IBM Research Report

CardKit: A Card-Based Programming Framework for Drones

Saad Ismail, Justin G. Manweiler, Justin D. Weisz

IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598 USA

CardKit: A Card-Based Programming Framework for Drones

Saad Ismail, Justin G. Manweiler, Justin D. Weisz

IBM Research

Yorktown Heights, NY 10598

saad@us.ibm.com, jmanweiler@us.ibm.com, jweisz@us.ibm.com

Abstract

Drones are being used in many industries for a variety of applications, including inspecting bridges, surveying farm land, and delivering cargo. Automating these kinds of scenarios requires more than following a sequence of GPS waypoints; they require integrating on-device hardware with real-time analysis to provide feedback and control to the drone. Currently, implementing these kinds of advanced scenarios is a complex task, requiring skilled software engineers programming with drone APIs. We envision an alternate model to enable drone operators to orchestrate advanced behaviors using a card-based approach. We describe the design of our card-based programming model, position it relative to other visual programming metaphors, share results from our paper prototype user study, and discuss our learnings from its implementation. Results suggest that a wide range of scenarios can be implemented with moderate mental effort and learning, balanced by intuitiveness and engagement.

1   Introduction

Unmanned Autonomous Vehicles (UAVs) or "drones" have begun to radically transform many industries. Drones are already being used for applications such as delivering packages, recording feature films, inspecting cell towers and wind turbines, mapping construction sites, and even shepherding and lifeguarding. Drones are especially advantageous for tasks that are dangerous (e.g., bridge inspection, forest fire mapping) or expensive (e.g., aerial surveying).

Because of the huge potential of drones to perform work tasks with increased safety and efficiency, we envision a future of diverse and complex drone applications. Some of these applications are already starting to emerge, such as using drones for
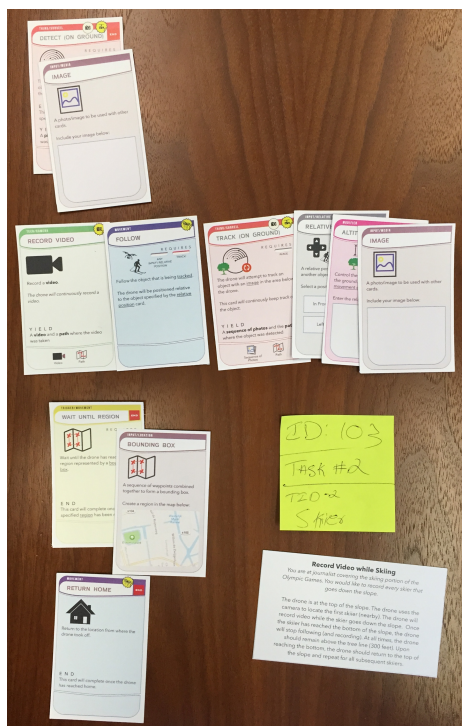
Figure 1: Card program instructs a drone to follow and record video of a skiier. This program is constructed as a sequence of four *hands*, sequenced vertically.

pizza, medicine, and package delivery, recording feature films, inspecting cell towers and wind turbines, and even shepherding and lifeguarding. These applications are dynamic and heterogeneous, and are typically "implemented" by having one or more human operators control the drone to perform the task. Mobile apps can ease the burden of controlling a drone's flight by pre-programming it to fly on specified paths, or to perform specific camera maneuvers (e.g. follow me, cable cam), but these apps tend to only work well for a single use case and do not adapt to alternative use cases.

At the heart of these applications is a worker who wishes to use a drone as a tool to automate a portion of their job. Many drones have programming APIs, and many of the applications mentioned above could be implemented with these APIs. However, drone APIs are typically complex and require knowledge of coding to use, making them inaccessible to those workers who do not have the requisite skills. For example, a farmer may wish to use their drone to find lost cattle, but building that application requires combining knowledge of a drone API and computer vision algorithms with the skill to implement it across a drone and a mobile device.

Therefore, we perceive both a need and an opportunity to design a simple and intuitive "programming" paradigm for drones that can automate a diverse set of drone behaviors.

As human-computer interaction researchers, our instincts led us to explore simplified and pedagogical programming environments, such as those aimed toward young or inexperienced programmers [1, 10, 3]. However, these languages typically expose low-level programming constructs such as conditional logic, looping and recursion, and data structures. Although providing Turing completeness, such sources of complexity *may not actually* be required to implement a number of complex drone scenarios. Instead, our aim is to create a programming paradigm in which *ease of use* trumps *computability* when such a tradeoff must be made.

Many of the programmable interactions with robots and other IoT devices concern its actions and behaviors. For example, a device may gather data (sense), process data (think), or manipulate its hardware state (act/communicate). These activities are high-level abstractions that mask the complexity of the underlying task (e.g. performing a computer vision task on an image, or driving a servo motor to achieve a desired movement). Our aim is to keep the *programming* of these activities at a suitably high level. Block-based metaphors [1] can mask some of these complexities, but block-based languages still expose lower-level programming constructs such as variables, loops, and functions. We questioned whether these constructs were truly necessary for writing useful, non-trivial programs. In our ideation, we settled on the notion of a "card" to encapsulate a high-level behavior for sensing, thinking, acting, or communicating. Our notion of a card is spiritually similar to other "cards" developed for programming robots [20], but it eliminates some lower level programming constructs (variables, loops, functions), and changes the use of others (conditional logic), used in block-based programming languages.

We readily admit that we did not attempt to devise a general-purpose programming language with strong theoretical underpinnings; rather, our aim was to create something easy to use for a wide, but possibly incomplete, range of scenarios. We examined a broad spectrum of programming abstractions, both pedagogical and for practitioners (see related work), and we concentrated on those that we believed would map well to mobile devices – convenient for "in the field" operations. Our insight was that "cards" was a good metaphor because many card games enable a player to implement complex strategies within constraining rule systems, while remaining usable on a mobile device.

Our vision for a card-based programming model converged across many iterations of design, paper prototyping, and testing. Our use of the physical medium enabled rapid design exploration, evaluation, and iteration. In this paper we only report on the final design upon which we converged; discussion of the numerous design options we attempted and rejected is beyond our scope.

Our paper makes the following contributions:

1. We created a programming model that uses a card "game" metaphor to orchestrate the behaviors of an autonomous physical system. The salient characteristic of this model is the use of *cards* in *hands* and *decks* to specify concurrent and sequential behaviors. Additionally, physical hardware is abstracted through a system of *token* cards, and data flows through a system of *yields* and *inputs*.

2. We designed a set of drone-specific cards on paper to rapidly evaluate our system. Our user testing demonstrated that our system was easy to learn, engaging, and highly expressive with acceptable levels of mental effort.

3. We created a prototype implementation of our system, called CardKit. It includes an implementation of drone-specific behaviors using the DJI API [5]. CardKit is implemented in the Swift programming language and has been released open source at **http://github.com/CardKit**.

