RobotIST: Interactive Situated Tangible Robot Programming

Yasaman S. Sefidgar*
University of Washington
einsian@cs.washington.edu

Thomas Weng*
University of Washington
tweng@cs.washington.edu

Heather Harvey
University of Washington
harvh@cs.washington.edu

Sarah Elliott
University of Washington
sksellio@cs.washington.edu

Maya Cakmak
University of Washington
cakmak@cs.washington.edu

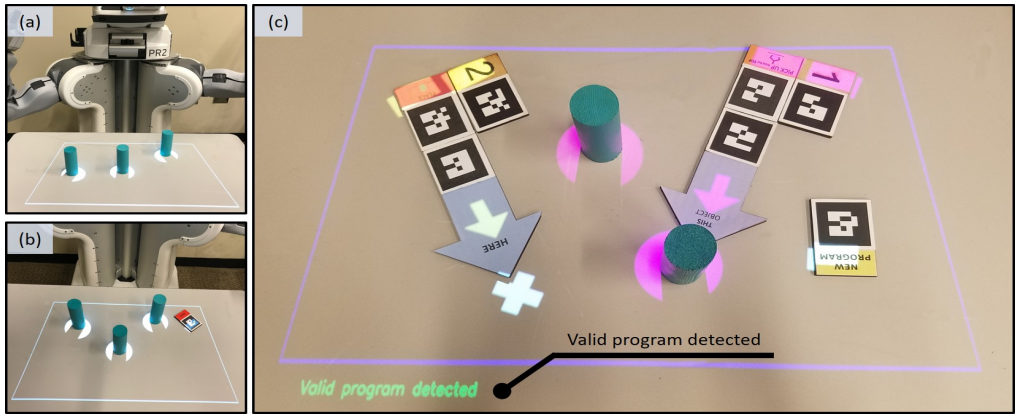

Figure 1: Overview of the RobotIST system for interactive situated programming of robot manipulation tasks using *tangible blocks*. Programmer places the blocks in the robot's workspace in reference to the locations and objects of interest. Robot projects its understanding of the environment and the program onto the workspace in real-time. (a) RobotIST highlights the workspace as well as the objects detected in the environment through *workspace* and *environment perception projections*. (b) it additionally highlights *tangible blocks*. (c) in the *programming mode*, workspace projection turns blue and the robot highlights different instructions, each in a different color. It additionally represents their semantics. For example, objects of interest are highlighted in the same color as the instruction referencing them. Similarly, the cross-hair indicates the location the system considers under the instruction. The program depicted above allows the robot to stack all green cylinders at the location of the cross-hair. The system confirms that it has identified this program by a message in the *status bar* (*Valid program detected*).

ABSTRACT

Situated tangible robot programming allows programmers to reference parts of the workspace relevant to the task by indicating objects, locations, and regions of interest using tangible blocks. While it takes advantage of situatedness compared to traditional text-based and visual programming tools, it does not allow programmers to inspect what the robot detects in the workspace, nor to understand any programming or execution errors that may arise. In this work we propose to use a projector mounted on the robot to provide such functionality. This allows us to provide an interactive situated tangible programming experience, taking advantage of situatedness, both in user input and system output, to reference parts of the robot workspace. We describe an implementation and evaluation of this approach, highlighting its differences from traditional robot programming.

CCS CONCEPTS

• **Human-centered computing** → **Interactive systems and tools**;

KEYWORDS

Robot Programming; Tangible Programming; Situated Programming; Direct Manipulation; Transparency

ACM Reference Format:
Yasaman S. Sefidgar, Thomas Weng, Heather Harvey, Sarah Elliott, and Maya Cakmak. 2018. RobotIST: Interactive Situated Tangible Robot Programming. In *Symposium on Spatial User Interaction (SUI '18), October 13–14, 2018, Berlin, Germany.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3267782.3267921

*Y.S. Sefidgar and T. Weng contributed equally to this work.

SUI '18, October 13–14, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5708-1/18/10…$15.00
https://doi.org/10.1145/3267782.3267921

1 INTRODUCTION

The key advantage of robotic manipulators over traditional automation is their programmability [32]. The same robot can be programmed to assemble phones in one factory and paint airplanes or package food in another. Programming robots to perform new tasks currently requires advanced knowledge and can take considerable time, even for experts. This dependence on experts and long development cycles are real barriers for small-to-medium enterprises who cannot afford to hire dedicated personnel, as well as for industries that require customized, small-batch production.

But why is programming robots challenging? While several factors contribute to the difficulty and uniqueness of robot programming, such as the difficulty of specifying motions with code, the friction of resetting a physical environment for testing, and high degrees of concurrency, one core challenge is *referencing task-relevant parts of the environment* (objects or locations) that the robot needs to interact with. Existing systems require deep understanding of the robot's perception system, specialized visualization tools to inspect the outcome of the robot's perception, and knowledge of coordinate frames, transform algebra, and robot kinematics. In contrast, people can instruct each other about spatial tasks with ease, referencing task-relevant parts of the environment using referential gestures. But interpreting such ambiguous input from humans is still an open problem.

Situated tangible programming [46] was proposed as a way to unambiguously reference parts of the environment using the so-called tangible blocks that can be robustly detected by the robot. The idea is to create robot programs simply by placing blocks in the robot's workspace to not only reference objects and locations in the environment, but also instruct the robot about what to do with them. This approach takes advantage of the intuitiveness of situated programming; however, it sacrifices many functionalities of modern integrated development environments (IDEs), such as real-time feedback on the correctness or completeness of a program. While it is possible to augment situated tangible programming with a screen-based interface, this approach re-introduces the challenges of establishing correspondence between the environment and the created program.

Instead, we propose to use *situated feedback* provided by the robot in real-time through projections onto its workspace to provide various IDE functionalities. We present RobotIST (Figure 1), a system that allows *interactive* situated tangible programming by combining situated input from the programmer to the robot, with situated feedback from the robot to the programmer. In this paper, we describe the implementation of RobotIST and walk through its situated IDE functionalities in the context of programming various robot tasks. We also present findings from an observational study of eight robot programmers with prior experience with traditional robot programming tools who used RobotIST to program a sample task, with aspects common to many industrial settings.

