

# Teaching Kodu With Physical Manipulatives

---

**David S. Touretzky**  
**Computer Science Department**  
**Carnegie Mellon University**  
**Pittsburgh, PA 15213-3891**

This article will appear in *ACM Inroads*, December 2014.

Categories and Subject Descriptors: K.3.2 Computer Science Education

General Terms: Languages, Human Factors

Keywords: Kodu, Manipulatives, Visual programming

## Abstract

In Microsoft's Kodu Game Lab, a visual programming environment for children, rules are constructed from sequences of tiles that act as virtual manipulatives. To further scaffold very young students' introduction to programming, I developed a set of physical tiles, now publicly available, to be used in pair programming activities and teacher demonstrations. While Kodu's virtual tiles are simple rectangles, the physical tiles use complex shapes and color to reinforce Kodu's 2D syntactic structure. The tiles were tested successfully in a pilot study with second grade students who had no prior programming experience and were seeing Kodu for the first time.

## Introduction

The value of manipulatives in K-12 mathematics instruction is widely recognized:

*“In every decade since 1940, the NCTM [National Council of Teachers of Mathematics] has encouraged active student involvement through the use of manipulatives at all grade levels. In fact, in their publication Principles and Standards for School Mathematics (2000) the NCTM explicitly recommends the use of manipulatives in the classroom.”* [10]

*“It is the position of the National Council of Supervisors of Mathematics (NCSM) that in order to develop every student's mathematical proficiency, leaders and teachers must systematically integrate the use of concrete and virtual manipulatives into classroom instruction at all grade levels.”* [17]

There are today a wealth of manipulative resources available to mathematics teachers, from simple counting tokens to Cuisenaire rods and algebra tiles. An even wider range is available in virtual form via the web, e.g., Utah State University's Java-based *National Library of Virtual Manipulatives* [30] or NCTM's *Illuminations* [16].

In computer science, physical manipulatives are a common feature of CS Unplugged activities [5]. GUI-based drag and drop programming frameworks such as Alice [4] [6] and Scratch [21] implement *virtual* manipulatives in the form of different shaped blocks corresponding to syntactic classes such as imperative statements, Boolean expressions, and loops. Scratch 2.0 has a total of 145 blocks in six different shapes [11]. Blocks can attach to one another only if they have the right shapes, making it impossible to construct syntactically invalid expressions.

One of the theoretical justifications offered for manipulatives is that they help students progress through a concrete-representational-abstract (CRA) learning sequence [7]. For example, in learning to count, students start with physical tokens (concrete), progress to marks on paper (representational), and then to written numerals (abstract). Manches and O'Malley critically survey current theories about manipulatives and suggest that their benefits lie in two areas: "*offloading cognition* – where manipulatives may help children by freeing up valuable cognitive resources during problem solving, and *conceptual metaphors* – where perceptual information or actions with objects have a structural correspondence with more symbolic concepts" [15].

Virtual manipulatives draw on our intuitive understanding of interactions with physical objects, but lie somewhere between the representational and concrete in the CRA sequence. In the opinion of some researchers, "virtual manipulatives do not replace the power of physical objects in the hands of learners" [17]. On the other hand, Clements [2] observed that with complex manipulative tasks, virtual manipulatives can offer advantages over physical materials, such as ease of saving state or making global changes. Clements goes on to suggest that rather than adhering to a strict CRA sequence, learners can derive greater benefit by exploring all three representations in parallel. This is the approach taken in the pilot study described later in this paper.

Microsoft's Kodu Game Lab [12] is a visual programming environment for children that uses a rule-based formalism to specify behaviors for characters in 3D virtual worlds. Kodu is one of the platforms recommended for K-8 educators by Code.org, along with Alice and Scratch [3]. In 2012, when the Republic of Estonia announced plans to teach programming to all its first graders [19], Kodu was cited as an initial framework to be used [27]. Kodu is also one of three languages, along with Scratch and Logo, recommended by the Computing at School Working Group for the new primary school computing curriculum in the U.K. [1]. Although Kodu is commonly described as a game development platform [9], it can be used to explore computation in a wide variety of ways.