2 Related Work

2.1 Programing Paradigms

Several important programming paradigms, including many visual paradigms [12, 18] inspired our approach as we developed our card-based metaphor.

2.1.1 Block-based Programming

Block-based programming environments, such as as Scratch [11, 10], RAPTOR [3], and Tickle [8] visualize traditional programming constructs as blocks of assorted shapes, helping users learn how to structure programs. Blocks often contain visual "connectors" which place constraints on which blocks can be used together, making it easier to understand the relationship between blocks, such as inputs or sequenced operations.

For programming drones, block-based metaphors might be expressive enough to encompass a broad range of drone activities. However, although these tools do simplify the act of programming by helping users focus on what code means versus the syntax used to express it [1], fundamental sources of complexity remain: users need to learn basic programming principles – variables, conditionals, loops, data structures, etc. – and work in some form of lower-level "code." Further, it may be challenging to program using these tools in the demanding physical environments

in which drones may operate, such as in the outdoors using a smartphone or tablet device, where manipulating code-level expressions might be challenging.

2.1.2 Dataflow Programming

Dataflow programming [15] is a style of visual programming in which functions and operations exist as nodes in a graph, with edges representing data dependencies. Node-RED [6], Cantana [19], and Fabrik [9] are examples of dataflow programming. It is a popular form of programming for IoT devices in which a stream of data is received, transformed, and then acted upon.

Our card-based programming model shares some similarities with the dataflow model. We allow data to flow through a card program by having data *yielded* by cards accessible as *input* to subsequent cards. However, the flow of data is strictly unidirectional; card programs are cycle-free DAGs, not arbitrary graphs. In addition, card programs contain a single entry point – the first hand – rather than an arbitrary number of data-producing sources.

2.1.3 Trigger-Action Programming

Trigger-action programming uses the construction of "if some event happens, then perform some action." In an evaluation of this metaphor, Ur et al. [17] found that "this paradigm can express many of the smart-home behaviors that immediately came to our participants' minds." However, this model is highly *reactive* and depends on triggers firing when an external event occurs in order to perform some action. For programming drones or other IoT devices that may need to *actively* initiate behaviors, a pure trigger-action model may not be adequate.

We have adopted the notion of a trigger in our model by defining cards that explicitly act as triggers, waiting for an external event to occur. For example, the SetATimer card triggers [1] after the specified amount of time has elapsed, and the WaitUntilLocation card triggers after a certain geolocation is reached.

2.1.4 Card Programming

Magic Cards [20] uses cards to implicitly define tasks a robot must perform in a physical environment. Cards are placed in different rooms to specify tasks to perform in those rooms. In a home, the robot moves around the house searching for tasks and executes those tasks when they are discovered. Cards are grouped into different categories (action, modifier, order, special purpose, and object) and some

---

[1] By *triggers*, we mean that the card's end condition becomes satisfied. We discuss end conditions in the next section.

cards can modify the behavior of others. This solution works well for robots that are situated in a small physical space, as the use of the cards is highly contextual – e.g. "mop the floor" is placed in the kitchen, and "vacuum the carpet" is placed in the bedroom. For UAVs and other robots operating in different environments – e.g. in the air, underwater, or other places inaccessible to people – the placement of physical cards in physical spaces to specify robot behaviors breaks down.

2.2   Existing Drone APIs

Popular drone manufacturers such as DJI and Parrot provide low-level APIs for controlling drones [5, 13]. These APIs are flexible, powerful, and well-supported with frameworks for programming languages including Swift, Python, and Java. Although these frameworks possess the broadest possible expressibility in terms of what kinds of tasks can be accomplished with a drone, writing such solutions may require substantial software engineering expertise and effort. In comparison, our card programming model reduces the possible space of expressible programs in favor of simplifying those programs that can be expressed. The degree to which complex programs are simplified is discussed later as part of the results of our user study.

2.3   Collectible Card Games

Our approach borrows several concepts from collectible card games (CCGs) such as *Hearthstone* and *Magic: The Gathering*. CCGs have a set of cards and rules that define how and when cards are played. Cards typically have standard designs depicting important information, with an icon or artwork representing the card, a description, a "cost" for playing the card, constraints on playing the card (e.g. prerequisites), and the outcome or effects of playing the card. Our model borrows these specific concepts from CCGs:

- We use the terminology of *cards* to represent single behaviors or actions, *hands* to represent a set of concurrent activities, and *decks* to represent a sequence of operations.

- Our use of tokens to limit concurrent behaviors is modeled after the notion of "mana" or "gold" in CCGs as a way of managing finite resources. For example, a camera token represents the fact that a drone possesses a single camera. Thus, within a hand, only one action card may be played having a dependency on the camera token.

- Our cards contain artwork for easy identification and are color-coded based on their function.

6

- Some cards are "played" on top of other cards, such as when they serve as inputs or modifiers.

3   Card Programming Mechanics

We present the details of our card-based programming model in the context of drones, as the operations of a drone are relatively simple and easily understood: a drone can *fly* to a particular location, it can *capture* data using a camera or other sensors, it can perform *computations*, such as computing flight plans or analyzing images, and it can *communicate* via feedback in a mobile app. These qualities comprise the sense, think, act, and communicate paradigm for robots [14]. We note that although our presentation is drone-centric, we discuss later how the model can be applied to other kinds of robots and IoT devices that are able to sense, think, act, and/or communicate.

Our ultimate aspiration is to create a digitized means to construct drone programs, and later in this paper we discuss our implementation effort. As a first step, we began our design process using paper cards in order to rapidly prototype and test the system, and iterate where needed.

3.1   General Structure: Cards, Hands, and Decks

A *deck* (program) consists of several sequential *hands* (steps). In each step, a "player" (programmer) lays out a set of cards, or *hand*, interpreted collectively as a unit.

A hand represents a set of concurrent drone behaviors; thus there is generally no defined order amongst cards in the same hand. However, there is one exception – some cards in the hand might *modify* or provide *input* to other cards. In that case, the card is played by stacking it on top of the card it modifies. Figure 2 depicts how computation occurs sequentially across hands and in parallel within a hand.

Figure 3(a) showcases a simple example drone program. It instructs a drone to fly to a location and take a photo. The first of two hands contains a FlyTo card and a Location card. The Location card acts as an input to the FlyTo card, specifying the location to which the drone should fly. The drone will first fly to that location and then proceed to the next hand. In the second hand, the TakeAPhoto card instructs the drone to take a photo. The drone will take a photo and proceed to the next hand. In this scenario, the sequence of hands ends here and the drone program has completed. In order to provide a degree of safety, we implicitly assume the presence of a Land card at the end of each deck to ensure the drone does not hover indefinitely after completing its mission.
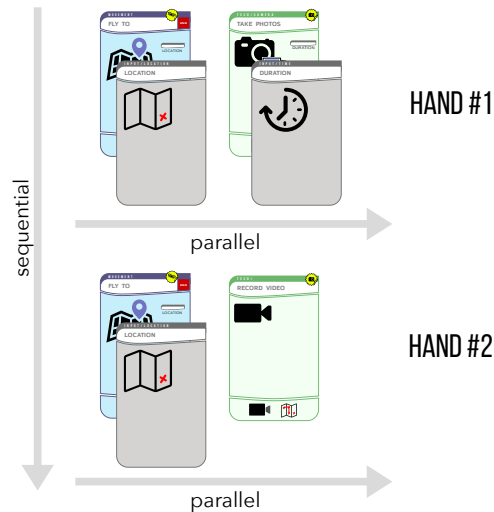
Figure 2: In each hand, multiple cards represent concurrent drone activities (horizontally). Stacked cards are of type *Input*. Multiple (vertical) hands represent a sequence of independent activities.

