Reactile: Programming Swarm User Interfaces through Direct Physical Manipulation

Ryo Suzuki[1], **Jun Kato**[2], **Mark D. Gross**[1], **Tom Yeh**[1]
[1]University of Colorado Boulder, [2]National Institute of Advanced Industrial Science and Technology
{ryo.suzuki, mdgross, tom.yeh}@colorado.edu, jun.kato@aist.go.jp

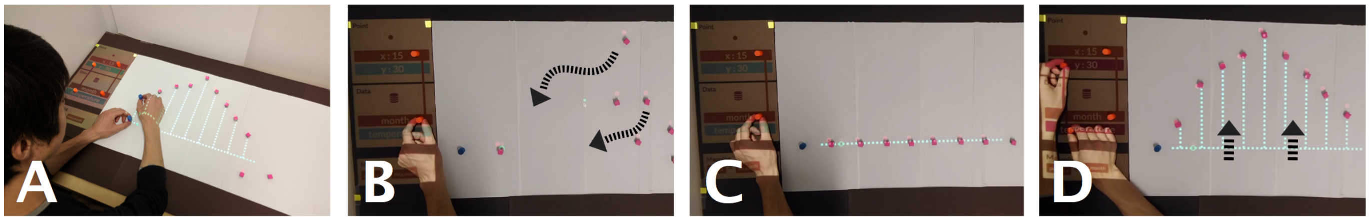

Figure 1. Reactile is a programming environment for swarm user interfaces. Reactile leverages physical demonstration for attribute abstraction and specification of data binding in Swarm UIs. Figures show an overview of data physicalization example (A), and a workflow to create a graph using Reactile (B-D). Reactile actuates a swarm of small magnets using PCB-based electromagnetic coils and displays program states using a projector.

ABSTRACT

We explore a new approach to programming swarm user interfaces (Swarm UI) by leveraging *direct physical manipulation*. Existing Swarm UI applications are written using a robot programming framework: users work on a computer screen and think in terms of low-level controls. In contrast, our approach allows programmers to work in physical space by directly manipulating objects and think in terms of high-level interface design. Inspired by current UI programming practices, we introduce a four-step workflow—create elements, abstract attributes, specify behaviors, and propagate changes—for Swarm UI programming. We propose a set of direct physical manipulation techniques to support each step in this workflow. To demonstrate these concepts, we developed Reactile, a Swarm UI programming environment that actuates a swarm of small magnets and displays spatial information of program states using a DLP projector. Two user studies—an in-class survey with 148 students and a lab interview with eight participants—confirm that our approach is intuitive and understandable for programming Swarm UIs.

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

Author Keywords

swarm user interfaces; tangible programming; programming by demonstration; direct manipulation

CHI 2018, April 21–26, 2018, Montreal, QC, Canada

ISBN 978-1-4503-5620-6/18/04. . . $15.00
DOI: https://doi.org/10.1145/3173574.3173773

INTRODUCTION

In recent years, Swarm User Interfaces (Swarm UI) [22] have emerged as a new paradigm of human-computer interaction. While the idea of coordinated miniature robots was originally proposed in the literature of swarm and micro-robotic systems [32, 36], HCI researchers have explored the use of these robots as a user interface [22]. In such interface, swarm robots can dynamically form shapes and morph to other shapes to display information in response to user inputs and surrounding environments [5, 21, 22]. Recent research has demonstrated the great potential of Swarm UI in many application domains, such as dynamic data physicalization [22], simulations and problem-solving [31, 30], wearable and tangible displays [5, 21], and accessibility assistants [48]. This emerging interaction paradigm opens up a new opportunity for practitioners to build novel applications for Human-Swarm Interaction [21].

However, this opportunity is currently limited to highly skilled programmers who are proficient in robot programming. For typical programmers inexperienced in robot programming who wish to build a Swarm UI application, it is unclear if the *robot programming* approach is the most appropriate for *UI programming*. To design interactive UI applications, programmers often must think in terms of higher-level design for user interaction, whereas robot programming tends to focus on low-level controls of sensors and actuators. Historically, a novel UI platform is adopted only after the advent of an effective programming tool that empowers a larger developer community, and even end-users, to create many applications for the platform; for example, HyperCard for interactive hypermedia, Phidgets for physical interfaces, and Interface Builder for GUI applications. We stipulate that current approaches to programming Swarm UI are too robot-centric to be effective for building rich and interactive applications. Then, what would be a better alternative?

As a first step toward answering this question, this paper explores a new approach to programming Swarm UI applications. To design an appropriate workflow for Swarm UI programming, we look into existing UI programming paradigm for inspiration. The common workflow of UI programming can be decomposed into four basic steps: create elements, abstract attributes, specify behaviors, and propagate changes. Based on these insights, we propose the following four-step workflow for Swarm UI programming: 1) creates shapes, 2) abstracts shape attributes as variables, 3) specifies data-bindings between dynamic attributes, and 4) the system changes shapes in response to user inputs (See Figure 2). With this workflow, a programmer can think in terms of high-level interface and interaction design to build interactive Swarm UI applications, compared to existing, low-level, robot programming approaches.

Moreover, given the physical nature of swarm user interfaces, we propose to support this programming workflow via *direct physical manipulation*. The motivation comes from an observation that the dominant programming environment is largely limited to coding on a two-dimensional computer screen [1]. This arrangement creates a large gulf of execution [26]; developers must continuously switch contexts between writing code on a screen and testing in physical space, which causes a significant cognitive distance between physical and virtual worlds [6]. To bridge this gulf, we present a set of direct manipulation techniques to perform each step of Swarm UI programming workflow. This approach allows a programmer to write and view a program in the same physical context, eliminating the aforementioned gulf.

To demonstrate these concepts, we developed Reactile, a programming environment for Swarm UI applications. Reactile actuates a swarm of small magnetic markers to move on a 2D canvas with electromagnetic force. We designed and fabricated a board of electromagnetic coil arrays (3,200 coils), which covers an 80 cm x 40 cm area. Reactile tracks the marker positions and detects interactions between a user and swarm markers using a standard RGB camera and computer vision techniques. The system displays spatial information using a DLP projector to allow a programmer to see program states in the same physical context. We show the proposed workflow and Reactile system can be effective to build various interactive applications such as data physicalization and explorable simulations.

To evaluate the proposed workflow and interaction design, we conducted two user studies; 1) a large-scale in-class survey with 148 students, and 2) an in-depth lab study with eight participants. Participants generally agreed that the proposed user interactions are intuitive (6.0), and the program is easy to understand (6.1), modify (5.0), and flexible for various applications (6.1), in response to 7-point Likert scale questions. The survey study also shows that the majority of students can understand the affordance of attribute abstraction (68-87%) and correctly predict the dynamic program behavior (33-46%). Based on qualitative feedback, we discuss three aspects (usability, interpretability, and flexibility) of our approach and draw design guidelines for Swarm UI programming.

In summary, we contribute:

- a design of the Swarm UI programming scheme, proposing a four-step workflow informed by existing UI programming.
- a set of interaction techniques that leverage direct physical manipulation to perform each step in this workflow.
- a demonstration of these concepts with Reactile, a working prototype consisting of a hardware device that actuates a swarm of magnetic markers and a software system that tracks user interaction and displays spatial information of program states.
- a mixed-method evaluation of our proposed approach, which shows potential advantages.

BACKGROUND AND MOTIVATION

Recent work in HCI envisions the world beyond tangible bits [16] where human interact with computers through dynamic physical objects. Under the vision of Programmable Matter [49], Ultimate Display [45], and Radical Atoms [15], research systems in actuated tangible interfaces [29, 33] and shape-changing interfaces [7, 35] have demonstrated interfaces that can dynamically change their physical shape in response to user interaction. In particular, a growing body of research investigates the potential of utilizing swarm of objects as user interfaces. Example applications include data physicalization [17, 22], wearable and ambient displays [5, 21], dynamic physical affordances and constraints [31], simulations and problem-solving [29], STEM education [27], and accessibility for people with visual impairments [48]. Despite such enthusiasm, it is still not easy to create swarm user interfaces, as discussed above. In this section, we review prior methods and current approaches to programming swarm user interfaces and explain the motivation behind our work.