2 RELATED WORK

The seminal survey by Lozano-Perez [32] characterized existing robot programming systems at the time as using a mix of three approaches: guide-through programming, robot-level programming, and task-level programming. Most industrial robots today (Kuka, ABB, Schunck, Universal Robots), just like those surveyed three decades ago, are programmed with a combination of guide-through and robot-level programming with domain specific languages and libraries. Most of the systems for programming them involve a GUI and peripheral devices, such as pendants to aid the complex robot programming and debugging process. More recent robots, such as Baxter/Sawyer and Franka, which are targeted for safe human-robot collaboration, place larger emphasis on making their programming interface more intuitive and accessible to non-experts. The last decade also witnessed the open source robotics movement, with the spread of the Robot Operating System (ROS)[41] across robotics companies and research labs, which now use common frameworks and tools for programming robots.

Most research in robotics in the last three decades has focused on enabling task-level programming and increasing the generalizability of the created programs. In particular, research in autonomous grasping [8, 48], motion planning [16, 30], and reinforcement learning [28] have all contributed to systems that synthesize robot behavior given a task-level goal. In addition, a large body of research under the umbrella of robot programming by demonstration (PbD) focused on learning generalizable robot actions from multiple demonstrations provided with guide-through interfaces [5, 9]. While most of these have historically been evaluated with expert programmers, recent work has started to explore their usability for novice users [1, 2, 12, 29, 50]. Besides programming by demonstration, a recently popularized end-user programming technique in robotics has been visual programming [3, 6, 14, 15, 19, 24, 45]. However, corresponding the 3D environment to the 2D representation on GUIs remains a challenge. There are also systems that have looked at programming robots with natural language [10, 18, 34, 36, 37].

The idea of tangible programming has been explored in the field of interaction design [47, 52], especially targeting early computer science education [17, 21, 22, 26, 35, 49]. Comparative studies have demonstrated several benefits of tangible programming over alternatives [23, 44]. Several commercial products for children informed by this research are already available. Other work has contributed toolkits or SDKs that lower the barrier for creating tangible interfaces [27, 53], that could then allow programming with tangibles. The issue of interactivity and feedback has been raised in many of the tangible programming interfaces [17, 21, 22], and alternative approaches to address them have been introduced. For example, Beckman & Dey used a dedicated predictive display to demonstrate the result of tangibly programmed smart-home control rules [7]. Some work involved translucent tiles on interactive tabletops that can directly display information on the sensed tile [42, 54]. The use of projected feedback with tangibles was proposed in a few systems [38].

While tangible programming has been applied to simple toy robots [22] and wearables [26], the use of tangibles in robotics has been limited. Luria *et al.* proposed using tangibles to command a smart-home robot non-verbally [33]. Sefidgar *et al.* introduced the idea of using tangibles for situated communication to reference objects, locations, and regions in the robot's workspace as part of programming [46]. Recent work in robotics explored the use of projections from the robot for situated communication [4, 11, 13, 31]. Our work aims to bring interactivity to situated and tangible robot programming through similar use of projected feedback.

3 CHALLENGES IN TRADITIONAL ROBOT PROGRAMMING

Lozano-Perez distinguished between four requirements of robot programming: sensing, world modeling, motion specification, and flow of control [32]. *Sensing* allows a robot to obtain identity, position, and features of objects in its environment; initiate and terminate motions; choose among alternative actions; and comply to external constraints. *World modeling* involves specification of task-relevant entities in the environment within one coordinate frame, including sensed objects with variable poses and fixed points or regions on the robot's workspace. Poses of objects, locations, and the robot's manipulator parts are often expressed with a homogeneous transform, represented with a 4x4 matrix. The transform represents the relation between coordinate frames with translation of the origin and rotations of the axes. *Motion specification* refers to describing the actual motion of the robot relative to task-relevant entities in the environment. As discussed in section 2, guide-through programming provides an efficient and intuitive way for specifying desired robot poses or full motion trajectories. *Flow of control* is the specification of high-level robot behavior, such as branching (deciding among alternative actions) or looping (repeating an action variable times) based on sensed information.

In the following subsections, we highlight the challenges in robot programming that traditional tools fail to address in relation to various elements enumerated above. We use as a working example a stacking task where all green cylinders are stacked up at a certain location (Figure 2). A common part of industrial tasks for which robots are widely used (*e.g.,* machine tending or packaging), this task illustrates important aspects of many robot programs.

We assume that the robot has a perception system that can detect and classify objects in its workspace with some accuracy. We also assume that simple actions for manipulating the objects, such as picking them up in different ways from detected poses and placing them at different target poses has already been programmed, *e.g.,* by guiding the robot through demonstrations. The robot program created for the stacking task should instruct the robot to sense the environment and evaluate the presence of green cylinders among the detected objects. The robot should then adjust its grippers to pick up the cylinder and move it to the location of interest, specified as a known constant in the program, and place it there, while adjusting for the height of the existing stack. Repeating these steps, the robot can stack all green cylinders present in the workspace at the desired location.

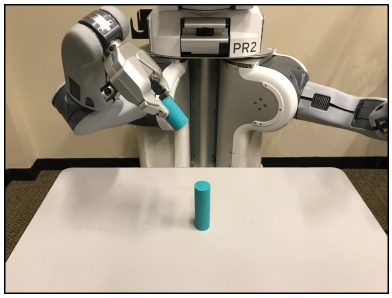

Figure 2: A PR2 robot executing the task for stacking all green cylinders at a certain point.