All cards in a hand must complete (or "end") before proceeding. The *end condition* is indicated with an "End" icon located at the top right corner of the card. If an end symbol does not exist, the card will execute indefinitely. In the example in Figure 3(a), since FlyTo contains an end condition, execution proceeds to the next hand only when the FlyTo card has satisfied its end condition by reaching its specified destination.

In Figure 3(b), the first half of the hand contains a FlyTo card, which has an end condition, and the second half contains a TakePhotos card, which does not. A Duration card specifies a period for taking photos. Thus, the drone will fly to the specified location while taking photos. If an indefinite duration is specified, the drone will continue hovering and taking photos "forever," until the battery is consumed and the drone lands itself.

Generally, card programs contain one hand per program step. However, we support a form of conditional branching logic via *branch* cards. These cards enable the drone to branch to a different hand depending on which *Action* ends first. Figure 4 shows an artificial example in which a drone either lands when a desired humidity level is measured, or returns home after a certain amount of time has elapsed.
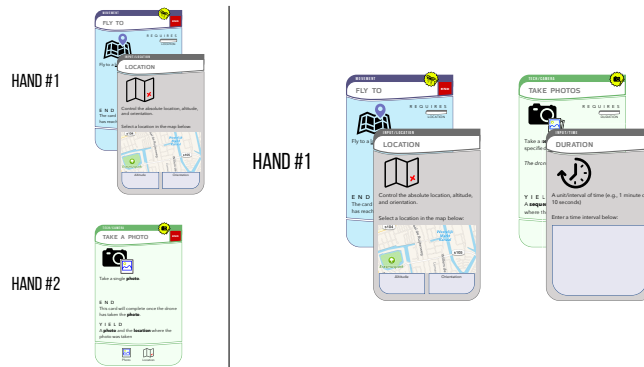
Figure 3: Sample card programs: (a) Fly to a location and take one photo. (b) Take photos while flying.
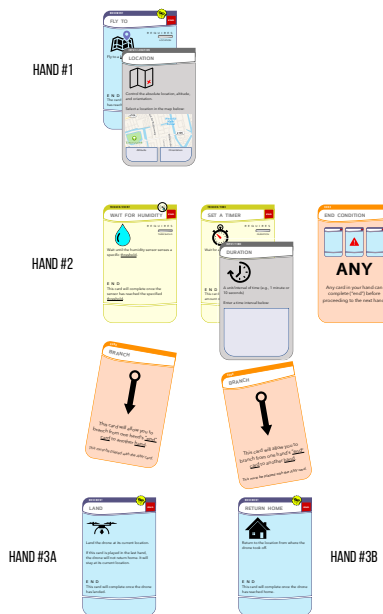

Figure 4: Card program for detecting and reacting to humidity levels. Note how branching directs control flow based on which *Action* card ends first.
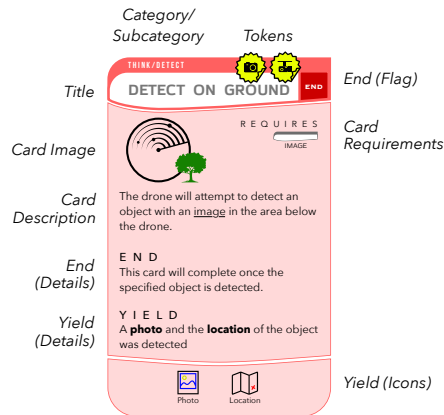
Figure 5: Example card DetectOnGround showing all possible elements: tokens, end condition, requires, end details, yield details, and yield icons. These elements are optional and will not appear on all cards.

3.2 Card Details

Figure 5 illustrates an example card design. The colored background (red) indicates the card class (*Think*). At the top-left, the card class and subclass are given with dark background (*Think/Detect*); below is the name of the card in large print ("Detect On Ground"). At the top-right, the yellow starbursts indicate the tokens consumed by this card (camera and gimbal), and the red "End" indicates that the card imposes an end condition. The curved design of the top header is intended to evoke the shape of a drone's propeller. In the middle section, every card has a unique card image for quick reference. To the right of image, we depict any prerequisite cards required for this card to be played, presented by color-coded class (a single *Image* card). Below, the card describes its purpose, the requirements of its end condition (if applicable), and the data it produces (called *yields*).

  We categorize cards by function into one of five classes.

1. *Action*: Causes the drone hardware to perform some behavior. Example: FlyTo commands the drone to go to the GPS coordinate specified by an *Input* card.

2. *Input*: Specifies a numerical, categorical, textual, or other data input used by another card. Example: Location represents a GPS coordinate and is used by FlyTo.

3. *Hand*: Modifies execution behavior of the entire hand. Example: Any makes a

10

hand require only one end condition to be satisfied before execution proceeds to the next hand.

4. *Deck*: Modifies the execution behavior of the deck. Example: RepeatDeck repeats execution of the deck from the first hand after all hands have completed. Presently, this is the only *Deck* card we have defined, but future *Deck* cards can exert global rule modifications across all hands (e.g. speeding up or slowing down all movements, setting resolution options for the camera, etc.).

5. *Token*: Grants access to underlying hardware. Starburst tokens are consumed (usable by only one card in a hand), circular tokens are not. Example: Movement is used to control a drone's movement, and Camera provides access to the camera.

In our design, *Action* cards have also been split up across a number of subclasses according to their function.

1. *Movement* cards control the movement of the drone. Example: FlyTo causes the drone to fly to the specified location.

2. *Tech* cards operate a hardware token. Example: TakePicture captures a single photo.

3. *Think* cards perform computational tasks. Example: DetectInAir applies computer vision on an Image to recognize nearby flying objects. Note that we do not specify how this computational task actually occurs, whether on the drone, on a ground station, or via a cloud service; we expect the specific implementations of these cards to determine their own implementation requirements.

4. *Trigger* cards are used for asynchronously-satisfied end conditions. Example: SetATimer becomes satisfied only after a specified amount of time has elapsed.

To aid discovery and recognition, cards of the same class, or subclass in the case of *Action* cards, are printed in the same color.

Cards from the *Input* class affect the execution of a single card, on top of which it is stacked. At a higher level, *Hand* cards may affect execution of multiple cards in one hand or the transition from one hand to the next. *Deck* cards affect execution across multiple hands. Otherwise, all cards within a hand are fully independent.