Toolkits for Robot Programming

Swarm user interfaces are usually implemented as a swarm of robots that serve as interfaces between a host computer and the users. However, most robotics research has been about designing autonomous behaviors of a single robot, and existing robot programming systems [2] are typically designed for that purpose. Prior work has explored end-user robot programming by simplifying the programming experience. For example, the LEGO Mindstorms series[1] provide a visual programming environment in which children can control the behavior of a LEGO-based robot. HCI researchers have explored using a single mobile robot as user interfaces [11], using photos of robots in the code editor to aid comprehension of posture data [19], and using GUI to program cooking robots [43].

Only relatively recently has the community begin exploring the potential of multi-robot systems, forming the research field of swarm robotics [3]. While middleware and software libraries for robotics engineers help abstracting the hardware, network, and algorithm layers, most do not provide specialized features for swarm robots, and using these libraries to build a working swarm robotics systems typically requires understanding complex layers of abstraction.

[1]LEGO Mindstorms. https://www.lego.com/en-us/mindstorms

To allow the programmer to see the status of robots, several systems use the top-down view of the environment for showing debugging information and defining the absolute coordinates. Several middleware platforms [10] and toolkits [18] implement such features and enable programming of multiple mobile robots. In particular, Phybots [18] is designed for interaction designers without prior knowledge of robot programming. These toolkits were used in several HCI research projects that involve multiple mobile robots [41, 43]. More recent work includes Zooids [22], an open source hardware and software framework specialized for building Swarm UIs.

However, these environments typically involve writing code on a computer screen and then deploying the code to see results in the physical space. If the robot behaves unexpectedly, the programmer must mentally map the error of the physical robot back to the source code. This creates a large gulf of execution for programmers and a great cognitive distance between physical and virtual spaces [6]. In contrast, we explore an alternative approach where the program can be manipulated through embodied interaction in the physical world.

Tangible Programming Languages

The idea of programming in the physical space is not new. Tangible programming languages [46] leverage embodied physical interaction to construct a program. While conventional programming languages use textual or visual representations, tangible programming allows a programmer to manipulate the structure of a program using physical objects [9, 12, 14, 25, 40, 54]. Prior work has shown that tangible programming languages can be significantly more engaging than a visual programming language, particularly in educational contexts [13].

However, while these systems make program *structures* tangible, the program *states* are not visible and tangible in the physical space. For example, users can manipulate control flow of the program by constructing blocks that represent program structures such as `for loop` and `if else`, but users cannot see and manipulate the dynamic states of a program in the physical space. In contrast, our approach shows dynamic program states as spatial information. This approach enables the user to see how the program behaves and understand the program by manipulating tangible objects.

Programming by Demonstrations

To lower the barrier of programming, Programming by Demonstrations aims to enable end-users to program robot behaviors without writing code. For example, Topobo [34] lets a user demonstrate a movement, which is recorded and can later be played back to animate the robot.

However, since programming by demonstration generates the program by an inference, it is difficult for users to explicitly specify desired behaviors. In these cases, if the generated result is different from a programmer's expectations, not enough clue is provided to help the programmer understand the error and fix it. In addition, in these systems, the internal states of a program are hidden, but it is well known that the hidden states make it difficult to understand and fix unexpected behaviors [24, 47]. Thus, existing programming by demonstration systems are often used for simple repetitive operations of a single robot. Our approach, by contrast, allows users to explicitly specify the program behavior, similar to the traditional programming paradigm. While we also infer the attribute type from the user's demonstration in the attribute abstraction step, the inference result can be always seen and fixed by the user. We will show that this approach could be more effective to program interactive behaviors of multiple robots.

Direct Physical Manipulation Interfaces

Direct manipulation techniques in the physical space have been studied since 1990s [51, 55]. Such interfaces leverage embodied physical interaction to create static elements (e.g., 3D models in Mockup Builder [4]) or interact with pre-programmed behaviors (e.g., optical simulation in HOBIT [8] or I/O Bulb [50]). However, there have been fewer investigations in direct manipulation of *authoring dynamic behaviors*. Historically, authoring dynamic behaviors has been done by coding as it requires abstraction [53], while recent research started exploring the direct manipulation of dynamic behavior in GUI applications [20, 38, 39, 52]. The core question in our paper is how we can expand this to interfaces in the physical world. While this paper specifically focuses on programming of Swarm UI, we expect the direct manipulation programming will become more important in spatial and physical interfaces, such as tangible, augmented reality, and shape-changing interfaces.

DESIGNING SWARM UI PROGRAMMING

We propose *Swarm UI Programming*, a new approach to building Swarm UI applications that focus on high-level UI design. The workflow of Swarm UI programming is inspired by the existing UI programming paradigm. We first review the common workflow of UI programming and decompose it into four basic elements that represent high-level steps. Then we discuss how to apply this workflow to Swarm UI programming.

Four Elements of Existing UI Programming

As we see in well-known design patterns for interactive UI applications such as reactive programming paradigm, the Model-View-Controller, and the observer pattern, they share a common workflow consisting of four basic elements: **1) create elements**, **2) abstract attributes**, **3) specify behaviors**, and **4) propagate changes**.

Consider, for example, making an interactive web application using HTML and JavaScript: **1) Create elements:** A user first creates basic elements of interface with HTML DOM such as `div`, `button`, and `text`. **2) Abstract attributes:** Then, the user abstracts these attributes as variables, such as the background color or font-size. These attributes can be changed dynamically by updating variable values. **3) Specify behaviors:** The user specifies behaviors to describe how abstracted attributes will change with data-bindings. For example, one can specify that the button's `background-color` will change in response to the `text` attribute of the input element. **4) Propagate changes:** Based on the user-defined data-bindings, the system automatically propagates the change by detecting user input or data changes. For example, detecting an input value such as "brown", automatically changes the `background-color` attribute of the button element.

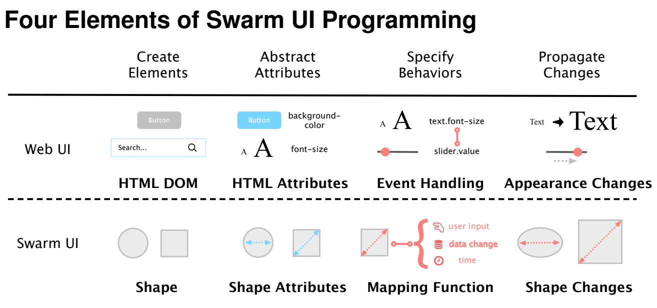

Figure 2. Four basic elements of Web UI and Swarm UI programming.

Now, we draw a parallel between UI programming and Swarm UI programming by introducing the following four-step workflow:

1. **Create elements:** In Swarm UI programming, we propose that shapes are basic UI elements, as the swarm can represent information and communicate with a user through changing shapes. A shape in Swarm UI comprises of a swarm of small tangible objects. In this paper, we denote each unit as a "marker" which can be either a robot or an actuated tangible object.
2. **Abstract attributes:** As in a web application, a shape is a static element. To dynamically change a shape, the user must introduce attributes such as width, height, scale, position, angle, radius, and curvature. For example, the user can define an angle attribute of an arrow or a radius of a circle, which can be changed through programming.
3. **Specify behaviors:** To make an interactive Swarm UI application, the user can specify how a shape's attributes change when an event occurs. The event can be user input, changes in the external data source, or the progress of time.
4. **Propagate changes:** Once the user specifies the behavior, the system can watch for changes to the control unit. For example, if someone moves the control marker, thereby increasing `x`, the system automatically updates the arrow shape's `angle` attribute.

SWARM UI PROGRAMMING VIA DIRECT MANIPULATION

Given the physical nature of Swarm UIs, we propose to support this programming workflow via *direct physical manipulation*. Rather than coding in a separate IDE on a computer screen, a programmer should be able to program a Swarm UI by physically manipulating the swarm. To achieve this goal, we propose the following direct manipulation workflow:

Step 1. Create Elements by Drawing and Construction

The first step to programming a swarm UI application is to make shapes. A programmer can make shapes in two ways; 1) moving and arranging individual swarm markers into the desired shape, 2) drawing the desired shape with a freehand stroke. In either case, the hand-made shape need not be perfect. The system should guess which basic shape (e.g., line, circle, triangle, rectangle) the programmer is trying to make and beautify it when possible. Then, a swarm of markers moves to corresponding positions to form a shape. The user can also manually modify the shape by placing or removing individual markers.