3.1 Specifying Objects and Locations of Interest

Although reading raw RGBD data from camera and segmenting it are usually encapsulated in high level modules and hidden from the programmer, most systems do not provide an easy way of *'using'* the visual information extracted from the environment, *e.g.,* they rely on defining coordinate frames of objects and their transformations. This requires mathematical expertise and is time-consuming. Moreover, these systems do not allow an easy way of accurately defining what object or location is relevant to various steps of the task. For example, they force the programmer to choose the object of interest from the list of all segmented objects, using numeric indices with no intrinsic meaning as reference (*e.g.,* object #5 for the green cylinder). Building on [46], RobotIST allows programmers to specify objects of a particular type as well as desired locations, and instruct the robot to perform actions at those places. RobotIST complements [46] by providing situated and immediate feedback of robot's perception for more informed and effective referencing of the environment as discussed below.

3.2 Expressing the Task

Any programming tool enforces programmers to represent their intent in the language of that tool (*e.g.,* through tool-specific instructions). Existing robot programming tools provide little support for facilitating the expression of different tasks in their language. For most commercial programming systems, programmers usually need to attend training workshops and consult the programming manual in order to learn how to express tasks in the language of the systems. There is no provision within the system to help programmers explore and understand the semantics of different instructions to help them translate the task using those semantics. RobotIST is based on the same language as [46] and allows the expression of the same tasks within that language. However, unlike [46], it communicates the semantics of different instructions both as the program is being constructed as well as during execution, making it easier for the programmer to learn and use the language.

3.3 Interpreting Robot's Behaviour

As with any type of program, errors can occur in a robot program. Programming tools should help programmers understand what causes these errors and how to address them.

3.3.1 Handling Perception Errors. Errors are inevitable in any recognition-based system that extracts information from sensor data. Robot programming systems are no exception. For example, there are usually inaccuracies in the size of segmented objects. That is, the robot's perception of the green cylinder can be much larger or smaller than the real one. Using this inaccurate information, the robot fails at adjusting its arm and grippers to properly grasp the cylinder. Unfortunately, very few robot programming systems provide support for the programmer to understand and address perception errors. Existing support is mostly available in research prototypes and usually comes in the form of perception information overlaid on the 3D feed of the environment (*e.g.,* from a robot's view or an overhead camera view). Navigating this information,

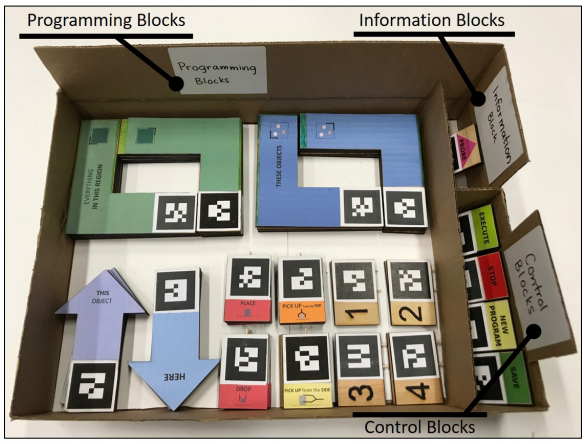

Figure 3: The full set of tangible blocks supported in RobotIST. *Programming blocks* of [46] are extended by *control* and *information* blocks in RobotIST. Refer to [46] for the space of tasks that can be expressed using the *programming blocks*, and for details on the shape and color of the blocks. See Figure 1 for an example program with these blocks.

Block Type	Name	Description
Programming	Selection	Specifies an object, set of objects, location, or region
	Action	Specifies the action applied to an object, location, or region
	Ordering	Defines the ordering of actions
Control	New	Switches to *programming* mode to create a new program
	Save	Saves the existing program if valid and switches to *idle* mode
	Execute	Switches to *execution* mode and executes a program in a loop
	Stop	Stops the execution and switches to *idle* mode
Information	Probe	Provides more information about the error when pointed to it

Table 1: Three types of blocks in RobotIST for programming, control, and information probing.

which is presented in 2D on a screen, is challenging and requires switching between viewpoints to fully understand the 3D environment. RobotIST communicates the robot's internal representation of the environment to the programmer in 3D space, making the potential perception errors easily available for examination. The programmer can thus painlessly interpret the robot's behavior.

3.3.2 Handling Misspecification Errors. A robot's undesired behavior is sometimes associated with errors in task specification. Existing robot programming tools provide little support to help the programmer understand what needs to change in their specification to achieve the desired behavior. Lack of support for understanding the semantics of instructions makes it difficult to know what changes are necessary to achieve the intended behavior. Undetected perception errors can lead to unexpected behavior even if the semantics are correct. For example, if the green cylinder is identified as a green rectangle during programming, the robot will not behave as expected. RobotIST's presentation of the environment as perceived by the robot, as well as the semantics of a program, help address misspecification issues from both perceptual and semantic sources.

4 WALK-THOURGH OF THE ROBOTIST SYSTEM

RobotIST builds upon the situated tangible programming language designed by Sefidgar *et al.* [46] to express a range of pick and place tasks common in industry. This language involves three types of tangible blocks: *selector* blocks to indicate objects, locations, or regions in the environment; *action* blocks to specify actions parametrized by objects, locations, or regions; and *order* blocks to specify the ordering of instructions. RobotIST extends this set of *programming blocks* with *control blocks* that allow switching between modes of the system and an *information block* that facilitates live interactions and feedback at the desired level of detail. The physical implementation of the blocks are shown in Figure 3 and their functionality is summarized in Table 1. Providing interactivity and situated feedback, RobotIST helps programmers more easily understand the semantics of the language, create and control programs, and understand errors they encounter while programming or at run-time. Next, we illustrate these key RobotIST features in various programming activities from the creation of a program to executing it. We use the simple stacking task introduced in Section 3, in which we instruct the robot to stack objects of a certain type (*i.e.,* green cylinders) at a specified location.

4.1 Program Creation

RobotIST projects the boundaries of the robot's operation space on the workspace to help the programmer understand what part of the workspace is visible and reachable to the robot. These are referred to as *workspace projections* (Figure 1 (a)). In addition, RobotIST highlights any objects that the robot has detected in the workspace; these are called *environment perception projections*. These allow programmers to see undetected objects and be aware of any inaccuracies in the existing detections. They can thus reconfigure the workspace to make sure all relevant objects are properly detected.