3.2.1 End Conditions and Branching

A hand is not complete until its *end conditions* are satisfied. Some cards (but not all) are marked with "End." These cards complete a finite amount of work prior to completion. For example, TakeAPhoto is marked as the card will complete as soon as the photo has been captured. Contrastingly, RecordVideo does not complete – it will keep recording video for an arbitrary amount of time, so long as the hand continues. By default, a hand ends when all end conditions in the hand are reached. This logic may be reversed using the Any card; when this card is played in a hand, once any card satisfies its end condition, it triggers the end of the hand and the movement to the next hand. Generally, all hands should have at least one card with an end condition. Hands in the last step – the "final" hand – are not required to have an end condition, as they will continue indefinitely until the drone's battery reaches a critical threshold, triggering it to immediately land.

The Branch enables logical branching, depending on which end condition is satisfied first. Using the And, Or, and Not cards, arbitrary Boolean logics can be created to determine when a hand ends and which hand should be executed next. For example, given *Action* cards A, B, C, and D, the following hand can be constructed: $(AND(A,B)$; $Branch(2a))$; $(OR(C,D)$; $Branch(2b))$. This hand will branch to hand *2a* when both *A* and *B* have ended, or it will branch to hand *2b* when either *C* or *D* have ended, whichever occurs first.

3.2.2 Side Effects and Yields

Drone activities have real-world consequences. For example, an item might be picked up and flown from one location to another. We do not model these kinds of physical side effects, but we do model side effects that result in data captured by the drone. Cards may have one or more *yields*. For example, the TakePhotos card yields a sequence of photos and a sequence of locations at which the photos were taken. In our design, yields are prominently presented at the bottom on each card so that one can quickly check for desired data capture.

During design and early testing, we debated whether yields should be reusable across game steps. Might the yield produced in one hand be consumed as an input to a later hand? While technologically feasible due to a strict adherence to strong typing, and perhaps desirable due to its added expressiveness, we felt this feature would add significant complexity and mental effort. Thus, we initially treated yields only as a persistent side effect – data yielded by the drone would be available for analysis only upon completion of program execution. However, during our evaluation, participants asked to reuse yields across hands. Therefore, our implementation of yields in CardKit does enable yields produced in prior hands

Symbol	Meaning	Example	Explanation
←	Denotes *Input* relationships	CoverArea ← Bounding-Box	BoundingBox is an input to CoverArea
() and +	Specifies multiple inputs to a card	Follow ← (RelativeToObject + TrackOnGround ← Image)	Follow requires a Relative card and a Track card as inputs. TrackOnGround also requires an Image as input
(#)	Indicates parameters to a card or identifies a branch	Repeat(3)	Repeats the hand three times
;	Separates cards in a hand	FlyTo ← Location ; TakePhotos	Drone will fly to a specified location and take photos at the same time
{}	Groups cards belonging to the same branch	(FlyTo ← Location ; Branch(1)) ; (DetectOnGround ← Image ; Branch(2))	Branch 1 will be followed if the drone arrives at the location. Branch 2 will be followed if the drone detects the input image.
[]	Specifies the kind of information provided to an Input card	Location [*my house*]	The location used for input is "my house".

Table 1: Syntax for card solutions

to be consumed as inputs in downstream hands.

3.2.3 Composition of a Single Hand and the Token System

In CCGs, there is often some notion of cost. Cost imposes limits and tradeoffs on what the player might be able to accomplish in the same hand. *Hearthstone* lets players cast spells, but only in number and strength as their casting costs allow with the current level of available mana. *Dominion* only allows players to buy a certain number of cards per turn based on available gold. We build on these models to express and enforce limits on what the drone might do concurrently, without requiring the user to explicitly understand any computer science concurrency idioms.

A single hand is composed of one or more cards. Although there are no specific limits to the number of cards in any one hand, there are some constraints on the number of times any particular card might be played and with which other cards it might be played. We represent these constraints through *tokens*, depicted as a starburst with an icon. In assembling a deck, the user adds a set of *Token* cards corresponding to physical characteristics of the drone on which the program will be run. Each *Token* card represents a single hardware capability of the drone. For example, a flying drone would have a single Movement token. If the drone possessed a camera mounted to a multi-axis gimbal, the player would also add Camera and Gimbal tokens to their deck. Within a single hand, a token may be *consumed* at most once. For example, all *Movement* cards consume a Movement token. Hence, it would be disallowed to play FlyTo and Circle in the same hand as they both require the Movement token. They could, however, be played in two successive hands.

13

Some tokens are not consumed upon use. These tokens define a hardware aspect of a drone that *can* be used concurrently. In our design, these tokens are represented as a circle rather than starburst (a circle is like a starburst with infinite points, reflecting that such cards may be used an arbitrary number of times). For example, the Humidity Sensor produces a stream of humidity values that can be consumed by multiple cards, such as LogHumidity, which logs the humidity value, and WaitForHumidity, which ends when the humidity reaches a given threshold.

3.2.4 Computational Play

In its paper form, users lay out a sequence of hands row by row on a flat surface. This layout specifies a program, which is read in sequence from top to bottom. As each hand is treated as an unordered collection of cards, left-to-right orderings may be disregarded. Figure 2 illustrates a representative layout of cards in hands.

In advanced scenarios, the program may incorporate conditional branching logic. In such cases, two or more hands may be considered as alternatives in the same step. We suggest each alternative hand be laid out horizontally with gaps to clearly indicate which subsequent hands branch from which end conditions in the previous hand.

4 Evaluation

Our evaluation consists of a paper prototype user study. We consider three primary metrics: ease, expressiveness, and engagement. Ease represents how much difficultly users perceive in selecting and laying out cards to accomplish their programming goal. We presented participants with specific tasks and evaluated their performance using several objective metrics such as accuracy, completion time, and established scales of cognitive load and usability. Expressiveness represents the diversity of tasks that can be programed. We asked participants to propose and implement a novel, creative drone task. Engagement represents the extent to which our format yields an enjoyable programming experience. Using Likert-format and free response metrics, we evaluate whether participants enjoyed our whimsical, game-like structure. We recruited 18 participants for an in-person, one-on-one, two-hour evaluation session at our facility. Table 2 shows a description of our participants. Our study protocol was reviewed and approved by our institution's human subjects research review board.

A member of our research team conducted the evaluation sessions. During these sessions, the evaluator began by explaining the format and goals of the study and received informed consent from the participant. The evaluator then explained the

Question	Participants (# / %)
Gender	Male (**11** / **61%**), Female (**7** / **39%**)
Age Range	≤20 (**2** / **11%**), 21-35 (**15** / **83%**), 36-50 (**1** / **6%**)
% time writing code	≤20 (**5** / **28%**), 21-40 (**3** / **17%**), 41-60 (**5** / **28%**), 61-80 (**3** / **17%**), ≥81(**2** / **11%**)
Experience with ...	
. . . Flying drones	Yes (**3** / **17%**), No (**15** / **83%**)
. . . Visual programming	Yes (**4** / **22%**), No (**14** / **78%**)
. . . Card games	Yes (**6** / **33%**), No (**12** / **67%**)

Table 2: Demographics and pre-study questionnaire.

basic rules via a short booklet and laid out all of the paper cards in stacks on a table in front of the participant. Explanations of rules and mechanics were kept consistent across participants to the extent possible.