Once a shape is made, the system constructs a class for that shape and adds it to the program space. This allows the programmer to later abstract the attributes of the shape and clone a shape as an instance. The current states of the program, such as the set of shape classes and associated variables, is visualized in a side panel as spatial information. Each shape class is represented by a similarly shaped icon in the control panel. To instantiate an object of the class, the programmer first places a marker at a class window and moves it to the workspace, then markers in the surroundings form the shape.

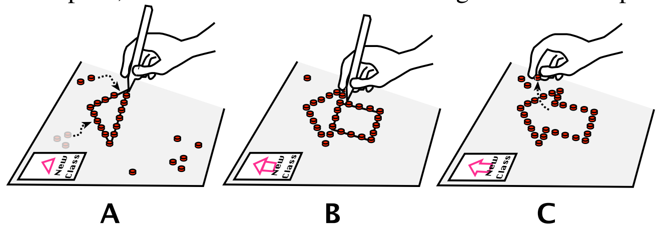

Figure 3. Create elements by drawing and construction. A programmer can create elements by arranging markers or drawing the desired shape.

Figure 3 shows a programmer making an arrow shape in the programming environment. The following pseudocode illustrates how a program evolves over the three steps. First, the programmer draws a triangle (Figure 3A), then the system adds a triangle in the program space.

```
<triangle x="0" y="10"/>
```

By drawing a rectangle (Figure 3B), the system adds another shape.

```
<triangle x="0" y="10"/>
<rectangle x="5" y="15"/>
```

The programmer can remove a horizontal line by directly picking up markers and putting them aside (Figure 3C). Once a shape is created, the environment adds the current shape as a class the programmer can name.

```
<arrow x="0" y="10"/>
```

Step 2. Abstract Attributes through Demonstrations

One important aspect of programming is the ability to generalize a specific case using a higher-level abstraction. Suppose a programmer has constructed an arrow shape and wants to change its orientation. To do so, the programmer can abstract an attribute of a defined shape by introducing a variable. For example, the following pseudocode illustrates how this operation can be done in a common programming language. To change the orientation of the arrow, the programmer can simply set `a` to a different value.

```
var a = 30
<arrow angle={ a } x="0" y="10"/>
```

To support a programmer to abstract variables through direct manipulation, we take inspiration from constraint-based drawing [44]. Our system uses tangible constraint markers. To define a variable to represent a certain shape attribute, a programmer puts constraint markers on an existing shape. The system environment infers which shape attribute the programmer is trying to demonstrate. For example, Figure 4 illustrates how a programmer uses constraint markers and demonstrations to define a variable to abstract angle attribute, as in the pseudocode above.

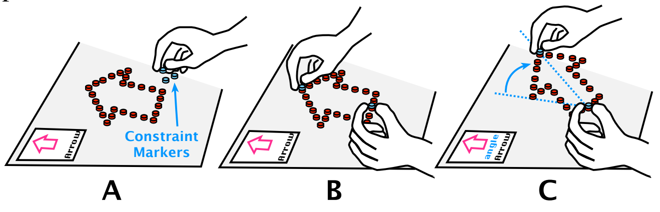

Figure 4. Abstracting attributes by demonstration. An arrow's "angle" attribute is abstracted as a variable using two constraint markers.

Different demonstrations can define different variables such as position, width, height, scale, and orientation. For example, Figure 5 shows other examples of abstracting A) a rectangle's scale attribute, B) a marker's *x* position attribute, and C) an angle of an arc shape.

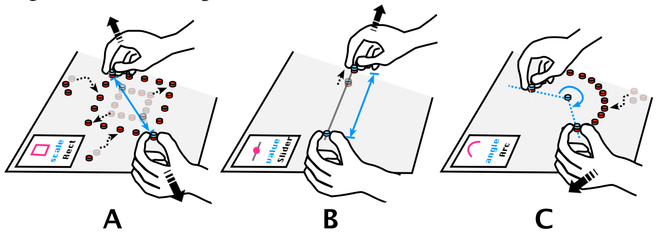

Figure 5. Using constraint markers to specify different shape attributes: diagonal length (A), position (B), and angle (C).

If our system sees that the programmer exhibits a behavior matching one of the heuristics above, it creates a variable for the attribute suggested by the heuristic and adds it to the program space. Each variable is visualized as a window containing the shape's icon and the attribute's name.

Step 3. Specify Behaviors by Connecting Attributes

After creating shape classes and abstracting some of their attributes as variables, the next step is to specify their behaviors. To specify how certain attributes may change based on the user input, the programmer can create a mapping function to relate each variable. In the left panel where the program space is visualized, variables already defined show up as individual windows. To specify a data binding, the programmer selects two variables, then the system adds a data binding to the program space. It also provides visual feedback by showing a line between the two variables.

Suppose a programmer wants to specify the following behavior: when a point is dragged to the right, the angle of the arrow rotates clockwise. The pseudocode implements this behavior.

```
1 var a = 30;
2 var b = 10;
3 bind(a, b)
4 <arrow angle={ a } x="0" y="10"/>
5 <point x={ b }/>
```

This implementation involves choosing a marker in the swarm to be the control (line 5) and abstracting the marker's *x* position attribute as a new variable *b* (line 2). Then, a binding is defined between *a* and *b*, using a pseudo-function `bind()` (line 3). Based on the current value (e.g., $a = 30$ and $b = 10$), the system automatically creates an appropriate mapping function (e.g., a = b + 20). If the user wants to define the different data binding, the user can select different expression suggested by the system (e.g., a = b * 3) or modify the expression (e.g, a = b * 360 / 100). When the program is running, the system can watch for changes in *b* and propagate the changes to *a*, achieving the desired behavior—the arrow rotates as the marker is moved right.

Step 4. Propagate Changes through Physical Interaction

Once a programmer specifies the behavior by connecting attributes, the system automatically detects the change in the value of the associated variable and propagates the changes.

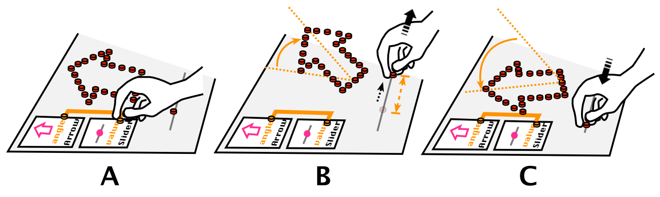

Figure 6. Specifying behaviors by creating bindings between variables. Once a programmer connects two attributes by placing selection markers, then the system automatically binds them and propagates the change.

Figure 6 shows how a programmer uses direct manipulation to bind two variables to specify the dynamic behavior described above. In A), he puts two markers on each variable's window, which is equivalent to `bind(a, b)`. In B), he drags the control marker to the right; the arrow rotates accordingly. In C), he drags the control marker to the left; the arrow rotates in the opposite direction.

REACTILE: SYSTEM AND IMPLEMENTATION

To demonstrate these concepts, we developed Reactile, a working prototype for our proposed Swarm UI programming environment. Reactile's hardware actuates a swarm of passive magnetic markers, which are basic elements of a Swarm UI. The use of passive markers does not fit in the strict definition of Swarm UI [22], but the proposed workflow and programming paradigm are general enough to be demonstrated on systems with either passive markers or self-propelled elements. To enable direct physical manipulation, Reactile tracks a set of distinctively colored markers using a mounted standard RGB camera and computer vision techniques. The dynamic program states are displayed spatially in the same physical context using a DLP projector. Reactile's software and hardware are open-sourced and available on GitHub [2]. In the following, we describe the hardware and software design and implementation of Reactile system.