The programmer creates a new program by placing the *new-program* block on the workspace. This block is one of the four *control blocks* that allow changing the mode of the RobotIST system. RobotIST acknowledges the detection of any tangible block placed within the robot's workspace by projecting a bright white light on it. These are referred to as *block projections* (Figure 1 (b)). This conveys to the programmer any failures in detecting the tangible blocks and saves her/him the frustration of not knowing why the robot does not respond as expected. RobotIST modifies the color of the workspace projection to communicate that it is in the *programming* mode.

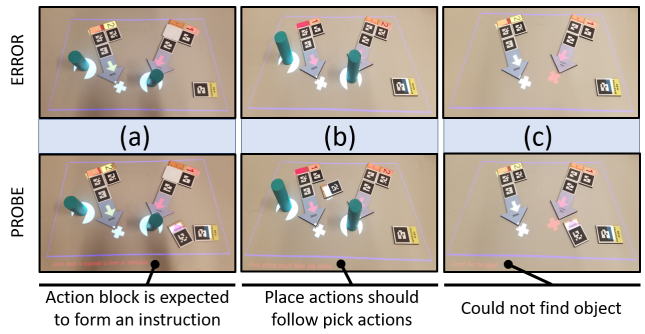

Figure 4: Errors encountered during program creation. (a) the action block is covered and is not detected (top image). By pointing the *Probe* block we get more information about the error in the *status bar* (bottom image). (b) pick and place instructions are ordered incorrectly (top) and the *Probe* provides the additional information (bottom). (c) the instruction looks for the object of interest and does not find it. The area where the object is expected is highlighted in red (top) and probing explains what has gone wrong (bottom).

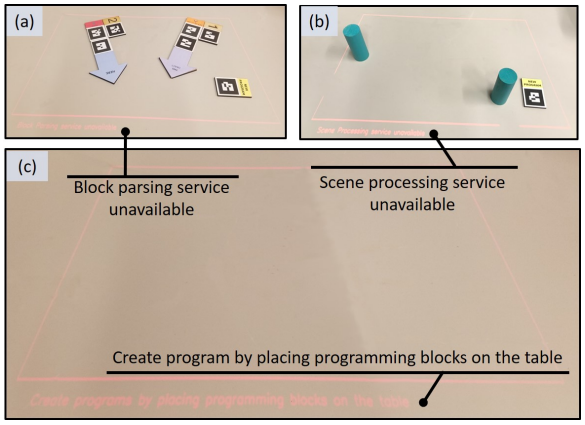

Figure 5: Internal functionality errors. The workspace boundaries turn red to inform the programmer of serious internal failures with messages in the *status bar* providing additional information. (a) the system cannot detect *tangible blocks*. (b) objects cannot be detected (no highlights) because the object processing has failed. (c) the system is expecting programming blocks in the *programming mode*.

Next the programmer uses a combination of *programming blocks* (*selection*, *action*, and *ordering*) to create instructions that make up a program. As hinted in the previous section, this involves an *object selection* block to indicate green cylinders and a *location selection* block to indicate the target pose where the cylinders are to be stacked. The programmer attaches a "pick up" action block to the object selector and a "place" action block to the location selector. Dowel pins and holes on the side of the action and selection blocks constrain how the blocks fit together, minimizing "syntax errors." The *ordering blocks* communicate to the robot in what order it should perform these actions. Any pick action in a program is expected to be followed by a place action.

When various blocks and objects are successfully detected and grouped into instructions, RobotIST indicates this grouping by changing the color of block and object projections. Each group gets a different color excluding white and red. For example, when an object selector block is grouped with a detected object, that object and the selector block are highlighted in pink. In addition, any other object of the same type (*e.g.*, green cylinders) are also highlighted in pink, albeit a bit dimmer than the one immediately in front of the arrow. Further, when the pick action is attached to the selector, the action block projection also becomes pink (Figure 1 (c)). Similarly, the projections associated with the location selector and the place action applied to it are highlighted in orange. We refer to the grouping of instructions in RobotIST as *instruction projections*. Through instruction projections, RobotIST improves the programmer's understanding of the semantics of tangible blocks and whether they have successfully expressed their intent using the blocks, a particularly helpful feature for those new to programming with the blocks.

When there are errors in forming instruction or matching blocks with objects, RobotIST highlights where the error occurs in red (*error projections*). Error projections help the programmer understand the errors as a prerequisite to debugging. For example, if block detection fails at detecting the action block of the pick instruction in Figure 1, RobotIST highlights in red both the selection and ordering blocks that are expected to connect to the action block (Figure 4 (a)). Similarly, errors in orderings—such as a place instruction being ordered before any pick instruction, or two consecutive pick instructions—are indicated by highlighting the problematic instruction entirely in red (Figure 4 (b)). If object detection fails and objects relevant to single or multiple object selector blocks are not found, RobotIST highlights the area searched in red to find the objects (Figure 4 (c)).

Additional information about what caused a particular error can be obtained by placing *probe block* in the workspace, pointing towards blocks that are highlighted with an error. For example, when the *probe block* is pointed to the selection or number blocks highlighted in red, RobotIST projects additional information inside a box at the edge of the workspace, explaining that the robot expected an action block (Figure 4). This is akin to a *status bar*. In other words, we first provide an overview of error information (red highlighting) and then allow the programmer to probe for more details if the fix is not obvious, visualizing details only on demand.

In the absence of errors, RobotIST informs the programmer on the status bar that it has identified a valid program. At this point the programmer can save the program by placing the *save block* in the workspace. The program is saved by a default name and is available even after the system is reset. Placing the *save block* additionally causes the system to enter the *idle mode*, when it no longer modifies the saved program.

