the landmark in the scene, it returns an empty list.

The algorithm first randomly samples points in the scene. For each sampled point, it attempts to align the landmark to the scene geometry at that point using the Iterative Closest

Point algorithm [8]. It then computes an error score for the alignment by summing two sources of error. Informally, the first source of error measures how well the shape of the landmark matches with the shape of the scene, while the second measures how much the scene intrudes on the empty space of the landmark. More formally, the first source of error is the sum of the distances between the points in the landmark's point cloud and each of their nearest points in the scene. The second is the sum of the distances between the scene points inside the landmark's box and their nearest points on the landmark. The errors are averaged to provide a single error score for the alignment. If the error score is below a certain threshold, the location of the alignment gets added to the output list. Note that the error metric does not use color information, although doing so is an area for future work.

3.1.3 Limitations

Because the landmark's shape is captured from a single RGBD point cloud, *CustomLandmarks* is sensitive to viewpoint differences, such as when an asymmetric landmark is rotated 90 degrees. To avoid this issue, users can create additional landmarks for rotated versions of the same object and search for all of those landmarks. We envision using *CustomLandmarks* (and *Code3* more generally) in semi-structured environments in which objects have known locations and a finite set of orientations.

3.2 CustomActions

Programming robot manipulation actions can also be challenging for developers, because it involves modeling the spatial positions of objects, planning arm motions, and adjusting these motions depending on the location of objects. As discussed in Section 2, PbD is a technique for defining manipulation actions that is accessible to novice users. *CustomActions* is a PbD system that builds upon the work of [3]. The main difference between the two systems is that the system of [3] uses a tabletop segmentation system to detect objects, while *CustomActions* uses *CustomLandmarks*.

Although it sounds like a minor change, using *CustomLandmarks* instead of tabletop segmentation allows users to create a much wider variety of manipulation actions than before. Tabletop segmentation searches for a nearby tabletop using the RANSAC algorithm [18]. Clusters of RGBD data points above the table are then considered to be objects. This only works in tabletop scenes and assumes that objects on the table are the only perceptual landmarks that need to be located. In contrast, using *CustomLandmarks* lets *CustomActions* locate non-tabletop objects like doorknobs and non-objects like the corner of a table.

Below, we describe the workflow for creating, representing, and executing PbD actions.

3.2.1 Action representation

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.

3.2.2 Programming an 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 4, users can define some poses to be relative to one of the landmarks they defined earlier. When done, users give the action a name, which *CodeIt* uses later.

3.2.3 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.

3.3 CodeIt

CustomActions helps users specify simple actions, but it is not expressive enough to represent complex task structure and logic like looping or branching. Our system uses *CodeIt* (an open-source and extended version of the system introduced in [23]) to enable programming of such high-level task structures. *CodeIt* implements a visual programming interface that lets users write code by connecting puzzle shaped blocks instead of typing (Fig. 1(c)). The interface provides general-purpose programming language features, including loops, conditionals, string manipulation, lists, functions, and variables. To control the robot, *CodeIt* makes calls to a high-level API that operates the robot. An example of an API method, which we call a *primitive*, is to move the robot's head to a certain pose. System developers must design and implement this API for their robot before using *CodeIt*.

The *CodeIt* interface is implemented in a web browser, enabling users to start programming the robot without having to set up a development environment. After writing a program, users click a "Run" button to execute their program on the robot. Once the program is running, the "Run" button turns into a "Stop" button that halts the program.

3.3.1 Integration of Code3 components

CodeIt provides APIs to use *CustomLandmarks* and *CustomActions*. The integration of *CodeIt* 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. *CodeIt* 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 integration of *CustomActions* with *CodeIt* 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.

CodeIt 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 *CodeIt* 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 5, we describe how we used preregistration to pick a game piece from the top of a stack. Fig. 1(c) illustrates how preregistration looks in *CodeIt*. 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.

3.3.2 Other primitives for the PR2

The work described in [23] was 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)`

Although we could have easily implemented the ability to autonomously navigate to named locations—as with the `goTo` primitive in [23]—our PR2 was immobile due to hardware issues.

3.4 Open-source code

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

4. NOVICE USER PROGRAMMING

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 or no robotics experience. To study this, we trained users to program the PR2 using *Code3* and asked them to program manipulation tasks.

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 *CodeIt*, 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 Sec. 4.1.1). 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.