Hardware

Electromagnetic Actuation

In Reactile, a user interface consists of a swarm of passive magnetic markers which move on a 2D workspace driven by electromagnetic forces. Reactile uses a grid of electromagnetic

[2] https://github.com/ryosuzuki/reactile

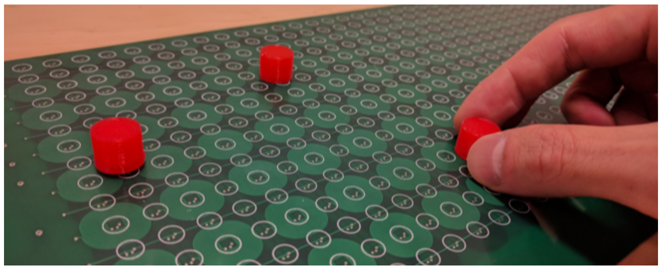

Figure 7. Reactile uses a field of electro-magnetic coils fabricated with a standard PCB manufacturing. Each board has 16 x 40 coils and the final prototype uses five boards to cover 80 cm x 40 cm area with 3,200 coils. This board can actuate passive magnetic markers shown as red objects with 10 mm diameter.

coils to actuate these magnetic markers. Running current through the circuit coils generates a local magnetic field so that each coil can attract a single magnet located within its area. Each coil is aligned with a certain offset in both horizontal and vertical direction with an effective area overlap, which allows the coil to attract the magnet located in the adjacent coil. We design electromagnetic coil arrays to be fabricated with a standard printed circuit board (PCB) manufacturing (Figure 7). This reduces the cost and fabrication complexity, making it easy for the actuation area to scale up.

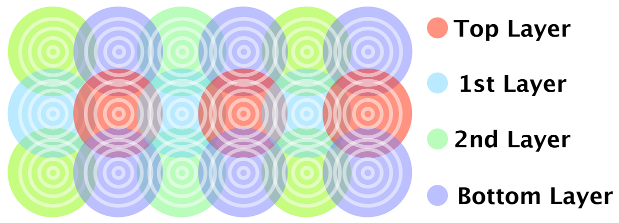

Figure 8. A simplified schematic of our coil design of a 4-layer PCB. Each layer has a set of coils aligned with a certain offset in both horizontal and vertical directions. Each coil is 15 mm diameter and has 2.5 mm overlap between nearby coils.

Figure 8 shows the simplified schematic of the coil design. Our PCB design is a 4-layer board, and each layer contains a set of coils, each of which has an identical circular shape with a 15 mm diameter and a 2.5 mm overlap between nearby coils. Each coil has 15 turns with 0.203 mm (8 mils) spacing between lines, and the distance between centers of two coils is approximately 10 mm, which makes a 10 mm grid for attractive points. Due to the maximum size of the PCB facility we used, a single board has 40 x 16 coils which approximately covers a 40 cm x 16 cm area. We design the actuation board to be scalable, so that we can extend the effective area without any design changes. The final prototype covers an 80 cm x 40 cm area with 80 x 40 coils by aligning five identical boards horizontally. The fabrication of each board costs approximately $80 USD, including manufacturing of PCB and electronic components.

Passive Magnetic Marker

Each marker consists of an N48 neodymium disc magnet and a 3D printed cap. As shown in Figure 9, the magnet is attracted with a local magnetic field generated by nearby coils. The basic requirement for a magnet is that its size is large enough to overlap with nearby coils (Figure 9). Thus, the minimum size of magnets depends on the size of the coil and offsets. In our prototype, the minimum size of the magnet is 6 mm diameter, and we used magnets with 10 mm diameter.

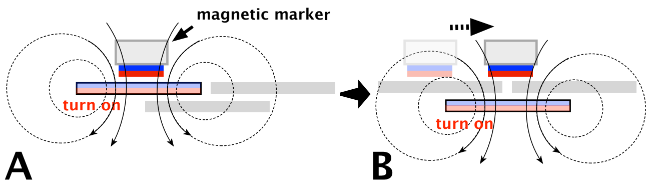

Figure 9. An actuation mechanism of Reactile. Running current through the coils generates a local magnetic field to attract magnetic markers located within its area.

All electromagnetic coils generate the same direction of a magnetic field to attract magnetic markers, similar to [42, 48]. Thus, each magnet is directed in the same direction (e.g., the north pole faces up and south pole faces down). As all the magnets face the same direction, they are prevented from attracting and connecting with others. The magnetic markers repel each other if the distance between two markers becomes closer than a certain distance. The minimum distance between magnets depends on the diameter and strength of the magnets, and in our prototype, this minimum distance is approximately 30 mm.

Marker Control

To produce a local magnetic field, we switch on the current for each coil. As our board has 80 x 40 coils, it requires 3,200 switches to control each coil. To reduce the required switches, we adopt a multiplexing technique for efficient current control. Similar to LED displays, this approach only requires 80 + 40 switches to control 3,200 coils. On the other hand, this allows us to control only one row at a time; By switching the current, it can move multiple markers with a relatively high refresh rate. In our settings, the system switches the current in 100 ms for each marker. For example, if there are 10 markers in different row, it takes approximately 1 second (=100 ms x 10) to move them independently (See Figure 10)..

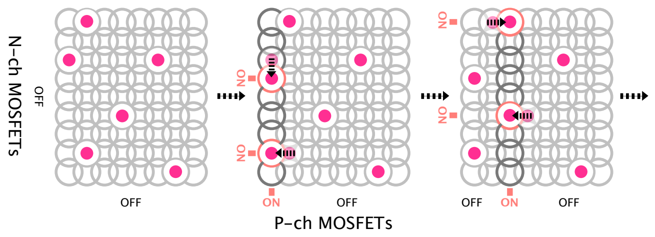

Figure 10. A control mechanism with push and pull pair of P-ch and N-ch MOSFETs. While only one column (or row) can be turned on at each time, switching with fast refresh rate (10Hz in our settings) allows to move multiple magnets nearly simultaneously.

To switch the current on/off for each row and column, we use the push and pull pair of P-channel and N-channel power MOSFET transistors. To run the current through a coil, the gate voltage of P-ch and N-ch MOSFETs should be set as LOW and HIGH respectively. For example, to turn on the coil at column 10 and row 8, we set P10 as LOW and the rest of columns (P-ch) as HIGH, and N8 as HIGH and the rest of the rows (N-ch) LOW.

The gate voltage of each MOSFET is controlled by daisy-chained shift registers. The five boards share the same data, latch, and clock pins of the shift register, so that only six pins are required to control the 80 x 40 coils in the entire board. The shift register is controlled by an Arduino microcontroller, which communicates with a host computer through I2C communication.

We used 74HC595 for 8-bit shift registers, MSS2P3 for diodes, and AO3401 and AO3400 for P-ch and N-ch MOSFETs respectively. All electric components are surface-mount parts and are attached to the bottom layer; therefore, the top layer is flat to allow the markers to move freely. The source voltage for P-ch MOSFET comes from a 5.5V external power supply, and the average and peak current for each coil were 0.4A and 1.2A respectively.

Software

Marker Tracking

To track the markers' positions, we use a standard RGB camera and computer vision techniques. The software first extracts an image of the workspace by detecting white color and finding the contours in the image. Then, we approximate contours with polygonal curves to obtain the positions of the four edges of the rectangle workspace. After extracting four edges of the rectangle workspace, the system warps the input image with a geometric transformation to eliminate distortion and fits the image to the rectangular workspace.

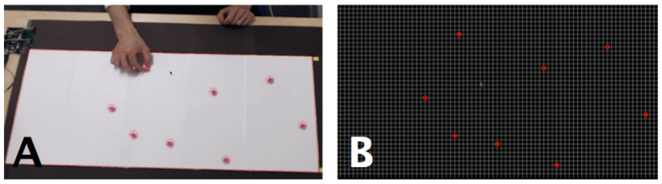

Figure 11. We use computer vision to detect a rectangle workspace and positions of the markers (A). The system uses detected position information within 80 x 40 grid for path planning and controlling marker movements (B).

To make it easy to track swarm markers, we color them in high contrast colors. To track markers in an image, we first convert the image's color scale to hue, saturation, and value (HSV) and detect a specific color with a lower and upper threshold for each value. Then, the input image is converted to a binary-colored image where the detected color is white and the rest is black. The detected colored marker position is then calculated as a relative position within the workspace by dividing its horizontal position by 80 and vertical position by 40. We use this technique to detect the standard red markers as well as the other special markers including constraint markers (blue) and selection markers (orange). Figure 11 A illustrates the input image captured by the camera and detected workspace highlighted with a red-lined rectangle. Figure 11 B shows the position of each marker projected onto an 80 x 40 grid based on the warped workspace. We used OpenCV for computer vision and Logitech C920 for the RGB camera, which is mounted 100 cm above the table.