4.2 Program Execution and Debugging

Having created a program, we can execute it by placing the *execution block* on the workspace. This will execute the last modified program. As with any other block, RobotIST acknowledges the detection of the *execution block* by highlighting it. Once the block is

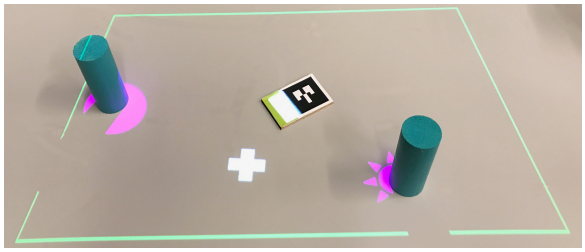

Figure 6: Projections at the time of execution. Workspace boundaries are green to indicate the system is in the *execution mode*. Program semantics are projected for the stacking task (Section 4): the objects of interest are highlighted in the same color and the location of interest is highlighted by a crosshair. The object that the robot is about to pick is signaled by changing the style of projections.

detected, the system enters the *execution mode*. This change is visually indicated by variation in the style of the workspace boundaries as shown in Figure 6.

Similar to program creation, RobotIST highlights all objects detected on the workspace to help the programmer understand the robot's internal representation of the environment (*environment perception projections*). It additionally projects information about the locations and objects relevant to the program (*program projections*). For the stacking program described earlier, any instance of the green cylinder and the location where the cylinders are to be stacked are all highlighted (Figure 6). As before, instructions are grouped by color. By presenting the program related information during execution, RobotIST helps the programmer better reason about the robot's behavior in relation to the program instructions.

Projections associated with objects or locations that the robot is about to act on change in style to inform the programmer where to expect an action, *i.e.,* pick or place (*execution projection*). This makes the robot's execution process transparent to the programmer (Figure 6).

If there are errors at run-time, RobotIST highlights where it encountered them in red (*error projections*) (Figure 7 (a)). For example, if the robot cannot reach a green cylinder it intended to pick, the cylinder is highlighted in red (Figure 7 (b)). As before, the *probe block* can be used to obtain additional information by pointing it to where the error is highlighted. The entire workspace is highlighted in red if no green cylinder is found. (Figure 7 (c)).

The programmer can stop the execution at any time by placing the *stop block* on the workspace. After acknowledging the detection of the *stop block*, RobotIST stops all motion and enters its *idle mode*. Execution of the program is resumed from where it was left off if the *execute block* reappears.

5 IMPLEMENTATION

RobotIST can be implemented on robot manipulator platforms with visual perception and basic contact control capabilities. Projections can be achieved with a robot-mounted or overhead projector. In this paper we used a PR2 robot for our proof-of-concept implementation detailed below.

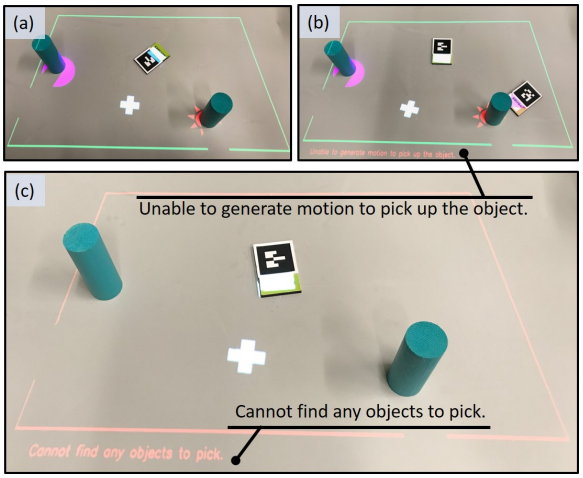

Figure 7: Errors at the time of execution. (a) the robot cannot pick the green cylinder at the bottom right. (b) probing provides additional information about the issue. (c) the robot cannot find any object to pick.

5.1 Robot Hardware and Software

PR2 is a 14 degree of freedom (DoF) dual-arm robot (7 DoFs per arm) on an omni-directional base. The 1-DoF grippers can open to a maximum of 9cm. We used the right arm only but the implementation can be extended to both arms. We implemented the system within the ROS framework, using off-the-shelf libraries such as PCL for tabletop segmentation, Alvar for AR tag tracking, and MoveIt! for motion planning [40, 43, 51].

5.2 Projections

We mounted a 500-lumen portable projector on the head of the PR2 to project textures onto the scene. We used OpenCV [25] to create the images displayed through the projector. The projector produces no light for pure black image pixels, enabling us to project light selectively on only relevant regions and objects in the scene. We use the pinhole camera model to represent the projector's intrinsic parameters. We assume the position of the robot is fixed and provide no projector-camera calibration routine, though prior work on projector-camera calibration exists and can be implemented for a mobile robot [39].

Projected shapes appear as flat shapes on the surfaces of the objects, blocks, and the workspace. Since our robot-mounted projector is at an angle to the target surfaces, a homographic transformation is required to produce shapes that are geometrically correct in the real world (See Figure 8). To accomplish this transformation, we find and apply the graphical projection from the desired real-world poses of the shapes onto the projector's output image plane [20].

5.3 Architecture

Figure 9 depicts the overall system architecture. A Microsoft Kinect mounted on the PR2 provides color images for tracking the AR tags associated with each programming block, and also provides depth clouds for identifying objects by table-top segmentation.

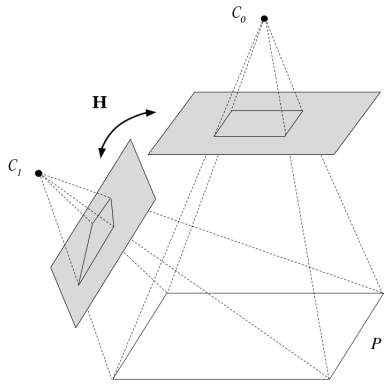

Figure 8: Depiction of the homographic transformation H, from the axis-aligned 2D projection of an initial frame C_0 to the off-axis projection of the robot-mounted projector's frame C_1 for texture P.