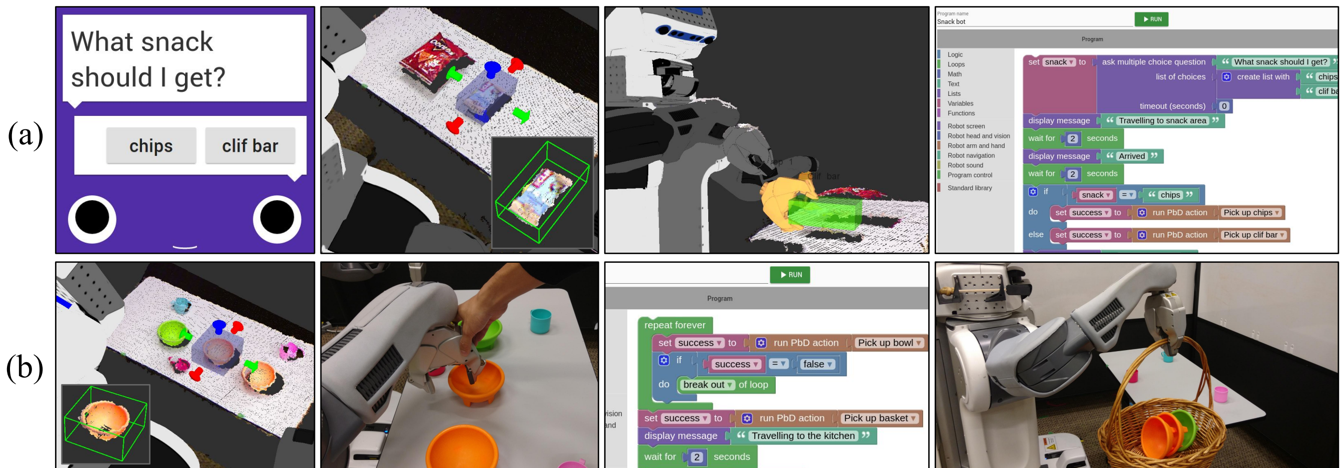

Figure 2: (a) In the *snack-bot* program, the robot fetches a user-requested snack. Users made landmarks using *CustomLandmarks* (second image, saved landmark shown inset), then demonstrated actions to pick the snacks. The third image shows the *CustomActions* GUI and the last image shows a user program made with *CodeIt*. (b) In the *waiter-bot* program, the robot places bowls into a basket and picks up the basket.

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.

4.1.1 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 *CodeIt* 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 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.

[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.

4.2 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 *CodeIt* interfaces. We also gathered video data and screen recordings.

In our final questionnaire, we first administered the System Usability Scale (SUS) survey [10]. 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 [7]. Our questionnaire also asked users to qualitatively describe what was difficult about using *CustomLandmarks*, *CustomActions*, *CodeIt*, and the *Code3* system as a whole.

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.

4.3 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.

4.3.1 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 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.

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 *CodeIt*	5m 31s
Mean time spent testing solution	5m 29s
Mean questions asked per task	1.38
Mean instances of proactive help given	1.92

Table 1: High-level summary of user study results.

4.3.2 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" [7].

4.3.3 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 *CodeIt*. Additionally, they spent an average of 5:29 minutes testing their solutions. Fig. 3 presents a visualization of 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 *CodeIt* 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 *CodeIt* 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.

4.3.4 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.

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 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 Fig. 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 (Fig. 1 and 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.

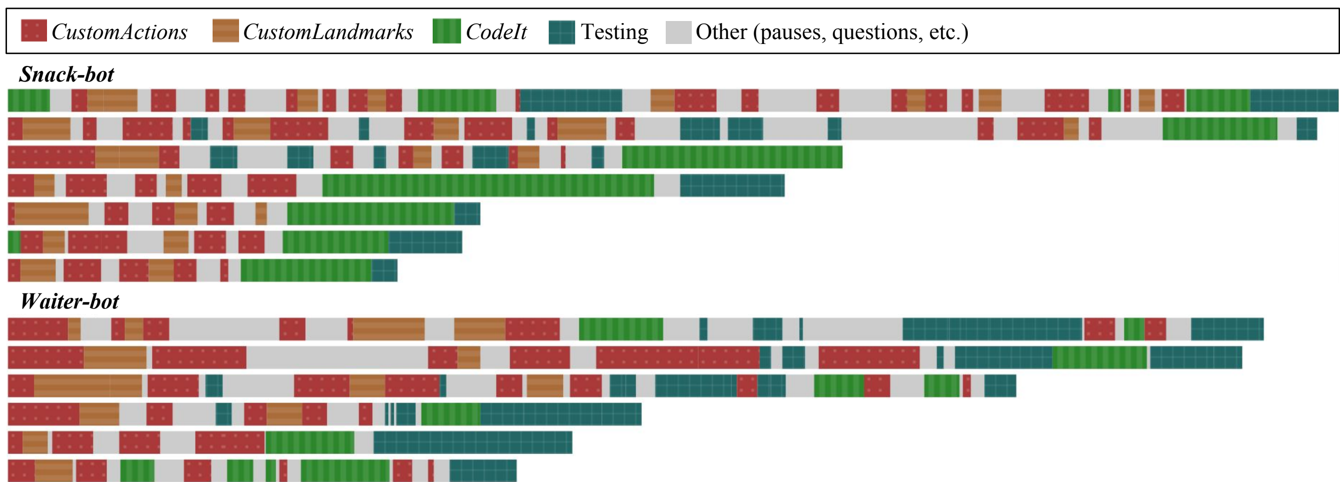

Figure 3: A visualization of which *Code3* components participants used while programming each of the user study tasks. For each row, the x-axis represents time that a participant spent working on a user study task. The color of the bars indicate which activity the user was doing; the length of the bars 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.