Shape Detection

Reactile allows a user to create elements by drawing and construction. To enable this, the system lets the user draw with a laser pointer and tracks the point using the same computer vision technique. While the user is drawing a freeform stroke, the system displays it using a DLP projector. Once the user finishes drawing, the system beautifies the drawing by inferring the possible shape type. We use the $1 unistroke recognizer [56] for this purpose. This technique allows our system to recognize a range of shapes with a few sample strokes. When the system recognizes the shape, it converts the freehand stroke into a more refined shape and displays the shape as feedback using a projector.

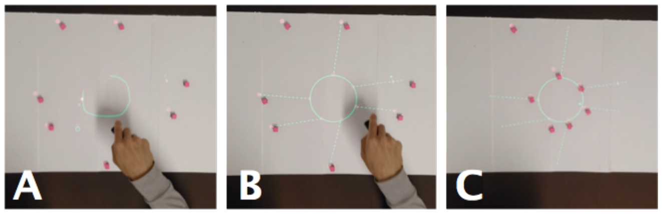

Figure 12. Reactile allows a user to draw a basic shape with a laser pointer. The system can converts freehand stroke into a beautified shape, and then determines the target positions.

Target Assignment and Path Planning

After detecting a shape, our system needs to move swarm markers to form the shape. Each swarm marker has a target position to move to. To determine these target positions, we use the Douglas-Peucker algorithm to sample n points along the path where n is the number of markers. Also, we enforce a minimum radial distance between markers so they do not interfere magnetically.

Once a set of target positions are known, we use the Munkres assignment algorithm to compute an optimal assignment of these positions to individual markers. The optimization objective is to minimize the total distance traveled by all markers. This assignment is dynamically updated at each step to more efficiently allocate target positions.

Attributes Abstraction

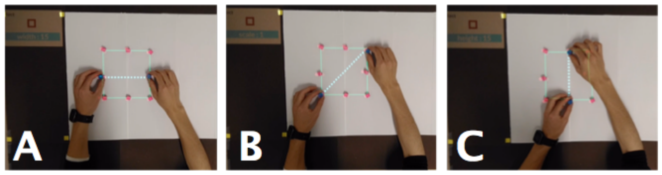

Figure 13. Reactile lets a user to abstract attributes as variables through demonstration with blue constraint markers. When the system detects the demonstration, it updates the left panel to show a list of variables and current states.

Each time a programmer creates a shape, the system adds a shape class and displays it in the left panel. The programmer can then use the constraint markers to specify an attribute to abstract out as a variable. Constraint markers have blue color and system separately detects the position of constraint markers. Once two constraint markers are placed on top of normal markers, the system starts inferring the most probable shape subject to the constraints. The system computes the relative position of two markers and selects candidate constraints from the available variable spaces. For example, Figure 13 illustrates how a user defines variables by placing two constraint markers on top of a rectangle shape. The different demonstrations result in the different constraints: placing two markers horizontally infers the width, while placing them vertically infers the height as a variable.

Some constraints may have ambiguities. For example, if a user puts one constraint marker at the origin and another at (0, 10), the user can mean either the angle or the height of a shape. In such cases, the system allows the user to move the constraint markers for disambiguation. The system tracks the delta of the positions when the user moves the markers as a demonstration. For example, if the user moves one constraint marker from $[10,10]$ to $[11,9]$, then the delta of the marker is $[1,-1]$. The system calculates the cosine between the initial position (e.g., $[10,10]$) and the delta (e.g., $[1,-1]$), then estimates an appropriate parameter. For example, if the user rotates around the center (Figure 14B), then the system infers as an "angle" attribute (e.g., $cos = 0$), while the user moves one marker in the radial direction, then the system infers as a "length" attribute (e.g., $cos = 1$).

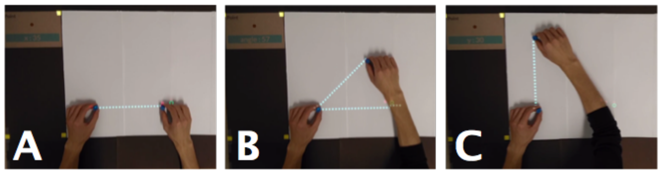

Figure 14. Defining x, angle, and y variable by changing the location of a single point marker.

Data-binding and Propagation

Once a user finishes defining variables, the system allows the user to specify a data-binding by demonstration. To do so, the user can simply place selection markers on top of a certain variable in the left panel. When the user selects two variables by placing selection markers, then the system automatically creates a mapping function to specify the data-binding between two variables.

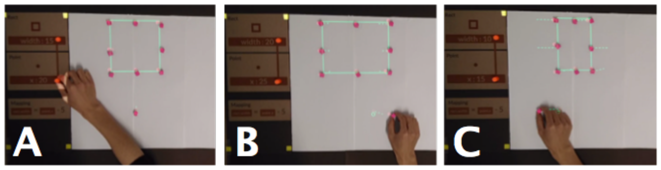

Figure 15. The user can create a mapping function with orange selection markers (e.g., rect.width = point.x - 5). Once the mapping function is created, the system can automatically propagate changes whenever the variable value is changed.

For example, Figure 15 A illustrates that a user connects a rectangle's width attribute and a single marker's *x* position attribute. Based on their current values (e.g., rect.width = 15 and point.x = 20), the system automatically creates *rect.width = point.x - 5* as a mapping function.

After defining data-bindings, the user can immediately interact with the Swarm UI she just programmed by physically manipulating the markers. For example, Figure 15 B and C show that a user can change the width of the rectangle by moving a marker to the right or the left. These variables can be not only shape attributes, but also time-dependent variables or external data sources. For example, Figure 16 shows that the user can make a constant animation which rotates a rectangle by connecting an angle variable to a time variable, whose value constantly changes over time.

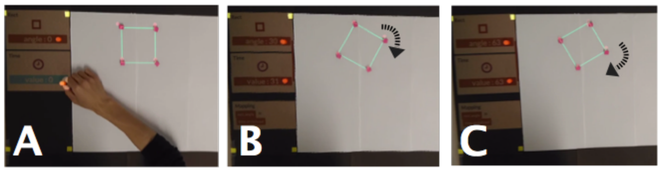

Figure 16. The user can also create a mapping function between attributes and time-dependent variable for continuous animation.

POSSIBLE APPLICATION SCENARIOS

In this section, we present example applications to demonstrate the capabilities of Swarm UI Programming. We specifically describe two different categories to demonstrate how our workflow can support the programming of swarm user interfaces: data physicalization and explorable simulations.

Data Physicalization

Data physicalization is a promising research area where Swarm UIs can be useful [22], particularly to help blind people understand and explore data [48]. While existing research work have studied how users interact with data, there is relatively less work investigating how users *author* their own dynamic data physicalizations. Using Reactile, users can "physicalize" data by connecting data values to representative shape attributes, such as the size of a circle or the length of a line. This connection can be specified using the direct manipulation techniques described above.

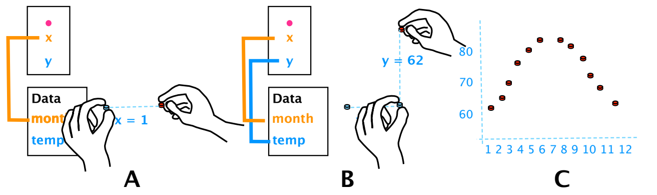

Figure 17. An application example in data physicalization. A user defines x and y attributes, then binds them respectively to month and temperature data (A-B). The system propagates the value to each object (C).

For example, in Figure 1 and 17, a user wants to create a graph that represents the temperature of a city throughout the year. She first defines the x and y variables using a reference point. When she connects the variable x to the month data (Figure 17 A), the system notices that the month data has twelve integer values and automatically instantiates eleven more objects from the same class and propagates to the next value with an one-to-many mapping. In this way, the user now has twelve single points that are horizontally distributed with different *x* values (Figure 17 B). Next, she connects a variable y to temperature data, and the system propagates the y value to each object (Figure 17 C). In this way, the Swarm UI displays a 2D plot whose *x*-axis represents the month and *y*-axis represents the temperature of that month.