In the training phase, the evaluator demonstrated three example tasks and their solutions. Next, in the evaluation phase, participants performed three experimental tasks on their own, with the evaluator providing feedback only in the case of syntactic rule violations. Each of these tasks were presented in a randomized order so that we may evaluate task difficulty independent of learning effects. When a participant made rule violations or other "syntactical" errors during the evaluation phase (i.e., errors that a real compiler could automatically detect and report in a digital format), we disclosed and explained these errors to the participant. No other feedback or hints were given during the evaluation phase; especially, logic errors were not disclosed. The accuracy of each participant's solution was scored against a pre-defined rubric.

After each task, participants completed the the NASA Task Load Index (TLX) [7] and were asked questions regarding their experience (e.g., how enjoyable or hard). The TLX measures perceived workload across six sub-scales: mental, physical, temporal (time pressure), performance, effort, and frustration. In our study, we applied it to understand the difficulty of each individual evaluation scenario. We also recorded the amount of time spent on each task and took a picture of the final outcome. At the end of the study, participants filled out a final questionnaire and completed the System Usability Scale (SUS) [2] to quantify effectiveness, efficiency, and satisfaction for the overall approach. Time and number of cards required per task serve as additional metrics of task complexity. For their time, participants received a $10 lunch voucher and the opportunity to fly a small consumer-grade drone indoors.

4.1 Drone Scenarios

We constructed six drone scenarios for our study, three for instruction and three for testing. For the training scenarios, participants were shown the solution and given

an explanation as to how the solution operated. For brevity, we show only the three testing scenarios below. Table 1 gives an overview of the syntax we use to present solutions in this paper; participants were shown solutions using the actual paper cards.

For the evaluation tasks, we provide our rubric to quantify the accuracy of participant answers. Since there is no single correct solution for any scenario, solutions will vary in number of cards, number of hands, and complexity. Participants' solutions were scored only for what they did correctly; no points were deducted for extraneous functionality.

Task A – Package Delivery*: A customer has put in a request for a package to be picked up from her house and dropped off at a friend's house. The drone will go to the customer's house and land. The drone will wait there until the customer has placed the package into the drone's claw. The customer will push the button when everything is ready. Once the button is pushed, the claw will close and the drone will fly to the friend's house. Next, the drone will drop the package off from a safe height (0-5 feet above the ground). After the package is dropped off, a sound will be played to indicate the package has been delivered. Return back home once finished.*

    Hand 1: FlyTo ← Location [*pickup*]
    Hand 2: Land ; WaitForButtonPush
    Hand 3: CloseClaw
    Hand 4: FlyTo ← Location [*delivery*]
    Hand 5: HoverToAltitude ← Distance [*5 ft.*]
    Hand 6: OpenClaw
    Hand 7: PlayAudio ← Audio [*success*]
    Hand 8: ReturnHome

Scoring rubric: go to customer's house (5), land (3), wait for button push (5), pick up package / close claw (5), go to friend's house (5), drop off package (5) from a safe height (3), play a sound (3), return home (5). 39 possible points.

Task B – Skiing Coverage*: You are a journalist covering the skiing portion of the Olympic Games. You would like to record every skier that goes down the slope. The drone is at the top of the slope. The drone uses the camera to locate the first skier (nearby). The drone will record video while the skier goes down the slope. Once the skier has reached the bottom of the slope, the drone will stop following (and recording). At all times, the drone should remain above the tree line (300 feet). Upon reaching the bottom, the drone should return to the top of the slope and repeat for all subsequent skiers.*

    Hand 1: Hover ; DetectOnGround ← Image [*skier*]
    Hand 2: Follow ← (RelativeToObject + TrackOnGround ← Image [*skier*]) ; Altitude [*300 ft.*] ; WaitUntilLocation ← Location [*bottom of slope*] ; RecordVideo
    Hand 3: ReturnHome ; Altitude [*300 ft.*]

Hand 4: RepeatDeck

Scoring rubric: detect skier (5), follow skier (5), record video (5), stay above tree line (3), stop at bottom of slope (5), repeat (5). 28 possible points.

Task C – Gas Detection*: You are a gas technician and a customer has called you for a gas problem. You need to find the gas leak in the area around the house to start working on the issue. You would like to use a drone to locate the gas leak. The drone will fly around the house and search for a higher than normal gas level. If this is found, the drone will wait there for 3 minutes while playing an alarm. After the 3 minutes are over, the drone will land. If a gas leak has not been detected, the drone will return back home.*

Hand 1: {CoverArea ← (BoundingBox [*yard*] + Avoid ← BoundingBox [*house*]) ; Branch(A)} ; {WaitForGas ← Threshold [*high*] ; Branch(B)} ; Any

Hand 2 Branch A: ReturnHome

Hand 2 Branch B: SetATimer ← Duration [*3 min.*] ; PlayAudioLoop ← Audio [*alarm*]

Hand 3: ReturnHome

Scoring rubric: survey area (5), detect gas (5), wait (3), play alarm (3), land (5), branching logic (10), return home if no leak (5). 36 possible points.

Task D — Create Your Own Scenario*: Define and implement any task of your choosing. If required, you may define new cards (the moderator will help ensure that another card does not already capture identical functionality).*

4.2 Ease of Use

Irrespective of correctness, participants used a similar number of cards as in our own solutions. Participants used the fewest cards for *Skiing* (median = 12), despite tending to spend the most time on that scenario (see Figure 6, a CDF of time elapsed by task). Figure 7 presents a histogram of rubric scores across all tasks. Overall, participants implemented an advanced drone scenario with about 15 cards in about 12 minutes, performing the task with a high level of accuracy (M=89% correctness) with minimal training.

Figure 8(a) presents a histogram of TLX scores across all tasks (lower is better). Figure 8(b) decomposes the TLX scores along the six sub-scales. As we did not inform participants of their mistakes between tasks, there were no significant differences in rubric score across task scenarios, $F_{(2,48)} = 0.82$, p=n.s.