In Kodu, rules are sequences of "tiles", and the rule editor supports this metaphor via its GUI interface [8]. Kodu does not use a drag-and-drop editor, and thus does not use shapes to enforce syntactic constraints the way Alice and Scratch do. As shown in Figure 1, tiles are simple rectangles. Instead, Kodu uses context-sensitive menus to add tiles to rules (by clicking on a plus sign), so that at each step, only syntactically valid tiles can be selected. One consequence of this is that unlike in Alice or Scratch, a student never has access to the entire inventory of primitives at one time. But this unusual GUI choice is not the most significant difference between Kodu and

Alice/Scratch/Blockly/etc. One could easily create a menu selection-based editor for any of the latter, or a drag-and-drop editor for Kodu.

What really distinguishes Kodu as a programming language is the level of primitives provided. Alice and Scratch are *graphics* languages, where the user specifies visual effects they want to achieve on the screen. Nothing happens in those systems unless the user writes code to make it happen. In contrast, Kodu is an *autonomous robot* programming language. Users specify actions a character is to perform, such as approaching or grabbing an object, but the graphical details and sound effects are handled by the character itself. When Kodu characters are not executing instructions, they look around, fidget, and make amusing noises. Another distinguishing feature of Kodu is that its worlds are true 3D environments with complex terrain and physics that includes gravity, friction, and collisions. Thus, even simple Kodu programs can be very engaging.

## Kodu Syntax

A Kodu world consists of terrain plus a collection of programmable objects. Characters are objects that can move on their own, such as kodus or flying fish. Other types of objects, such as apples, coins, and trees, cannot move on their own, but can still be programmed to interact with characters and other objects. Each object's "program" is a collection of WHEN-DO rules organized into pages. Only the rules on the currently active page are eligible to execute.

Unlike conventional programming languages where expressions can be arbitrarily complex, Kodu rules do not have a recursive structure. They are regular expressions with a simple syntax:

$$\mathbf{WHEN} \ [predicate \ [predicate\text{-}argument\dots]] \ \mathbf{DO} \ [action \ [action\text{-}argument\dots]] \quad (1)$$

Two common predicates are "see" and "bump". Although there are exceptions, predicate arguments are usually nouns (e.g., "apple") or adjectives (e.g., "red"). Two common actions are "move" and "eat". Action arguments are typically prepositions (e.g., "toward"), pronouns ("me" or "it"), or adverbs (e.g., "slowly"). With one exception, objects cannot appear explicitly as action arguments. Thus, the only way for a rule to act on a specific object is to establish a reference to it with a WHEN predicate and then use "it" on the DO side, as in rule 2 of Figure 1. The exception is the "create" and "launch" actions, which generate new objects and thus require a noun as argument.

The arguments to a predicate or action are unordered<sup>1</sup>, so “red apple” and “apple red” have the same meaning. Likewise, a “5 points” tile followed by a “10 points” tile, or the reverse, both encode a value of 15 points.

Kodu implements a form of block structure through rule indentation. A rule that is indented can run only if its parent rule (the closest rule above it that is less indented) is true. Thus, in Figure 1, the character will play the coin sound only when it bumps an apple. If rule 3 were not indented, the character would play the coin sound all the time, since an empty WHEN predicate is considered to be always true.

## Kodu Tile Design

The Kodu rule editor is highly context-sensitive. When a tile is to be added, the editor dynamically constructs a pop-up menu taking into account both the capabilities of the object being programmed and the nature of the tiles already selected for the current rule. In constructing physical Kodu tiles, we cannot duplicate this level of semantic constraint, but we have the opportunity to introduce geometric features to enforce most syntactic constraints. The current design uses four basic tile shapes: (1) WHEN-Predicate, (2) Predicate-Argument, (3) DO-Action, and (4) Action-Argument, plus four additional shapes for less common cases described below, and two shapes for rule indentation that are not associated with any Kodu virtual tile. All ten shapes are shown in Figure 2. The WHEN-related tiles use green plastic, while the DO-related tiles use blue plastic, to reinforce the notion that rules have a bipartite structure consisting of a WHEN part and a DO part. Thus, every rule begins with green tiles and ends with blue ones. The indentation tiles are yellow. It should be evident from the first two rows of the figure that the only tile sequences that can be constructed are those that fit the regular expression given in (1). A schematic page of Kodu code using all ten tile shapes is shown in Figure 3. Some observations about the tile design follow:

- Kodu rules may have an empty predicate or action. While these could be represented by full-size WHEN-Predicate or DO-Action tiles with no label, we instead use special short tiles to more closely match the look of the Kodu GUI. These Empty-WHEN and Empty-DO tiles are the first two tiles in the second row of Figure 2. The Empty-DO tile is a terminal tile; it is the only non-indentation tile with no right hand connector.
- To accommodate the “create” and “launch” tiles, which are the only actions that can take an object name as argument, we use a special DO-Create tile with a unique connector. This connector mates only with another special tile, the Create-Argument tile, which is labeled with an object name such as “apple” or “ball”. The DO-Create and Create-Argument tiles are the last two tiles in the second row of Figure 2. The Create-Argument tile can be followed by regular Action-Argument tiles such as adverbs or color names.
- The Predicate-Argument, Action-Argument, and Create-Argument tiles are physically symmetric about their horizontal axes, so labels can be affixed to both sides, reducing the number of plastic pieces required.

---

<sup>1</sup> The only exception to arguments being unordered is the “random” tile, which takes two distinct numeric arguments, one immediately to its left and the other immediately to its right. Both are optional.

- Color names can appear as either predicate arguments or action arguments. Since these take different shapes, color tiles must be duplicated. Kodu uses a slightly different graphic for colors used as score names, e.g., there is a special tile for “red score”, but it seemed wasteful to retain this distinction in the physical tiles, so we just use the generic “red”.
- Kodu provides a large number of sounds an object can play. We can keep costs down by only creating tiles for the most common sounds and providing a generic “sound” tile to stand in for any of the others.
- Rules are linked into sequences using the vertical connectors on the WHEN-Predicate and indentation tiles. The remainder of each rule floats free, but is somewhat constrained by the horizontal connectors. This makes it easy to experiment with different rule orderings or indentation levels, as only the WHEN tile needs to be detached in order to move the entire rule.
- A well-formed “page” of physical Kodu rules has a solid left edge and is completely connected, so that dragging any tile in any direction will move the entire page.

The tiles are constructed from laser-cut eighth-inch thick acrylic. Laser-cut acrylic can be quite sharp, so for safety, all corners have a 0.5 mm fillet applied. The first prototypes used laserprinted icons affixed by covering the artwork and the tile with clear tape. The current design uses one inch square self-adhesive labels, which can be printed on a photocopier and are easier to apply than tape. Although they are more expensive, polyester labels are preferred because they offer better durability and washability than paper labels. The physical tile realization of the apple eating program is shown in Figure 4.

These tile manipulatives are open source. Anyone wishing to use them is invited to visit my Kodu Resources page [29] for the necessary files.

## The Issue of Rule Indentation

Kodu’s use of indentation to create block structure is not intuitive for young children, an issue first brought to our attention by Jill Denner. Children may not be familiar with the uses of indentation in written language, and there is nothing in the visual depiction of an indented Kodu rule to associate it specifically with the rule above rather than the rule below. Touretzky et al. previously proposed a graphical convention to make this association explicit [28], but as of this writing, the Kodu visual interface has not changed. To help children associate indented rules with the parent rule above, there are special indentation tiles shown in the bottom row of Figure 2.

The tile on the right in Figure 2 is the basic Indent tile. It is used to indent the current rule by one level relative to the parent, which is all that Kodu permits. Examples can be seen in Figure 3, rules 2 through 6. The tile’s top connector mates with the bottom connector of the tile in the rule above, and the shape of the tile prevents any combination with other tiles that would produce a rule with further indentation.