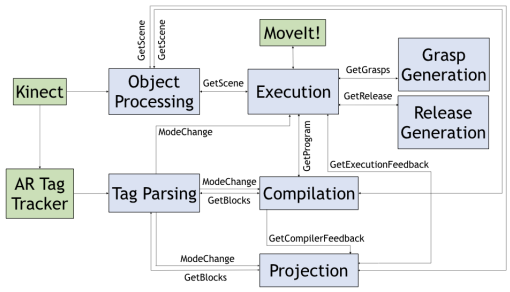

Figure 9: Diagram of the RobotIST software architecture. The Tag Parsing node detects mode blocks present in the scene and broadcasts a mode change to control the program flow. In *idle mode*, the projection node directly receives and visualizes scene items detected by the Tag Parsing and Object Processing nodes, whereas in *programming* and *execution* modes the items are first processed by the Compilation and Execution nodes, respectively.

The detected tags are processed by the tag parser, which matches each tag with its semantic meaning as a tangible block. The tag parser also maintains the mode of the system (*idle*, *programming*, or *execution*), updating the mode when it detects a new control block in the scene. The system mode determines how blocks and objects are parsed and processed by the compilation, execution, and projection nodes.

In *programming mode*, objects and blocks are passed to the compilation node, which attempts to compile a program by grouping related blocks and objects together to form instructions. It passes these groups to the projection node, which projects each instruction-forming set of objects and blocks in a unique color. If the compilation node fails to construct a valid program, it passes the errors associated with each item in the workspace to the projection node, which colors them red to denote an error. The probe block can then be used in this state to select and display each error.

In *execution mode*, objects and blocks are passed to the execution node, which runs a saved program over the current objects in the workspace. The node also sends instruction-based groupings of objects, locations, and blocks to the projection node for visualization. Objects that are next in line to be manipulated by the robot are accented with a unique projection. Errors in the workspace stop execution of the saved program and are similarly passed to the projection node so the relevant elements are colored in red.

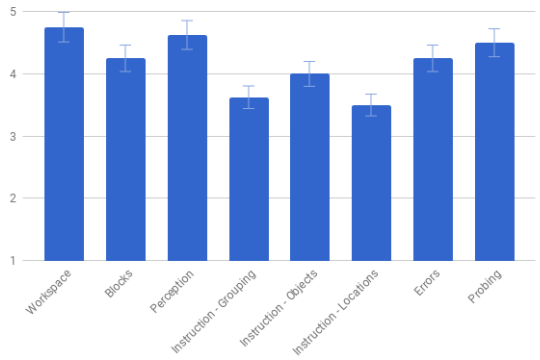

Figure 10: Usefulness ratings averaged across eight participants for different RobotIST features. Participants rated each feature on a 5-point Likert scale, ranging from 1 (strongly disagree) to 5 (strongly agree).

In *idle mode*, objects and blocks are directly passed to the projection node to generate simple, white projections that denote what the robot detects in the workspace, but display neither color-based grouping nor error information, since the robot is in standby.

6 EVALUATION

We performed an observational study to better understand the use of RobotIST. Our system not only lowers the barrier to robot programming, but also makes it more efficient to create robot programs. Therefore, it benefits both novice and experienced programmers; it enables the former to create programs they would not be able to create using traditional tools and it makes it faster and less arduous for the latter to program. Our observations reported in this paper are focused on what experts gain using our tool. We recruited 8 participants (1 female), ages ranging from 21-34 (mean=26.75), all recruited from the same university department. While all participants had computer science backgrounds and seven had previously programmed manipulator robots, none had prior experience with the situated tangible programming paradigm.

Following a short training about the tangible programming language and features of RobotIST, we asked each participant to create a program for the stacking task. We then asked them to fill out a questionnaire about their experience with RobotIST, asking what they found helpful, what was challenging, and suggestions for improvement. For participants who had prior experience programming robots, we also asked how RobotIST compares to other tools they use for programming robots.

All participants successfully created a program for the stacking task within 4 to 8 minutes (approximately 6.5 minutes on average). All participants found programming task to be easy using RobotIST. They noted both situatedness and feedback as what they liked about the system; *e.g., "No coding required - I was just able to program*

visually, which is way faster" and *"I liked that I was able to see what the robot understood via the highlighting function. It was relatively intuitive."*

While participants positively rated all RobotIST features, *workspace* and *environment perception projections* were rated as most helpful (Figure 10). Describing how different features helped them, they elaborated on their ratings. *Workspace projection* was popular as it helped the participants understand where they can place tangible blocks and objects to create their programs. *Block* and *object projections* were indicated as being helpful in figuring out whether the robot "knows" what is in front of it and can help with debugging. This was clearly articulated in participants' comments: *e.g., "I was able to debug my program when it could not see a certain block."* Participants described *instruction projections* as a way to both *"sanity check"* and ensure the robot's interpretations of the tangible instructions matched their expectation: *"This was especially useful in the scenario that uses 'This Object' (object selection) blocks, since it shows that the robot did extrapolate that I meant for it to see multiple instances of a specific kind of object."* Participants seemed to be confident that the robot reliably identified the locations they referenced and did not find the projections of location helpful. This is reflected both in their ratings (*instruction projections* of locations got the lowest rating of 3.5) and comments; many said they barely noticed the feature.

Error projections were also described as helpful; *e.g., "Very helpful, since I was able to see what exactly was the issue with the program instead of having to guess.* The probing feature was also popular as it helped them get only the necessary information about what was wrong: *"I liked how this could give me an error about a *specific* part of the program, which let me march through errors."*

When comparing with other robot programming tools, participants appreciated the ease of programming and feedback *"I liked that I received a lot of feedback in an intuitive way (sight) and being able to see what the robot "understood." As a "hands on" type of person I find this style of programming quite enjoyable"* but raised concerns about the expressiveness *"Faster, but more limited in the scope of what it can do. I don't need to hardcode a bunch of positions, but it's obviously not as expressive as something like Python."* They nonetheless recognized the value of being able to quickly prototype robot programs: *"It's a bit limited in expressive power in comparison to a traditional programming language, but intuitive and nice for quickly putting together high-level tasks."*

7 DISCUSSION

RobotIST leverages the situatedness that is inherent to physical manipulation tasks to address many of the major challenges in traditional robot programming. With RobotIST, programmers not only express the manipulation in the physical environment, where it belongs, but also receive feedback in that environment. RobotIST communicates the semantics of the underlying programming language in real-time to help the programmer better understand how different instructions impact the overall program. RobotIST also allows users to obtain a better mental model of the system's capabilities, by adding transparency about what the robot perceives or how it groups different blocks with objects.