Explorable Simulations

Tangible representation serves as a powerful medium to engage people with physical objects. Prior work has shown that two-handed tangible interaction helps users to explore simulations and problem-solving [31]. With Reactile, users can not only interact with such explorable simulations as a consumer, but also create them as an author.

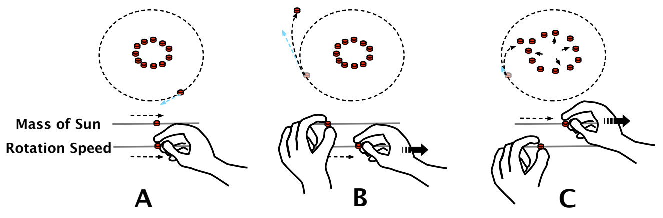

Figure 18. An application example in explorable simulation. A teacher in a physics class can create tangible explanation on the fly, so that students can interact with markers to see how the mass of the Sun affects the orbit of the Earth.

For example, a teacher in a physics class wants to demonstrate how the mass of the Sun affects the orbit of the Earth. First, she makes a circle shape to represent the Sun and abstracts the circle's "radius" as a variable. Next, the teacher uses a marker to represent the Earth; she makes it revolve around the sun by connecting it to a time-dependent variable. She creates two slider objects: one controls the radius of the Sun and the other controls the velocity of the Earth's orbit. To demonstrate to her class, she holds the two slider objects and moves them sideways simultaneously to animate the shape of the Sun and the movement of the Earth (Figure 18 A). She shows that if the difference between the two values is too big, the Earth falls into the Sun or flies into outer space (Figure 18 B-C). By showing this, the teacher interactively demonstrates how gravity and the velocity of an orbiting object affect each other.

Ambient Display and Animation

Swarm UIs are also promising for ubiquitous interfaces which show information as an ambient display [21]. Creating interactive animation of such displays could be also an interesting application. For example, a user could make a timer or a progress bar to indicate its status with Reactile. To make a radial progress bar, a user first creates an arc shape and abstract its angle as a variable, so that she can bind the angle variable to the real-time data. Then, when the progress data increases, the arc becomes a circle shape to indicate its progress.

USER STUDY

We conducted a survey study and a lab study to understand programmers' experiences as well as the appropriateness of our approach, focusing on the following research questions:

RQ1: Is the representation and behavior of a program easy to understand, predict, and modify?
RQ2: Do programmers find the proposed interaction techniques intuitive?

where, we used the term "intuitive" as "the behavior of the interface is easy to expect".

Participants

For the survey study, we recruited subjects from a large upper-level computer science course. Students were expected to all have prior programming experience. A total of 148 students participated in our survey. Because the survey was anonymous, we do not have demographics.

For the lab study, we recruited eight participants (7 male, 1 female), ages 19-31 (average: 24.3) years old from our institution. Having prior programming experiences was an inclusion criterion. All participants were from engineering majors (4 computer science, 2 mechanical engineering, and 2 electronic engineering). Each session approximately took 45 minutes.

Method

For the survey study, we designed a set of quiz questions to test to what extent participants were able to understand the programming techniques we proposed for the four-step workflow. Before seeing the questions, participants watched a short demonstration video. Each question contained one or more photos to illustrate a direct manipulation technique and asked participants to predict the outcome by selecting from four choices. Twelve questions were included in the survey.

For the lab study, the goal was to provide participants with an opportunity to physically interact with our programming environment. Each participant was explained the purpose of the study, shown a demonstration of the system, and given a simple programming task to perform. After the task was finished, the participant received a short survey containing eight questions. Five questions asked them if the proposed interaction design was intuitive. The other three questions examined the participants' opinions on whether the program is easy to understand and modify, and if the proposed interface seems flexible for many different applications. Participants answered on a 7-point Likert scale where 1 is strongly disagree and 7 is strongly agree.

Result

Our survey study yielded mixed results. Participants performed relatively well on the two quiz questions about the prediction of Step 2 Abstract attribute with a correctness rate of 67%(93/139) and 87%(128/146). Among those who got incorrect answers, the most common type of confusion was between the *height* attribute and the *y* attribute of a rectangle. Note that because participants could skip questions, the *n* was slightly different for each question.

On the three questions concerning Step 4, however, only 43%(58/135), 46%(59/129), and 33%(46/138) of the participants answered correctly. The two questions that most challenged the participants concerned Step 3. Only 22%(30/138) and 35%(44/126) of the participants answered correctly. The accuracy rate was close to random. The results were below our expectation. One reason could be that the survey instrument did not provide the fully *tangible* interaction experience; participants only saw video and photo illustrations.

	Average Score (SD)
Overall user interactions	6.0 (0.7)
Step 1: Create elements	5.8 (0.6)
Step 2: Abstract attributes	5.5 (1.3)
Step 3: Specify behaviors	5.4 (1.2)
Step 4: Propagate changes	5.0 (1.3)
Easy to understand	6.1 (0.9)
Easy to modify	5.0 (1.6)
Flexible for different applications	6.1 (1.3)

Table 1. Summary of 7-point Likert-scale responses.

Our lab study, on the other hand, showed more promising results. Overall, participants had a positive view of their experiences with our proposed Swarm UI programming environment. The table above shows a summary of 7-point Likert scale response to each question. Overall, participants generally agreed that the proposed interaction techniques were intuitive ($6.0, \sigma = 0.7$). Also, they thought that the program was both easy to understand ($6.0, \sigma = 0.7$) and to modify ($5.0, \sigma = 1.6$) and that it can be flexible for different applications ($6.1, \sigma = 1.3$). The next section discusses these results to gain insights for an appropriate design for Swarm UI programming.

DISCUSSION

Usability: The participants in our lab study generally agreed that the proposed interactions are natural and intuitive, by stating that using two-hand interactions makes programming fun (P7) and engaging (P8). Particularly, three participants, who have prior experiences in robot programming, identified the benefits of programming in the physical space. P1 mentioned that how our approach reduces the barrier of programming such swarm user interfaces by comparing it to his past experience in programming swarm robots; *"while programming these robots, I usually need to compile it, deploy, and see how it works every single time."* (P1)

Moreover, participants are excited by the new opportunity for users to create Swarm UI applications without programming knowledge. For example, P2 saw a great potential for classroom use such as in math education, stating that *"One application I had in mind was education. For example, teachers in middle schools can teach geometry such as sine or cosine by interactively demonstrating with these markers. Students can also interact with it to understand math."* (P2)

Interpretability: Similar to survey study participants, some lab study participants found it difficult to predict program behaviors. This difficulty might be due to the task design. In general, participants may have difficulty with correctly understanding and predicting a program without actually constructing it, particularly in an unfamiliar system or programming language. Although participants generally agreed that Step 3 and Step 4 are easy to understand, they also commented that these steps can be improved. For example, P3 suggested that the system should visualize data-bindings directly on the swarm markers, as opposed to in the left panel only. Indicating the active attributes with highlighted auxiliary lines can help improve the interpretability of variable mappings and specified behaviors.

Flexibility: While contextual information helps, a separate program space contributes to the flexibility and generalizability of a program. For example, P4 stated that the displayed information in the left panel was helpful for him to understand the structure within a standardized view. Thus, one important design implication is the need to make the appropriate connections between the abstract (e.g., variables and class) and the concrete (e.g., shapes) spaces in order to enable better mental models between these representations, while still maintaining the flexibility and generalizability of the program.

Scalability: In the user study, P7 wondered if the program could scale to more than a few shapes and attributes. One way to handle a large number of shapes and parameters is to provide contextual information which only shows the related parameters or binding information in the left panel. As we see similar experimental programming interface in GUI, such as Apparatus [39] and Sketchpad14 [37], we expect this approach can also handle scalability with a similar design.