The tile on the left in the bottom row of Figure 2 is called the More-Indent tile. It appears to the left of the Indent tile and is used for rules that are indented multiple levels. If a rule requires  $n$  levels of indentation, it will be assembled using  $n-1$  More-Indent tiles followed by one Indent tile, as shown in rules 4 and 5 of Figure 3.

The arrow cut into the indent tiles, modeled after the arrows in [28], visually connects the indented rule with the predicate of its parent rule. So rules 2, 3, and 6 depend on the predicate of rule 1. With multiple levels of indentation, as in rules 4 and 5, there are multiple arrows, each referencing a controlling predicate above. These arrows are visual reminders that rules 4 and 5 can only run when the predicates of *both* rules 1 and 3 are true.

While indentation can increase by at most one level relative to the rule immediately above, it can decrease by any number of levels at once. Thus, a Kodu rule at indentation level  $n$  can be followed by a rule with anywhere from 0 to  $n+1$  levels of indentation. The tile design makes this visually apparent. One need only look at the bottom connectors on a rule to see the indentation options available for the rule that follows. (The  $n$  open bottom connectors allow for  $n+1$  indentation levels because the rightmost bottom connector can be used to either continue the current indentation level or increase it by one.)

## Observations from the Pilot Study

A group of four second grade students (mean age 8 years 5 months) from a public charter school (Propel McKeesport, in McKeesport, PA) received three sessions of Kodu instruction, each 45 minutes to an hour in length, over the course of two weeks in June 2014. All students had prior experience using computers, including playing computer games, but no prior exposure to programming. Students worked in pairs consisting of one boy and one girl. Each pair ran Kodu on a laptop with an Xbox 360 game controller as the input device.<sup>2</sup> In addition to the laptops, each pair of students was supplied with a set of Kodu tile manipulatives, and three flashcards with Kodu idioms they were to learn. These flash cards were inspired by the Scratch Cards created by Natalie Rusk [23]. The Kodu idioms were defined by the author and will be the subject of a separate publication.

Students first took turns using the game controller to navigate through a 3D Kodu world, looking for specific objects in a “treasure hunt” task. Once they were comfortable with navigation, they were shown a world with five apples and a kodu. At this point the laptops were put aside temporarily and the students were shown the flash card with the code for pursuing and consuming apples (the first two rules of Figure 1). The teacher discussed this two-rule program and had the students reproduce it using the tiles. After they had done so, they went back on the laptops and were introduced to the Kodu rule editor. They then programmed the kodu to eat the apples, using as a reference the two-rule tile structure they'd created.

---

<sup>2</sup> Kodu Game Lab runs on the Xbox 360, Windows PCs, and the Microsoft Surface Pro.

Students moved back and forth between the tiles and the rule editor in all three sessions. In the second session they learned to add a color filter tile to treat red and blue apples differently. In the third session they were introduced to rule indentation. The following general observations were made during the sessions:

- Students took to the tile manipulatives immediately and had no trouble assembling and rearranging them.
- Shared use of physical materials can facilitate peer interaction [32]. We saw this with the tiles when students were asked to take turns constructing rules. Without explicit instruction, students cooperated by having one student in each pair search the pile for needed tiles while the other one assembled the rule.
- We also observed students spontaneously browsing through the tiles, getting an idea of the range of primitives available.
- For manipulatives to be effective as learning aids, students need to be allowed to experiment with them rather than sticking to a prepared script [17]. It is during this exploration that the use of shapes to enforce Kodu syntactic constraints comes into play. We observed an instance of this when a student tried to make a rule that said “WHEN bump apple DO eat apple”. This isn’t allowed in Kodu, and the tiles wouldn’t fit together. The student’s partner then suggested (correctly) that the action should be “eat *it*”.
- At times, the teacher also manipulated the tiles to illustrate the discussion of the effects of adding or removing a color filter or adding or removing indentation.
- In the second session, the two groups constructed slightly different solutions to the problem of eating only red apples and grabbing only blue ones. The shorter solution was:

```
[1] WHEN see apple DO move toward (2)
[2] WHEN bump red apple DO eat it
[3] WHEN bump blue apple DO grab it
```

The longer solution was:

```
[1] WHEN see red apple DO move toward (3)
[2] WHEN bump red apple DO eat it
[3] WHEN see blue apple DO move toward
[4] WHEN bump blue apple DO grab it
```

Students discovered that these solutions produced different behaviors<sup>3</sup>. The teacher was able to place both groups’ tile solutions side by side on the table to facilitate a discussion of their differences. This could not have been done in the Kodu rule editor.