Figure 9(a) presents a histogram of SUS scores. Figure 9(b) decomposes the SUS scores along the ten sub-scales. We find that participants rated our system's usability positively (M (SD) = 67% (15%)), but qualitative feedback shows room for improvement (Table 4). In addition, we expect some depression of SUS scores
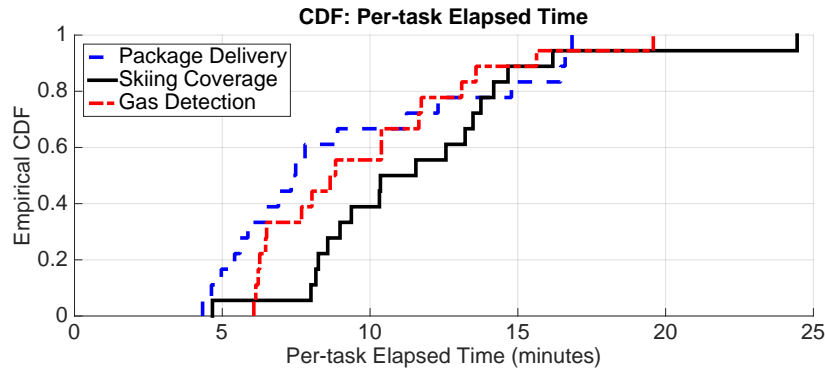
Figure 6: Cumulative distribution (CDF): time elapsed, by task. There was no significant difference in completion times across scenarios, $F(2,48) = 0.44$, p=n.s.
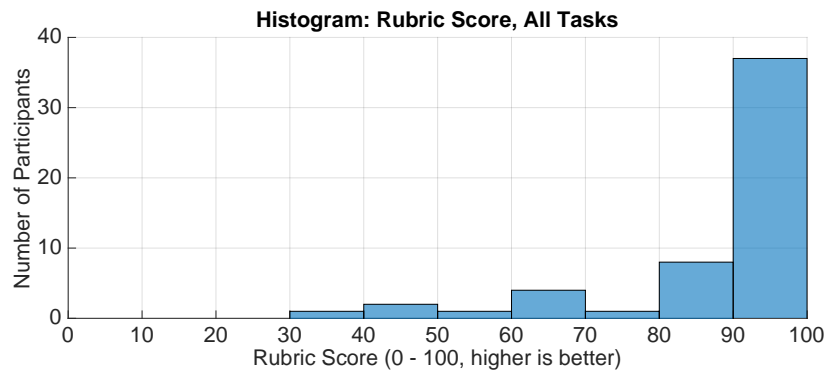

Figure 7: Histogram: Normalized rubric score. M (SD) = 89 (16) points.

due to novelty effects and the moderate levels of Mental Demand and Effort shown in the TLX (Figure 8(b)).

In the post-study questionnaire, we asked participants to rate their feelings toward the drone cards (positive or negative), the effectiveness of the cards in implementing the scenarios, their enjoyment in using the cards, their overall satisfaction with the cards, and the extent to which they felt the cards could be used to implement a wide variety of scenarios (Table 3). These items were all rated on Likert scales. The responses to these items were significantly correlated and thus averaged to form a single scale of efficacy (Cronbach's α = .78). Overall, normalized efficacy was rated moderately (M (SD) = 61% (11%)).

Table 4 summarizes manually-coded free responses from the post-study questionnaire. Participants generally found the cards effective for implementing drone
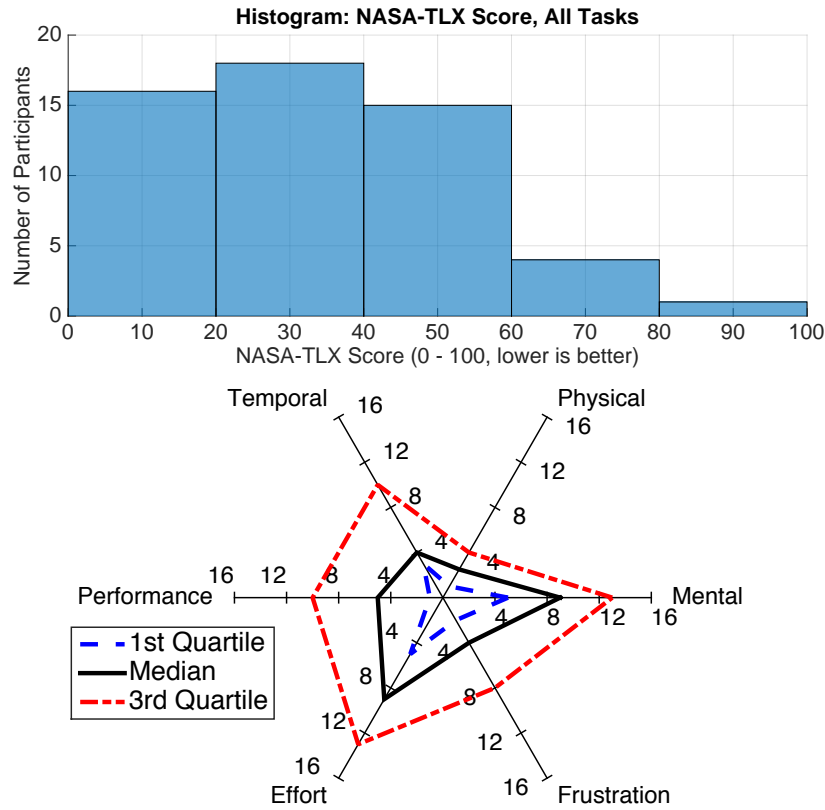
Figure 8: (a) Histogram: TLX score. M (SD) = 33 (18) points. (b) Radar: TLX score (all tasks), by subscale.

scenarios. In free response, a majority of participants found the cards intuitive (72%), and many complemented the visual aspects of our design (39%). Several criticized the quantity of cards (33%) and complexity of the rules (22%). Control flow and movement were most commonly cited for confusion.

4.3 Expressiveness

Participants' ambitions in the "create your own scenario" task varied widely. Solutions ranged from simple one-hand programs with four cards to complex programs with many hands and 27 cards – far more elaborate than any of our training or evaluation scenarios. Content ranged from practical (e.g., watering plants) to fantastical (e.g., killing zombies). In the post-study questionnaire, participants rated that they felt drone cards are expressible enough to implement "a great deal" (50%) or "some"
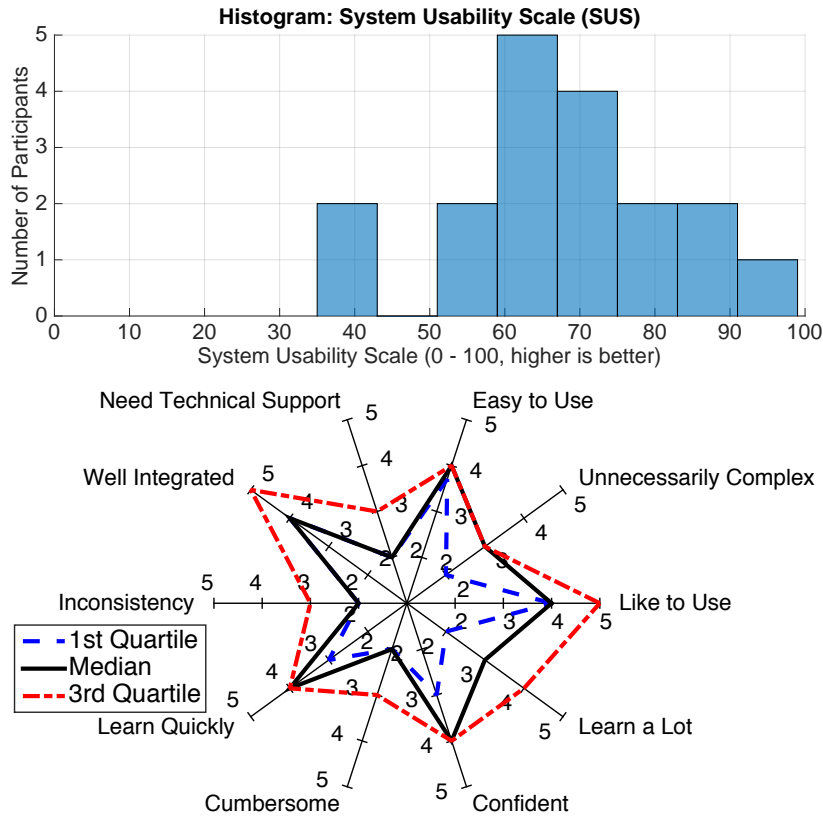
Figure 9: (a) Histogram: SUS score. M (SD) = 67 (15) points. (b) SUS score, by subscale. 1 = Strongly Disagree, 2 = D, 3 = N, 4 = A, 5 = SA.