LIMITATIONS AND FUTURE WORK

Although the proposed interactions were generally appreciated, the technical limitations of our hardware prototype sometimes limited the usability and capability of our approach. For example, the refresh rate in the current implementation depends on the number of objects and it becomes non-negligible as the number of markers increases. While this can be addressed through different implementations (e.g., using a transistor for each coil instead of multiplexing), this prevented users from receiving immediate feedback, making it difficult to predict a program's behavior (P1, P7). Another hardware limitation is the information resolution. Although an individual marker is small in size, the minimum distance between two markers (30 mm) prevents them from forming a high-density shape. We also tested smaller and weaker magnets with N35 and 8 mm diameter, which requires only 12-15 mm distance. However, there is a trade-off with the weight of the marker. For example, in the above case, if we attach a 3D printed cap (1.1 g), it can become difficult to travel. However, we expect the minimum distance can be decreased using a galvanized steel case [23].

While these limitations are not about our proposed interaction techniques, having appropriate hardware can be crucial for better programming experiences. The hardware design option explored by this work is based on a swarm of simple, passive magnets actuated by a board. Another option for future work is to explore other types of robots such as wheel-based robots. One concern is that these robots may introduce cost increases and additional technical challenges in coordinating them as a swarm. However, we are seeing a growing body of research projects [22] and commercial efforts to address this concern [28], which can eventually lead to significant cost reduction and make Swarm UIs practical in the near future. This work contributes to that possible future by proposing a programming paradigm people can use to create a range of novel Swarm UI applications.

In terms of the capability of our programming paradigm, an interesting discussion is how to extend our approach to additional dimensions. For example, P3 mentioned that additional attributes such as color, texture, and z height can be helpful for creating more expressive applications such as data visualizations, accessibility, and shape-displays. However, it is not trivial to extend our approach to these modalities. For future work, we will explore how our approach can be generalized to these other application domains.

ACKNOWLEDGMENTS

We thank Kevin Kuwata and Zhixian Jin for work on Reactile hardware. This research was supported by the NSF CAREER award IIS 1453771 and the Nakajima Foundation.

REFERENCES

1. Gregory D. Abowd. 2012. What Next, Ubicomp?: Celebrating an Intellectual Disappearing Act. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing (UbiComp '12)*. ACM, New York, NY, USA, 31–40. DOI:http://dx.doi.org/10.1145/2370216.2370222

2. Geoffrey Biggs and Bruce MacDonald. 2003. A survey of robot programming systems. In *Proceedings of the Australasian conference on robotics and automation*. 1–3.

3. Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. 2013. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence* 7, 1 (01 Mar 2013), 1–41. DOI: http://dx.doi.org/10.1007/s11721-012-0075-2

4. Bruno R. De Araùjo, Géry Casiez, and Joaquim A. Jorge. 2012. Mockup Builder: Direct 3D Modeling on and Above the Surface in a Continuous Interaction Space. In *Proceedings of Graphics Interface 2012 (GI '12)*. Canadian Information Processing Society, Toronto, Ont., Canada, Canada, 173–180. http://dl.acm.org/citation.cfm?id=2305276.2305305

5. Artem Dementyev, Hsin-Liu (Cindy) Kao, Inrak Choi, Deborah Ajilo, Maggie Xu, Joseph A. Paradiso, Chris Schmandt, and Sean Follmer. 2016. Rovables: Miniature On-Body Robots As Mobile Wearables. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 111–120. DOI: http://dx.doi.org/10.1145/2984511.2984531

6. Johan Fabry and Miguel Campusano. 2014. Live robot programming. In *Ibero-American Conference on Artificial Intelligence*. Springer, 445–456.

7. Sean Follmer, Daniel Leithinger, Alex Olwal, Akimitsu Hogge, and Hiroshi Ishii. 2013. inFORM: Dynamic Physical Affordances and Constraints Through Shape and Object Actuation. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. ACM, New York, NY, USA, 417–426. DOI: http://dx.doi.org/10.1145/2501988.2502032

8. David Furió, Stéphanie Fleck, Bruno Bousquet, Jean-Paul Guillet, Lionel Canioni, and Martin Hachet. 2017. HOBIT: Hybrid Optical Bench for Innovative Teaching. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 949–959. DOI: http://dx.doi.org/10.1145/3025453.3025789

9. Daniel Gallardo, Carles F Julia, and Sergi Jorda. 2008. TurTan: A tangible programming language for creative exploration. In *Horizontal Interactive Human Computer Systems, 2008. TABLETOP 2008. 3rd IEEE International Workshop on*. IEEE, 89–92.

10. Brian Gerkey, Richard T Vaughan, and Andrew Howard. 2003. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th international conference on advanced robotics*, Vol. 1. 317–323.

11. Darren Guinness, Daniel Szafir, and Shaun K. Kane. 2017. GUI Robots: Using Off-the-Shelf Robots As Tangible Input and Output Devices for Unmodified GUI Applications. In *Proceedings of the 2017 Conference on Designing Interactive Systems (DIS '17)*. ACM, New York, NY, USA, 767–778. DOI: http://dx.doi.org/10.1145/3064663.3064706

12. Michael S. Horn and Robert J. K. Jacob. 2007. Designing Tangible Programming Languages for Classroom Use. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction (TEI '07)*. ACM, New York, NY, USA, 159–162. DOI: http://dx.doi.org/10.1145/1226969.1227003

13. Michael S. Horn, Erin Treacy Solovey, R. Jordan Crouser, and Robert J.K. 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 (CHI '09)*. ACM, New York, NY, USA, 975–984. DOI:http://dx.doi.org/10.1145/1518701.1518851

14. Felix Hu, Ariel Zekelman, Michael Horn, and Frances Judd. 2015. Strawbies: Explorations in Tangible Programming. In *Proceedings of the 14th International Conference on Interaction Design and Children (IDC '15)*. ACM, New York, NY, USA, 410–413. DOI: http://dx.doi.org/10.1145/2771839.2771866

15. Hiroshi Ishii, Dávid Lakatos, Leonardo Bonanni, and Jean-Baptiste Labrune. 2012. Radical Atoms: Beyond Tangible Bits, Toward Transformable Materials. *interactions* 19, 1 (Jan. 2012), 38–51. DOI: http://dx.doi.org/10.1145/2065327.2065337

16. Hiroshi Ishii and Brygg Ullmer. 1997. Tangible Bits: Towards Seamless Interfaces Between People, Bits and Atoms. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '97)*. ACM, New York, NY, USA, 234–241. DOI: http://dx.doi.org/10.1145/258549.258715

17. Yvonne Jansen, Pierre Dragicevic, Petra Isenberg, Jason Alexander, Abhijit Karnik, Johan Kildal, Sriram Subramanian, and Kasper Hornbæk. 2015. Opportunities and Challenges for Data Physicalization. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 3227–3236. DOI: http://dx.doi.org/10.1145/2702123.2702180

18. Jun Kato, Daisuke Sakamoto, and Takeo Igarashi. 2012. Phybots: A Toolkit for Making Robotic Things. In *Proceedings of the Designing Interactive Systems Conference (DIS '12)*. ACM, New York, NY, USA, 248–257. DOI: http://dx.doi.org/10.1145/2317956.2317996

19. Jun Kato, Daisuke Sakamoto, and Takeo Igarashi. 2013. Picode: Inline Photos Representing Posture Data in Source Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 3097–3100. DOI: http://dx.doi.org/10.1145/2470654.2466422

20. Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, Shengdong Zhao, and George Fitzmaurice. 2014. Draco: Bringing Life to Illustrations with Kinetic Textures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 351–360. DOI: http://dx.doi.org/10.1145/2556288.2556987

21. Lawrence H. Kim and Sean Follmer. 2017. UbiSwarm: Ubiquitous Robotic Interfaces and Investigation of Abstract Motion As a Display. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 1, 3, Article 66 (Sept. 2017), 20 pages. DOI:http://dx.doi.org/10.1145/3130931

22. Mathieu Le Goc, Lawrence H. Kim, Ali Parsaei, Jean-Daniel Fekete, Pierre Dragicevic, and Sean Follmer. 2016. Zooids: Building Blocks for Swarm User Interfaces. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 97–109. DOI: http://dx.doi.org/10.1145/2984511.2984547

23. Rong-Hao Liang, Han-Chih Kuo, Liwei Chan, De-Nian Yang, and Bing-Yu Chen. 2014. GaussStones: Shielded Magnetic Tangibles for Multi-token Interactions on Portable Displays. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 365–372. DOI: http://dx.doi.org/10.1145/2642918.2647384

24. Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing Misconceptions About Code with Always-on Programming Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2481–2490. DOI: http://dx.doi.org/10.1145/2556288.2557409

25. Timothy S. McNerney. 2004. From Turtles to Tangible Programming Bricks: Explorations in Physical Language Design. *Personal and Ubiquitous Computing* 8, 5 (Sept. 2004), 326–337. DOI: http://dx.doi.org/10.1007/s00779-004-0295-6

26. Donald A. Norman and Stephen W. Draper. 1986. User Centered System Design; New Perspectives on Human-Computer Interaction. (1986).