Situated tangible programming is unique in allowing programmers to accurately and unambiguously reference objects and locations where the manipulation happens, and to do so naturally with no intermediate abstraction of the environment (*e.g.,* list of object ID's or their locations). In RobotIST, we further enable the robot to accurately and naturally reference relevant objects and locations.

While we do not currently have any explicit debugging functionality in RobotIST, its presentation of program semantics provides valuable debugging information to programmers at both programming and execution time. Programmers can form mental models more easily as they program with real-time, situated information provided by RobotIST.

Our current implementation of RobotIST has several limitations. RobotIST provides feedback only as soon as the scene is clear and the programmer's body does not occlude any part of the blocks or objects. However, the position of the camera and projector on the robot's head can result in situations where objects closer to the robot occlude blocks and other objects, hindering scene detection as well as projection. We mitigated this issue by mounting the projector and camera at a steep downward angle relative to the scene surface, but better solutions could be mounting the devices directly overhead or using rear-detection and projection. The AR tags on blocks are also visually distracting and could be replaced by rear-sensing or RFID technology in a non-prototype system.

Our evaluation only provides preliminary support for the value of RobotIST. More rigorous investigations are necessary to fully understand the uses and benefits of system features, as well as the kind of user groups and scenarios it best supports. For example, it would be interesting to compare RobotIST and a visual programming system in both single and multi-user contexts.

8 CONCLUSION

We introduce RobotIST: an interactive situated tangible robot programming interface. In addition to *situated input* from programmers (through selection blocks to reference task-relevant parts of the environment such as objects, locations, and regions), it allows *situated feedback* from the robot to the programmer (through projections onto the robot's workspace). We discuss the differences of RobotIST from traditional programming systems and illustrate its features by walking through an example programming task. Finally, we present an observational study with programmers, demonstrating ease of use and highlighting subjective qualitative differences from traditional robot programming.

9 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."

REFERENCES

[1] Baris Akgun, Maya Cakmak, Jae Wook Yoo, and Andrea Lockerd Thomaz. 2012. 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*. 391–398.

[2] S. Alexamdrova, M. Cakmak, K. Hsaio, and L. Takayama. 2014. Robot Programming by Demonstration with Interactive Action Visualizations. In *Robotics: Science and Systems (RSS)*.

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

[4] Rasmus S Andersen, Ole Madsen, Thomas B Moeslund, and Heni Ben Amor. 2016. Projecting robot intentions into human environments. In *Robot and Human Interactive Communication (RO-MAN), 2016 25th IEEE International Symposium on*. IEEE, 294–301.

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

[6] Emilia I Barakova, Jan CC Gillesen, Bibi EBM Huskens, and Tino Lourens. 2013. End-user programming architecture facilitates the uptake of robots in social therapies. *Robotics and Autonomous Systems* 61, 7 (2013), 704–713.

[7] Chris Beckmann and Anind Dey. 2003. Siteview: Tangibly programming active environments with predictive visualization. In *adjunct Proceedings of UbiComp*. 167–168.

[8] Antonio Bicchi and Vijay Kumar. 2000. Robotic grasping and contact: A review. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, Vol. 1. IEEE, 348–353.

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

[10] Rehj Cantrell, Paul Schermerhorn, and Matthias Scheutz. 2011. Learning actions from human-robot dialogues. In *2011 RO-MAN*. IEEE, 125–130.

[11] Ravi Teja Chadalavada, Henrik Andreasson, Robert Krug, and Achim J Lilienthal. 2015. That's on my mind! robot to human intention communication through on-board projection on shared floor space. In *Mobile Robots (ECMR), 2015 European Conference on*. IEEE, 1–6.

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

[13] L. Claassen, S. Aden, J. Gaa, J. Kotlarski, and T. Ortmaier. 2014. Intuitive Robot Control with a Projected Touch Interface. In *Social Robotics*. Springer, 95–104.

[14] Jennifer Cross, Christopher Bartley, Emily Hamner, and Illah Nourbakhsh. 2013. A visual robot-programming environment for multidisciplinary education. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE, 445–452.

[15] James P Diprose, Bruce A MacDonald, and John G Hosking. 2011. Ruru: A spatial and interactive visual programming language for novice robot programming. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*. IEEE, 25–32.

[16] Mohamed Elbanhawi and Milan Simic. 2014. Sampling-based robot motion planning: A review. *Ieee access* 2 (2014), 56–77.

[17] Daniel Gallardo, Carles Fernandes Julià, and Sergi Jorda. 2008. TurTan: A tangible programming language for creative exploration.. In *Tabletop*. Citeseer, 89–92.

[18] Guglielmo Gemignani, Emanuele Bastianelli, and Daniele Nardi. 2015. Teaching robots parametrized executable plans through spoken interaction. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems, 851–859.

[19] Dylan F Glas, Takayuki Kanda, and Hiroshi Ishiguro. 2016. Human-Robot Interaction Design Using Interaction Composer: Eight Years of Lessons Learned. In *The Eleventh ACM/IEEE International Conference on Human Robot Interation*. IEEE Press, 303–310.

[20] J Harvent, Benjamin Coudrin, Ludovic BrÃČÅąthes, Jean-JosÃČÂĺ Orteu, and Michel Devy. 2013. Multi-View Dense 3D Modelling of Untextured Objects From a Moving Projector-Cameras System. 24 (11 2013).

[21] Michael S Horn and Robert JK Jacob. 2007. Designing tangible programming languages for classroom use. In *Proceedings of the 1st international conference on Tangible and embedded interaction*. ACM, 159–162.

[22] Michael S Horn and Robert JK Jacob. 2007. Tangible programming in the classroom with tern. In *CHI'07 extended abstracts on Human factors in computing systems*. ACM, 1965–1970.

[23] Michael S Horn, Erin Treacy Solovey, R Jordan Crouser, and Robert JK Jacob. 2009. Comparing the use of tangible and graphical programming languages for informal science education. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 975–984.