> *User 9:* I wish I could just drag that box instead of having to expand the edges... Otherwise I think it was pretty simple.

4.4 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 *CodeIt* 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.

5. EXPERT USER PROGRAMMING

This section describes how an expert user (the first 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*.

5.1 Tic-tac-toe

The first task we programmed was to play a game of tic-tac-toe with a human player (Fig. 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 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 *CodeIt*'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.

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.

5.2 Picking challenge

For the second task, the author programmed the robot to pick a box of crayons from a shelf (Fig. 4(b)). The box was easy to grasp while standing upright, but impossible for the

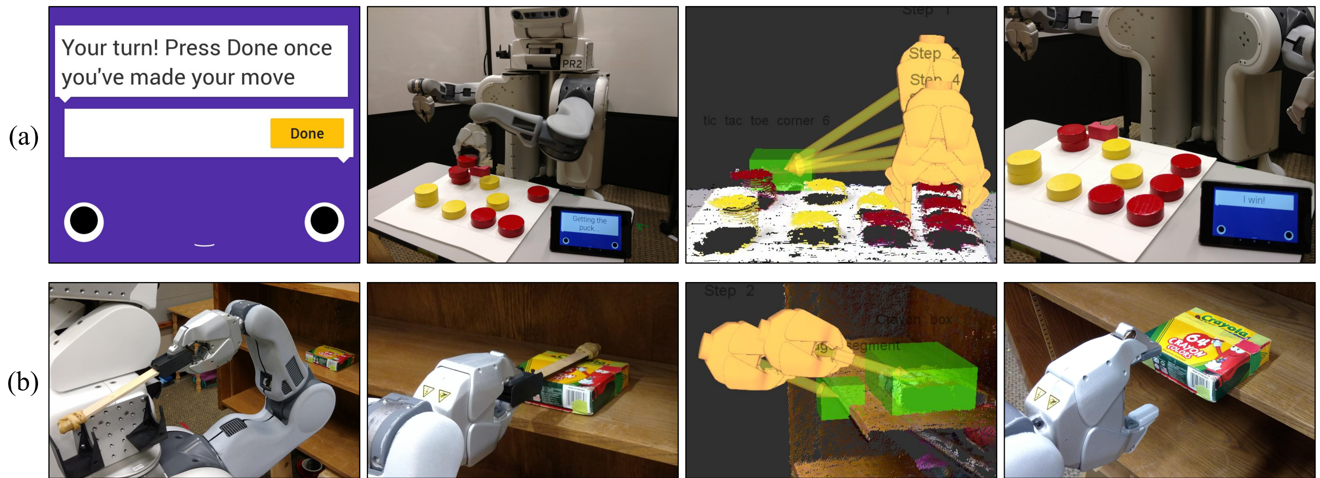

Figure 4: (a) We programmed the robot to play tic-tac-toe. The touchscreen interface facilitated turn-taking. Cell positions were known as offsets (top row, yellow arrows) from a corner landmark (green box). The robot could read the board and determine if someone had won. (b) The robot used a tool (bottom row, 1st and 2nd images) to pull a crayon box to a graspable position. The crayon box was used as a landmark to make contact with it but the edge of the shelf was used to know how far to pull (bottom row, third image).

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 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.

This task took us 2 hours and 28 minutes to program and test. To test the program, the robot had to 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.

6. CONCLUSION

This paper introduced *Code3*, a system that enables roboticists and non-roboticist programmers alike to program a mobile manipulator to do complex tasks. We described how the system is divided into tightly integrated components for perception, manipulation, and control flow. In particular, the *CustomLandmarks* and *CustomActions* components abstract the process of programming perceptual detectors and manipulation tasks in a user-friendly way. Our user study showed that non-roboticist programmers could quickly create useful programs—such as a snack-retrieving robot or a robot that cleared a table—after just 90 minutes of training with the system. The study also revealed usability issues with our interface, which we want to improve in the future. For example, we want to improve the design of the *CustomActions* user interface to make actions easier to visualize and edit, and we want to make custom landmarks reusable across multiple actions. Finally, we showed how an expert user of *Code3* could rapidly program a PR2 to perform highly complex tasks, such as playing tic-tac-toe with a human player. In the future, we want to investigate how more advanced perception and manipulation systems can be integrated into *Code3* while remaining intuitive to use. Finally, we want to add support for more robots to our open-source release.