27. Ayberk Özgür, Séverin Lemaignan, Wafa Johal, Maria Beltran, Manon Briod, Léa Pereyre, Francesco Mondada, and Pierre Dillenbourg. 2017. Cellulo: Versatile Handheld Robots for Education. In *Proceedings of the 2017 ACM/IEEE International Conference on Human-Robot Interaction (HRI '17)*. ACM, New York, NY, USA, 119–127. DOI: http://dx.doi.org/10.1145/2909824.3020247

28. OzoBot. 2017. http://ozobot.com. (2017). (Accessed on 09/09/2017).

29. Gian Pangaro, Dan Maynes-Aminzade, and Hiroshi Ishii. 2002. The Actuated Workbench: Computer-controlled Actuation in Tabletop Tangible Interfaces. In *Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology (UIST '02)*. ACM, New York, NY, USA, 181–190. DOI: http://dx.doi.org/10.1145/571985.572011

30. J Patten. 2014. Thumbles-robotic tabletop user interface platform. *TED. com* (2014).

31. James Patten and Hiroshi Ishii. 2007. Mechanical Constraints As Computational Constraints in Tabletop Tangible Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*. ACM, New York, NY, USA, 809–818. DOI: http://dx.doi.org/10.1145/1240624.1240746

32. Ron Pelrine, Annjoe Wong-Foy, Brian McCoy, Dennis Holeman, Rich Mahoney, Greg Myers, Jim Herson, and Tom Low. 2012. Diamagnetically levitated robots: An approach to massively parallel robotic systems with unusual motion properties. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. IEEE, 739–744.

33. Ivan Poupyrev, Tatsushi Nashida, and Makoto Okabe. 2007. Actuation and Tangible User Interfaces: The Vaucanson Duck, Robots, and Shape Displays. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction (TEI '07)*. ACM, New York, NY, USA, 205–212. DOI: http://dx.doi.org/10.1145/1226969.1227012

34. Hayes Solos Raffle, Amanda J. Parkes, and Hiroshi Ishii. 2004. Topobo: A Constructive Assembly System with Kinetic Memory. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*. ACM, New York, NY, USA, 647–654. DOI: http://dx.doi.org/10.1145/985692.985774

35. Majken K. Rasmussen, Esben W. Pedersen, Marianne G. Petersen, and Kasper Hornbæk. 2012. Shape-changing Interfaces: A Review of the Design Space and Open Research Questions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*. ACM, New York, NY, USA, 735–744. DOI: http://dx.doi.org/10.1145/2207676.2207781

36. Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal. 2014. Programmable self-assembly in a thousand-robot swarm. *Science* 345, 6198 (2014), 795–799.

37. Hesam Samimi, Alessandro Warth, Mahdi Eslamimehr, and Alan Borning. 2015. Constraints As a Design Pattern. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) (Onward! 2015)*. ACM, New York, NY, USA, 28–43. DOI: http://dx.doi.org/10.1145/2814228.2814244

38. Arvind Satyanarayan and Jeffrey Heer. 2014. Lyra: An interactive visualization design environment. In *Computer Graphics Forum*, Vol. 33. Wiley Online Library, 351–360.

39. Toby Schachman. 2015. Apparatus. *URL: http://aprt.us/* (2015).

40. 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 (HRI '17)*. ACM, New York, NY, USA, 473–482. DOI: http://dx.doi.org/10.1145/2909824.3020240

41. Thomas Seifried, Michael Haller, Stacey D. Scott, Florian Perteneder, Christian Rendl, Daisuke Sakamoto, and Masahiko Inami. 2009. CRISTAL: A Collaborative Home Media and Device Controller Based on a Multi-touch Display. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces (ITS '09)*. ACM, New York, NY, USA, 33–40. DOI:http://dx.doi.org/10.1145/1731903.1731911

42. Evan Strasnick, Jackie Yang, Kesler Tanner, Alex Olwal, and Sean Follmer. 2017. shiftIO: Reconfigurable Tactile Elements for Dynamic Affordances and Mobile Interaction. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 5075–5086. DOI: http://dx.doi.org/10.1145/3025453.3025988

43. Yuta Sugiura, Daisuke Sakamoto, Anusha Withana, Masahiko Inami, and Takeo Igarashi. 2010. Cooking with Robots: Designing a Household System Working in Open Environments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 2427–2430. DOI: http://dx.doi.org/10.1145/1753326.1753693

44. Ivan E Sutherland. 1964. Sketchpad a man-machine graphical communication system. *Transactions of the Society for Computer Simulation* 2, 5 (1964), R–3.

45. Ivan E Sutherland. 1965. The Ultimate Display. *Multimedia: From Wagner to virtual reality* (1965).

46. Hideyuki Suzuki and Hiroshi Kato. 1995. Interaction-level Support for Collaborative Learning: AlgoBlock&Mdash;an Open Programming Language. In *The First International Conference on Computer Support for Collaborative Learning (CSCL '95)*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 349–355. DOI: http://dx.doi.org/10.3115/222020.222828

47. Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Bjoern Hartmann. 2017a. TraceDiff: Debugging unexpected code behavior using trace divergence. In *Visual Languages and Human-Centric Computing (VL/HCC), 2017 IEEE Symposium on*. IEEE.

48. Ryo Suzuki, Abigale Stangl, Mark D. Gross, and Tom Yeh. 2017b. FluxMarker: Enhancing Tactile Graphics with Dynamic Tactile Markers. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '17)*. ACM, New York, NY, USA, 190–199. DOI: http://dx.doi.org/10.1145/3132525.3132548

49. Tommaso Toffoli and Norman Margolus. 1991. Programmable Matter: Concepts and Realization. *Physica D: Nonlinear Phenomena* 47, 1-2 (Jan. 1991), 263–272. DOI: http://dx.doi.org/10.1016/0167-2789(91)90296-L

50. John Underkoffler and Hiroshi Ishii. 1998. Illuminating Light: An Optical Design Tool with a Luminous-tangible Interface. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '98)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 542–549. DOI: http://dx.doi.org/10.1145/274644.274717

51. John Underkoffler and Hiroshi Ishii. 1999. Urp: A Luminous-tangible Workbench for Urban Planning and Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*. ACM, New York, NY, USA, 386–393. DOI: http://dx.doi.org/10.1145/302979.303114

52. Bret Victor. 2013. Drawing Dynamic Visualizations. *URL: http://worrydream.com/DrawingDynamicVisualizationsTalk* (2013).

53. Bret Victor. 2014. Humane Representation of Thought: A Trail Map for the 21st Century. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 699–699. DOI: http://dx.doi.org/10.1145/2642918.2642920

54. Danli Wang, Cheng Zhang, and Hongan Wang. 2011. T-Maze: A Tangible Programming Tool for Children. In *Proceedings of the 10th International Conference on Interaction Design and Children (IDC '11)*. ACM, New York, NY, USA, 127–135. DOI: http://dx.doi.org/10.1145/1999030.1999045

55. Pierre Wellner. 1993. Interacting with Paper on the DigitalDesk. *Commun. ACM* 36, 7 (July 1993), 87–96. DOI:http://dx.doi.org/10.1145/159544.159630

56. Jacob O. Wobbrock, Andrew D. Wilson, and Yang Li. 2007. Gestures Without Libraries, Toolkits or Training: A $1 Recognizer for User Interface Prototypes. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST '07)*. ACM, New York, NY, USA, 159–168. DOI: http://dx.doi.org/10.1145/1294211.1294238