---

<sup>3</sup> This difference is due to the way Kodu binds objects to predicate arguments and prioritizes conflicting rules. The solution in (3) causes the character to eat all the red apples before grabbing its first blue one, while the solution in (2) has the character always proceed to the closest apple of any color.

At the conclusion of each session, students took a short quiz to assess what they had learned. The first two quizzes mostly involved recapitulation of the material in the lesson, e.g., the basic “Pursue and Consume” idiom and the use of a color filter. All the students showed mastery of these concepts.

In the third session, indentation was introduced via the “Do Two Things” idiom (Figure 5), and reinforced with the “Count Actions” idiom, a special case of Do Two Things. During the lesson the students also experimented with removing indentation from rules to see how this affected the character’s behavior. The first example was the program in Figure 1 that they had just written under the teacher’s guidance. They found that removing the indentation from rule 3 caused the kodu to play the coin sound all the time. They then re-applied the indentation and verified that the program worked correctly again. In the second example, a cannon was launching red or blue hearts toward the kodu, and the kodu had to eat and count only the red ones. The code for this, which introduced the Count Actions idiom and was the first use of scores the students had seen, was:

```
[1] WHEN see red heart DO move toward                                     (4)
[2] WHEN bump red heart DO eat it
→ [3] WHEN DO score red 1 point
```

The students discovered that removing the indentation from rule 3 cause the red score to run up at a very high rate, with accompanying graphics and sound effects, which they found highly amusing. Again, they were instructed to re-apply the indentation and verify that this fixed the problem.

In the quiz for session 3 we decided to show the students some buggy programs – without telling them that they were buggy – to see if they could correctly predict the programs’ behavior. Students were shown the program in Figure 6 and asked several questions about it. In session 2 they had explored a world where a “cycle” (motorcycle) character tossed out balls one at a time, and the kodu had to eat or grab them. They were shown an image of this world along with the buggy program, and asked the series of questions shown in Table 1, along with their answers.

Students 2 and 3 answered perfectly, and their responses to Q3 show that they understand the dependency relationship between rules 1 and 2. Student 1’s answer of “all the time” is not correct, since the kodu will not play the sound during the brief intervals between its eating one ball and the cycle launching the next ball. But the kodu spends most of its time chasing down balls, and so it will be playing the coin sound *almost* all of the time. Unfortunately, the responses to Q2 and Q3 show that the student has not made the connection between rule 2’s indentation and its dependence on rule 1’s predicate.

Student 4’s responses are also incorrect, but for an interesting reason. This student chose to interpret the indentation as associating rule 2 with the rule *below* rather than above, and they illustrated this by drawing in the



arrow shown in Figure 6. Why did the student do this? Most likely because it recast the buggy program as a correct one, equivalent to Figure 1, and shown on the Do Two Things idiom flash card the students had available to them (Figure 5). In the brief session introducing indentation, we had not shown the students any code where an indented line was *followed* by a non-indented one, nor had we emphasized that indentation only attaches upward, not downward. The student's answers to Q1 and Q3 are actually correct with respect to their not unreasonable re-interpretation of the program. Their reference to "apple" rather than "ball" in Q1 is probably just a slip because most of the examples they'd seen so far had used apples.

In the last part of the quiz, students were presented with the two programs shown in Table 2, which differ only in the indentation of rule 3. This was the first time that students were asked to read and reason about Kodu programs in purely textual form rather than tile form, although we had previously asked them to write some rules by filling in blanks with words, and they had seen fragments in textual form on the flash cards. The questions about these programs and their responses are shown in Table 3.