Acknowledgements

This work was supported by the National Science Foundation, Awards IIS-1552427 "CAREER: End-User Programming of General-Purpose Robots" and IIS-1525251 "NRI: Rich Task Perception for Programming by Demonstration."

7. REFERENCES

[1] Beer me, robot. In *Willow Garage Inc. Blog. http://www.willowgarage.com/blog/2010/07/06/beer-me-robot*, 2010.

[2] B. Akgun, M. Cakmak, J. W. Yoo, and A. L. 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, 2012.

[3] S. Alexandrova, M. Cakmak, K. Hsiao, and L. Takayama. Robot programming by demonstration with interactive action visualizations. In *Robotics: Science and Systems (RSS)*, pages 48–56, 2014.

[4] S. Alexandrova, Z. Tatlock, and M. 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.

[5] B. D. Argall, S. Chernova, M. Veloso, and B. Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009.

[6] C. G. Atkeson and S. Schaal. Robot learning from demonstration. In *ICML*, volume 97, pages 12–20, 1997.

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

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

[9] A. Billard, S. Calinon, R. Dillmann, and S. Schaal. Robot programming by demonstration. In *Springer Handbook of Robotics*, pages 1371–1394. Springer, 2008.

[10] J. Brooke et al. SUS-a quick and dirty usability scale. *Usability Evaluation in Industry*, 189(194):4–7, 1996.

[11] M. Cakmak. Labs. CSE 481C website. https://sites.google.com/site/cse481a/labs, 2013.

[12] M. Cakmak and A. L. Thomaz. Eliciting good teaching from humans for machine learners. *Artificial Intelligence*, 217:198–215, 2014.

[13] S. Calinon and A. Billard. Statistical learning by imitation of competing constraints in joint space and task space. *Advanced Robotics*, 23(15):2059–2076, 2009.

[14] S. Chernova and A. L. Thomaz. Robot learning from human teachers. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(3):1–121, 2014.

[15] N. Correll, K. E. Bekris, D. Berenson, O. Brock, A. Causo, K. Hauser, K. Okada, A. Rodriguez, J. M. Romano, and P. R. Wurman. Analysis and observations from the first Amazon Picking Challenge. *IEEE Transactions on Automation Science and Engineering*, 2016.

[16] C. Datta, C. Jayawardena, I. H. Kuo, and B. 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.

[17] S. Ekvall and D. Kragic. Robot learning from demonstration: A task-level planning approach. *International Journal of Advanced Robotic Systems*, 5(3):223–234, 2008.

[18] M. A. Fischler and R. C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[19] D. Glas, S. Satake, T. Kanda, and N. Hagita. An interaction design framework for social robots. In *Robotics: Science and Systems*, volume 7, page 89, 2012.

[20] D. F. Glas, T. Kanda, and H. 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.

[21] D. D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, 3(1):69–101, 1992.

[22] J. Huang and M. Cakmak. Programming by demonstration with user-specified perceptual landmarks. *arXiv preprint arXiv:1612.00565*, 2016.

[23] J. Huang, T. Lau, and M. Cakmak. Design and evaluation of a rapid programming system for service robots. In *2016 11th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 295–302. IEEE, 2016.

[24] E. S. Kim, D. Leyzberg, K. M. Tsui, and B. 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.

[25] T. 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.

[26] T. Lourens and E. 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.

[27] T. Lozano-Perez. Robot programming. *Proceedings of the IEEE*, 71(7):821–841, 1983.

[28] A. Mohseni-Kabir, C. Rich, S. Chernova, C. L. Sidner, and D. 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.

[29] H. Nguyen, M. Ciocarlie, K. Hsiao, and C. 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.

[30] S. Niekum, S. Osentoski, G. Konidaris, and A. 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, 2012.

[31] M. Pardowitz, S. Knoop, R. Dillmann, and R. 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.

[32] E. Pot, J. Monceaux, R. Gelin, and B. 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.

[33] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, volume 3, page 5. Kobe, Japan, 2009.

[34] A. Sauppé and B. 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.

[35] S. Schaal, J. Peters, J. Nakanishi, and A. Ijspeert. Learning movement primitives. In *International Symposium on Robotics Research.*, pages 561–572, 2005.

[36] H. B. Suay, R. Toris, and S. Chernova. A practical comparison of three robot learning from demonstration algorithm. *International Journal of Social Robotics*, 4(4):319–330, 2012.

[37] A. L. Thomaz and C. 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.