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

[25] Itseez. 2015. Open Source Computer Vision Library. http://opencv.org/

[26] Majeed Kazemitabaar, Jason McPeak, Alexander Jiao, Liang He, Thomas Outing, and Jon E Froehlich. 2017. Makerwear: A tangible approach to interactive wearable creation for children. In *Proceedings of the 2017 chi conference on human factors in computing systems*. ACM, 133–145.

[27] Scott R Klemmer, Jack Li, James Lin, and James A Landay. 2004. Papier-Mache: toolkit support for tangible input. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 399–406.

[28] Jens Kober and Jan Peters. 2012. Reinforcement learning in robotics: A survey. In *Reinforcement Learning*. Springer, 579–610.

[29] A. Kurenkov, B. Akgun, and A. L. Thomaz. 2015. An evaluation of GUI and kinesthetic teaching methods for constrained-keyframe skills. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*. 3608–3613. https://doi.org/10.1109/IROS.2015.7353881

[30] Jean-Claude Latombe. 2012. *Robot motion planning*. Vol. 124. Springer Science & Business Media.

[31] D. Lazewatsky and W.D. Smart. 2012. Context-sensitive in-the-world interfaces for mobile manipulation robots. In *IEEE Intl. Symp. on Robot Human Communication (ROMAN)*. IEEE, 989–994.

[32] Tomas Lozano-Perez. 1983. Robot programming. *Proc. IEEE* 71, 7 (1983), 821–841.

[33] Michal Luria, Guy Hoffman, Benny Megidish, Oren Zuckerman, and Sung Park. 2016. Designing Vyo, a robotic Smart Home assistant: Bridging the gap between device and social agent. In *Robot and Human Interactive Communication (RO-MAN), 2016 25th IEEE International Symposium on*. IEEE, 1019–1025.

[34] Cynthia Matuszek, Evan Herbst, Luke Zettlemoyer, and Dieter Fox. 2013. Learning to parse natural language commands to a robot control system. In *Experimental Robotics*. Springer, 403–415.

[35] Timothy S McNerney. 2004. From turtles to Tangible Programming Bricks: explorations in physical language design. *Personal and Ubiquitous Computing* 8, 5 (2004), 326–337.

[36] Dipendra K Misra, Jaeyong Sung, Kevin Lee, and Ashutosh Saxena. 2016. Tell me dave: Context-sensitive grounding of natural language to manipulation instructions. *The International Journal of Robotics Research* 35, 1-3 (2016), 281–300.

[37] Shiwali Mohn and John Laird. 2014. Learning Goal-Oriented Hierarchical Tasks from Situated Interactive Instruction. In *Proceedings of the Twenty-eighth National Conference on Artificial Intelligence (AAAI)*.

[38] David Molyneaux and Hans Gellersen. 2009. Projected interfaces: enabling serendipitous interaction with smart tangible objects. In *Proceedings of the 3rd International Conference on Tangible and Embedded Interaction*. ACM, 385–392.

[39] Daniel Moreno and Gabriel Taubin. 2012. Simple, accurate, and robust projector-camera calibration. In *3D Imaging, Modeling, Processing, Visualization and Transmission (3DIMPVT), 2012 Second International Conference on*. IEEE, 464–471.

[40] Scott Niekum. 2013. Alvar. http://wiki.ros.org/ar_track_alvar

[41] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.

[42] Jun Rekimoto, Brygg Ullmer, and Haruo Oba. 2001. DataTiles: a modular platform for mixed physical and graphical interactions. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 269–276.

[43] Radu Bogdan Rusu and Steve Cousins. 2011. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China.

[44] Theodosios Sapounidis, Stavros Demetriadis, and Ioannis Stamelos. 2015. Evaluating children performance with graphical and tangible robot programming tools. *Personal and Ubiquitous Computing* 19, 1 (2015), 225–237.

[45] Allison Sauppé and Bilge Mutlu. 2014. Design patterns for exploring and prototyping human-robot interactions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1439–1448.

[46] Yasaman S Sefidgar, Prerna Agarwal, and Maya Cakmak. 2017. Situated Tangible Robot Programming. In *Proceedings of the 2017 ACM/IEEE International Conference on Human-Robot Interaction*. ACM, 473–482.

[47] Orit Shaer and Eva Hornecker. 2010. Tangible user interfaces: past, present, and future directions. *Foundations and Trends in Human-Computer Interaction* 3, 1–2 (2010), 1–137.

[48] Karun B Shimoga. 1996. Robot grasp synthesis algorithms: A survey. *The International Journal of Robotics Research* 15, 3 (1996), 230–266.

[49] Arnan Sipitakiat and Nusarin Nusen. 2012. Robo-Blocks: designing debugging abilities in a tangible programming system for early primary school children. In *Proceedings of the 11th International Conference on Interaction Design and Children*. ACM, 98–105.

[50] Maj Stenmark, Mathias Haage, and Elin Anna Topp. 2017. Simplified Programming of Re-usable Skills on a Safe Industrial Robot: Prototype and Evaluation. In *Proceedings of the 2017 ACM/IEEE International Conference on Human-Robot Interaction*. ACM, 463–472.

[51] Ioan A. Sucan and Sachin Chitta. 2011. MoveIt! http://moveit.ros.org

[52] Brygg Ullmer and Hiroshi Ishii. 2000. Emerging frameworks for tangible user interfaces. *IBM systems journal* 39, 3.4 (2000), 915–931.

[53] Robert van Herk, Janneke Verhaegh, and Willem FJ Fontijn. 2009. ESPranto SDK: an adaptive programming environment for tangible applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 849–858.

[54] Manuela Waldner, Jörg Hauber, Jürgen Zauner, Michael Haller, and Mark Billinghurst. 2006. Tangible tiles: design and evaluation of a tangible user interface in a collaborative tabletop setup. In *Proceedings of the 18th Australia conference on Computer-Human Interaction: Design: Activities, Artefacts and Environments*. ACM, 151–158.