For this problem, students 2, 3, and 4 answered all questions correctly. Student 1 incorrectly chose answer A in Q4, but showed in Q6 that they did understand that there were consequences to incorrect indentation.

## Facilitation of Learning Goals

To become effective Kodu programmers, children must master several abstract ideas. The first is that rules are sequences of a fixed set of tokens with a systematic combinatorial structure. Kodu's visual interface presents these tokens as virtual tiles, and the rule editor prevents the creation of ungrammatical sequences. But the idea of *lawful combinations* may be more effectively conveyed – and at the very least can be reinforced – by giving students a pile of physical tiles that only fit together in certain ways. In the CRA framework cited earlier, physical Kodu tiles are the *concrete* tokens, the virtual tiles are truly *representational*, and what children are creating at the *abstract* level are noun, verb, adjective, and adverb primitives with a rich and largely unformalized semantics.<sup>4</sup>

The second key abstraction children must acquire is the concept of block structure and rule dependency. A full appreciation of this topic requires substantially more instruction and experimentation than the one-hour session of this pilot study, but the use of physical Indent tiles to represent dependency relationships may help students assimilate this abstract idea, perhaps even more so when multiple levels of indentation are introduced.

The third essential abstraction, not addressed in the present work, is finite state machines. The pages of a Kodu program can be viewed as nodes in a state machine diagram, with the "switch to page *n*" actions constituting the transitions [26]. For example, if we want a character to alternately eat red and blue apples, this is most naturally

---

<sup>4</sup> For example, the meaning of "move" is complex and depends on the type of character doing the moving (only some can fly, only some can float), the type of terrain being traversed (land vs. water), and the settings of certain character physics attributes such as acceleration rate.

expressed in Kodu as a two node state machine. It is easy to imagine extensions to the manipulatives toolkit that would facilitate modeling such simple state machines, but we leave this for future work.

## Tangible Programming

Several colleagues' reactions to the tiles were to suggest that they could serve as input tokens to a Kodu interpreter. This is actually an old idea, dating back to Perlman's Slot Machine [20], which used plastic cards bearing Logo motion commands such as "forward 60". These card manipulatives plugged into slots in a reader that controlled a Logo turtle. The idea has been explored many times since, and is now referred to as "tangible interfaces" or "tangible programming". McNerney [14] provides a good review of early work in this area.

Tangible programming systems fall into two classes. In the passive type, the manipulatives are inert tokens that the user arranges and an external mechanism senses and interprets. In systems such as tactusLogic [24] the sensing was done by a video camera, but more commonly it is done by making some mechanical or electrical connection to each token. A current example is the forthcoming open source Primo system [25], a reinvention of the Slot Machine where wooden tokens denoting forward travel, left or right turns, or "subroutine call" are inserted into an interface board to construct motion sequences for a small wheeled robot.

In the active version of tangible programming, the manipulatives are actual computing devices that communicate with their neighbors electrically. An example is the Tangible Programming Bricks of McNerney [13], which used LEGO bricks fitted with embedded microprocessors and several electrical connectors. Each brick was labeled with an operation such as "Measure" or "Multiply", and performed a specialized function such as a reading a sensor value or multiplying a value by a constant. A computational pipeline could be constructed by stacking bricks together.

While tangible programming systems all have something to offer, they provide an extremely limited view of computation. Many lack conditionals, and some, such as E-Block [31], support only straight line code. The active systems more closely resemble circuit design than programming [14], since data flows through the elements but there is no control structure.

Real programming is far richer than straight line code or data flow diagrams. In Kodu, every rule is a conditional, rules execute repeatedly and in parallel, multiple rules on a page interact via priority arbitration and dependency relationships, and multiple pages of rules form a state machine. Furthermore, much of the power of Kodu (and Alice and Scratch) comes from having several characters, each running its own code, interact with each other. This much complexity would be cumbersome to represent with physical tokens, but experience shows that it is not too much for children to master. Thus, the motivation for introducing Kodu tile manipulatives is not to eliminate the

need to look at a computer display and manipulate virtual tiles with the Kodu rule editor, but rather to help scaffold the initial set of mental abstractions that will make children proficient programmers.