(50%) drone scenarios (Table 3).

4.4 Engagement

Overwhelmingly, participants felt "very positive" or "generally positive" (94%) about the drone cards, and found the process "very enjoyable" or "generally enjoyable" (94%). Only one participant was "generally negative." (Table 3). Two participants explicitly identified our approach as "fun" or "playful" in free response (Table 4).

Participants did find several disagreeable aspects, especially the number of cards to learn and complexity of programming. Many offered suggestions for improvement; chiefly, digitizing the process through an app (39%) or by making specific changes to the cards (e.g. adding or replacing cards).

Question	Participants (%)
Overall, how do you feel about the drone cards?	V. positive (**44%**), Generally positive (**50%**), Mixed (**0**), Generally negative (**6%**), V. negative (**0**)
Overall, how effective do you think the drone cards are for implementing drone scenarios?	Excellent (**17%**), Very good (**61%**), Good (**17%**), Fair (**6%**), Poor (**0**)
Overall, how enjoyable was it to implement drone scenarios with the cards?	Very enjoyable (**50%**), Generally enjoyable (**39%**), Somewhat enj. (**11%**), Very unenjoyable (**0**)
Overall, how satisfied were you with the cards?	Very satisfied (**44%**), Generally satisfied (**50%**), Somewhat dissatisfied (**6%**), V. dissatisfied (**0**)
To what extent do you feel that this system can be used to express a wide variety of drone scenarios?	A great deal (**50%**), Some (**50%**), Only a few (**0**), Not many (**0**)

Table 3: Post-study questionnaire. Responses were significantly correlated (Cronbach's $\alpha = .78$), forming a single scale of efficacy, M (SD) = 61% (11%).

5 Discussion

Our results generally confirm the efficacy of our card-based programming model. Our primary metrics of ease, expressiveness, and engagement rated favorably for most participants. Some participants were extremely positive, even after completing a demanding two-hour test session. Though, perceptions were not uniform. A few participants found the approach unintuitive, complex, or inferior to traditional programming. Likely, these impressions are unavoidable; although many people enjoy card games, others find them complex, frustrating, or simply not fun. Overall, we find substantial evidence that card-based programming provides a useful abstraction to control complex systems such as drones.

5.0.1 Applicability to Other IoT Devices

The card-based programming metaphor can be applied to domains beyond drones, and we readily see applicability to robots and other IoT devices that possess a capacity for sensing, thinking, acting, and communicating. The abstraction of an *Action* card for performing behaviors, coupled with *Token* cards as a serializing API to the underlying hardware, cover the capabilities of action and communication. The data flow provided by *Input* cards and *yields* can be used for sensing capabilities, and the decoupling of the *use* of a *Think* card from its underlying implementation shift the burden of managing difficult computation from the end-user to the implementer of the hardware tokens.

As a concrete example, TJBot [4] is an open-source paper robot designed to teach people about building cognitive systems. The computation involved in

"bringing TJBot to life" involves the use of IBM Watson services to make him speak, listen, converse, and understand language and emotion. Cards for each of these facilities could be developed, abstracting the technical details of how (using machine learning to generate speech) and where (in the cloud) this behavior is performed from the end user who simply wishes to make their robot speak. The onus of responsibility for implementing these technical details (accessing Watson APIs via an SDK) lies with the developer in the implementation of TJBot's token cards.

5.0.2 Real-world Considerations

Up to this point, our presentation of our card-based programming metaphor has been purely theoretical – the design and evaluation of the cards was conducted on paper. In order to understand whether the system would actually work for programming a real drone, we have implemented our system as a series of Swift frameworks. These frameworks are responsible for the construction and execution of card-based programs, as well as the specific behaviors of our drone cards and their interface to a real drone API (DJI). Overall we have found the implementation to be a straightforward task, but we learned a few key lessons.

Separation of functionality. A clean separation of functionality makes the card metaphor scale across different devices. Our implementation consists of these separate frameworks, with each subsequent framework dependent on the ones listed prior:

- *CardKit*: specification of card programs (e.g. the "compiler").

- *CardKit Runtime*: execution engine for card programs, manages threaded execution of *Action* cards, yield dataflow, hand branching logic, and error handling.

- *Drone Cards*: definition of drone-specific *Action*, *Input*, and *Token* cards (e.g. FlyTo, TakePhoto), plus implementation of *Action* cards against generic token interfaces.

- *Drone Tokens*: implementation of drone token cards against a specific drone API (e.g. the hardware interface layer). Support for new drone APIs is achieved by creating *only* additional drone token implementations.

Action card implementations communicate with *Token* implementations using a well-defined, generic interface; thus, our set of drone cards can be used across multiple drone APIs or simulators by having separate implementations of just the token cards. Or, a wholly new kind of device can be supported (e.g. TJBot [4]) by

defining a new set of cards and token implementations, without having to change the underlying specification and execution engines.

Error handling. Errors happen in code. To cope with this, we implemented an emergency stop feature [16] that can be triggered either by *Action* or *Token* cards. Emergency stop halts execution of the card program and signals token implementations to recover from the error. For example, the e-stop handler of the Movement token triggers a landing.

Platform mobility. During development of the frameworks, we recognized that the task of *constructing* a card program may occur at a different time or on a different device than the task of *executing* it. Thus, not only did we map these two tasks into different frameworks, we also strongly enforced that the data structures comprising a card program be *serializable* and *portable*. In our frameworks, card programs are serializable to JSON, enabling them to be created on one device (e.g. a laptop) and executed on another (e.g. an iPad).

We have open sourced our frameworks [2] and encourage developers to build end-user apps that allow people to explore and create their own card-based programs for drones or other IoT devices.