## Conclusions

The pilot study demonstrates that with less than 3 hours of exposure, second graders could develop some proficiency with rule syntax, conditional behavior, predicate tests (color filters), implicit iteration, and simple block structure. We cannot credit these results solely to the use of tile manipulatives, as there were many other factors involved, including the explicit teaching of Kodu idioms, the use of flash cards to illustrate these idioms, and the use of pair programming. However, the pilot study does show that tile manipulatives are easily integrated into Kodu instructional sessions and readily used by both children and instructors.

With simple syntax and powerful primitives, Kodu programs can produce a variety of interesting behaviors in just a few lines. Kodu is thus one of the few languages where it is both fruitful and practical to represent executable code with a handful (literally) of plastic tiles. The physical tiles presented here go beyond what is seen on the screen because their shapes enforce syntactic constraints and reify indentation, but they use essentially the same icons as the Kodu GUI, so students have no difficulty mapping between physical and virtual tiles.

There is presently a good deal of interest in, and some controversy over, moving first programming instruction from high school to middle and primary school [18][22]. As we refine our techniques for teaching programming to very young children, we can continue to draw inspiration from mathematics instruction, where the value of manipulatives is clear.

## Acknowledgments

The pilot study was made possible by a gift from Microsoft Research to Carnegie Mellon University. I thank Jason Myers, Director of Technology Integration at Propel Schools, for arranging for the pilot and serving as teacher during the instructional sessions. Thanks also to Muhsin Menekse of the University of Pittsburgh for comments on some of the materials used in the study, and Stephen Coy of Microsoft FUSE Labs for helpful discussions about Kodu semantics.

## References

- [1] Berry, M. "Computing in the national curriculum: A guide for primary teachers." *Computing At School*, 2013. Available online at <http://www.computingatschool.org.uk/primary>.

- [2] Clements, D. H. "Concrete' manipulatives, concrete ideas." *Contemporary Issues in Early Childhood* 1, 1 (1999):45-60.
- [3] Code.org. "3rd Party Educator Resources". <http://code.org/educate/3rdparty>. Accessed June 30, 2014.
- [4] Conway, M., et al. "Alice: Lessons learned from building a 3D system for novices." In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems* (The Hague, The Netherlands: Association for Computing Machinery, 2000): 486-493.
- [5] CS Unplugged. <http://csunplugged.org>. Accessed June 21, 2014.
- [6] Dann, W.P., Cooper, S., and Pausch, R. *Learning to Program with Alice*, 3<sup>rd</sup> edition. (Upper Saddle River: Prentice Hall, 2011).
- [7] ETA hand2mind. "Research on the benefits of manipulatives." [http://www.hand2mind.com/pdf/Benefits\\_of\\_Manipulatives.pdf](http://www.hand2mind.com/pdf/Benefits_of_Manipulatives.pdf). Accessed June 18, 2014.
- [8] Fowler, A., Fristoe, T., and MacLaurin, M. "Kodu Game Lab: A programming environment." *The Computer Games Journal*, 1, 1 (2012):17-28.
- [9] Kelly, J. F. *Kodu for Kids: The Official Guide to Creating Your Own Video Games*. (Indianapolis: Que Publishing, 2013).
- [10] Learning Resources. "Research on the benefits of manipulatives." <http://www.learningresources.com/text/pdf/Mathresearch.pdf>. Accessed June 23, 2014.
- [11] Lifelong Kindergarten Group (MIT). "Blocks". <http://wiki.scratch.mit.edu/wiki/Blocks>. Accessed June 18, 2014.
- [12] MacLaurin, M.B. "The design of Kodu: A tiny visual programming language for children on the Xbox 360." In *Proceedings of the 38<sup>th</sup> Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin: Association for Computing Machinery, 2011): 241-246.
- [13] McNerney, T. S. "Tangible Programming Bricks: An approach to making programming accessible to everyone." Masters thesis, MIT, Cambridge, MA, 2000. Available at <http://www.media.mit.edu/people/mc/tangible-programming.html>.
- [14] McNerney, T. S. "From turtles to Tangible Programming Bricks: explorations in physical language design." *Personal and Ubiquitous Computing*, 8, 5 (2004):326-337.
- [15] Manches, A., and O'Malley, C. "Tangibles for learning: a representational analysis of physical manipulation." *Personal and Ubiquitous Computing*, 16, 4 (2012): 405-419.
- [16] National Council of Teachers of Mathematics. "Illuminations: Resources for Teaching Math." <http://illuminations.nctm.org>. Accessed June 26, 2014.
- [17] National Council of Supervisors of Mathematics. "Improving student achievement in mathematics by using manipulatives with classroom instruction." NSCM Improving Student Achievement Series, No. 11, Spring 2013. Available at <http://www.mathedleadership.org/member/docs/resources/positionpapers/NCSMPositionPaper11.pdf>.
- [18] The New York Times. "Computing in the classroom." The Opinion Pages, Room for Debate, May 12, 2014. Available online at <http://www.nytimes.com/roomfordebate/2014/05/12/teaching-code-in-the-classroom>.