5.1 Limitations

Although the spirit of the card programming metaphor is to be accessible to non-programmers, our evaluation used participants who were highly skilled in computational thinking. Given our desire to evaluate the expressiveness and effectiveness of our system, controlling the technical background of participants enabled us to understand whether difficulties in solving the programming tasks were due to failures of our own system rather than failures of participants to grasp the mechanics of the task. We are eager for future evaluations of our system with new populations of non-programmers and non-technical users to fully understand whether it fully lives up to its intended design.

Additionally, our system is not meant to enable the development of *arbitrary* programs. We do not aim to provide a Turing-complete programming language; rather, we are satisfied if our model is able to express some percentage of the possible drone scenarios easily. The "Create Your Own Scenario" task aimed to test the limits of peoples' creativity to find breakdowns or limitations in our model, and participants came up with many scenarios we never anticipated. Participants also rated our model as having a high degree of expressiveness, giving us confidence in its utility.

---

[2]https://github.com/CardKit

6 Conclusion

Drones are just beginning to revolutionize industries and day-to-day life. However, their use today is generally constrained to a fraction of their overall potential. We propose a card-based programming paradigm to automate diverse use cases in an intuitive, expressive, and fun manner. Heartened by participants' generally positive response to our paper prototype, we have developed and released CardKit, an open-source framework for card-based programming.

References

[1] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable Programming: Blocks and Beyond. *Commun. ACM* 60, 6 (May 2017), 72–80. DOI:http://dx.doi.org/10.1145/3015455

[2] John Brooke. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry* 189, 194 (1996), 4–7.

[3] Martin C. Carlisle, Terry A. Wilson, Jeffrey W. Humphries, and Steven M. Hadfield. 2005. RAPTOR: A Visual Programming Environment for Teaching Algorithmic Problem Solving. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '05)*. ACM, New York, NY, USA, 176–180. DOI:http://dx.doi.org/10.1145/1047344.1047411

[4] Victor C. Dibia, Maryam Ashoori, Aaron Cox, and Justin D. Weisz. 2017. TJBot: An Open Source DIY Cardboard Robot for Programming Cognitive Systems. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA '17)*. ACM, New York, NY, USA, 381–384. DOI:http://dx.doi.org/10.1145/3027063.3052965

[5] DJI. 2017 (accessed September 5, 2017). *DJI Developer*. http://developer.dji.com.

[6] JS Foundation. 2017 (accessed September 5, 2017). *Node-RED*. http://nodered.org.

[7] Sandra G Hart and Lowell E Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. *Advances in psychology* 52 (1988), 139–183.

[8] Tickle Labs Inc. 2017 (accessed September 5, 2017). *Tickle: Program Star Wars BB-8, Drones, Arduino, LEGO, Dash & Dot, Sphero, Robots,*

Hue, Scratch, Swift, and Smart Homes on your iPhone and iPad. `http://tickleapp.com`.

[9] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. 1988. Fabrik: A Visual Programming Environment. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA '88)*. ACM, New York, NY, USA, 176–190. DOI: `http://dx.doi.org/10.1145/62083.62100`

[10] David J. Malan and Henry H. Leitner. 2007. Scratch for Budding Computer Scientists. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '07)*. ACM, New York, NY, USA, 223–227. DOI:`http://dx.doi.org/10.1145/1227310.1227388`

[11] John Maloney, Leo Burd, Yasmin Kafai, Natalie Rusk, Brian Silverman, and Mitchel Resnick. 2004. Scratch: a sneak preview [education]. In *Creating, Connecting and Collaborating through Computing, 2004. Proceedings. Second International Conference on*. IEEE, 104–109.

[12] B. A. Myers. 1986. Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '86)*. ACM, New York, NY, USA, 59–66. DOI:`http://dx.doi.org/10.1145/22627.22349`

[13] Parrot. 2017 (accessed September 5, 2017). *Parrot for Developers*. `http://developer.parrot.com`.

[14] Mel Siegel. 2003. The sense-think-act paradigm revisited. In *Robotic Sensing, 2003. ROSE'03. 1st International Workshop on*. IEEE, 5–pp.

[15] Tiago Boldt Sousa. 2012. Dataflow programming concept, languages and applications. In *Doctoral Symposium on Informatics Engineering*, Vol. 130.

[16] A. Steinfeld. 2004. Interface lessons for fully and semi-autonomous mobile robots. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, Vol. 3. 2752–2757 Vol.3. DOI:`http://dx.doi.org/10.1109/ROBOT.2004.1307477`

[17] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. 2014. Practical Trigger-action Programming in the Smart Home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 803–812. DOI:`http://dx.doi.org/10.1145/2556288.2557420`

[18] Kirsten N. Whitley. 1997. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing* 8, 1 (1997), 109–142.

[19] Mark Young, Danielle Argiro, and Steven Kubica. 1995. Cantata: visual programming environment for the Khoros system. *ACM SIGGRAPH Computer Graphics* 29, 2 (1995), 22–24.

[20] Shengdong Zhao, Koichi Nakamura, Kentaro Ishii, and Takeo Igarashi. 2009. Magic Cards: A Paper Tag Interface for Implicit Robot Control. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. ACM, New York, NY, USA, 173–182. DOI:http://dx.doi.org/10.1145/1518701.1518730

What did you like about this card methodology?	#	%
Intuitive (e.g., easy to learn, simplifies complex tasks)	13	72%
Visual aspects of design (e.g., iconography, color coding)	7	39%
Systematic / consistent	3	17%
Expressiveness	2	17%
Fun / playful approach	2	11%
Useful	1	6%

What did you dislike about this card methodology?	#	%
Many cards / rules to remember	6	33%
Programming difficulties / complexity	4	22%
Control flow (hand system of sequential vs. parallel)	2	11%
Limited constructs (e.g., missing programming features)	2	11%
Size of physical layout	2	11%
Aspects of visual design	1	6%
Difficulty locating physical cards	1	6%
Inconsistencies	1	6%
Not intuitive	1	6%

What improvements would you make to this system?	#	%
iPad app / computerize / GUI	7	39%
Add cards (support new use cases, new functionality)	3	17%
Replace cards (merge functionality into sophisticated cards)	3	17%
Aspects of design / presentation	2	11%
Add programming constructs	1	6%
Improve consistency	1	6%
Separate "basic" from "advanced" cards	1	6%

Which cards most confused you?	#	%
Control flow (e.g., Branch / Any cards, Wait)	6	33%
Movement (i.e., *Movement* cards + related inputs / modifiers)	5	28%
Non-movement actions (e.g, *Tech* cards)	3	17%
Computation (i.e., *Think* cards)	2	17%

Which cards were most useful to you?	#	%
Movement (i.e., *Movement* cards + related inputs / modifiers)	13	72%
Computation (i.e., *Think* cards)	5	28%
Control flow (e.g., Branch / Any cards, Wait)	5	28%
Non-movement actions (e.g, *Tech* cards)	4	22%

Table 4: Post-study questionnaire free response after coding.