- [19] Olson, P. "Why Estonia has started teaching its first-graders to code." *Forbes*, September 6, 2012. Retrieved March 4, 2013, from <http://www.forbes.com/sites/parmyolson/2012/09/06/why-estonia-has-started-teaching-its-first-graders-to-code/>.
- [20] Perlman, R. "Using computer technology to provide a creative learning environment for preschool children." Technical report, MIT Logo Memo #24, 1976. Cited in [14].
- [21] Resnick, M., et al. "Scratch: Programming for everyone." *Communications of the ACM* 52, 11 (2009):60-67.
- [22] Richtel, M. "Reading, writing, arithmetic, and lately, coding." *The New York Times*, May 11, 2014, page A1.
- [23] Rusk, N. "Scratch cards." <http://scratched.gse.harvard.edu/resources/scratch-cards>. June, 2009.
- [24] Smith, A. C., et al. "tactusLogic: Programming using physical objects." IST-Africa Conference Proceedings (Gaborone: IEEE, 2011): 1-9.
- [25] Solid Labs. "Primo." <http://www.primo.io/home>. Accessed August 16, 2014.
- [26] Stolee, K.T., and Fristoe, T. "Expressing computer science concepts through Kodu Game Lab." In *Proceedings of the 42<sup>nd</sup> ACM Technical Symposium on Computer Science Education*, (Dallas: Association for Computing Machinery, 2011): 99-104.
- [27] Tiger Leap Foundation. "Programming at schools and hobby clubs." <http://www.tiigrihype.ee/en/programming-schools-and-hobby-clubs>. Accessed December 5, 2013.
- [28] Touretzky, D.S., et al. "Accelerating K-12 computational thinking using scaffolding, staging, and abstraction." In *Proceedings of the Forty-Fourth ACM Technical Symposium on Computer Science Education* (Denver: Association for Computing Machinery, 2013): 609-614.
- [29] Touretzky, D. S. "Kodu resources." <http://www.cs.cmu.edu/~dst/Kodu>. Accessed September 18, 2014.
- [30] Utah State University. "National Library of Virtual Manipulatives." <http://nlvm.usu.edu/en/nav/vlibrary.html>. Accessed June 21, 2014.
- [31] Wang, D., Zhang, Y., and Chen, S. "E-Block: A tangible programming tool with graphical blocks." *Mathematical Problems in Engineering*, 2013 (2013). <http://dx.doi.org/10.1155/2013/598547>.
- [32] Zorfass, J., Brann, A., and PowerUp WHAT WORKS. "Interacting with peers in mathematics." <http://www.ldonline.org/article/61470/>. Accessed June 25, 2014.

**Table 1:** Quiz questions referencing the code in Figure 6, and student responses.

Question	Student 1	Student 2	Student 3	Student4
<b>Q1. When will the kodu play the coin sound?</b>	“All of the time.”	“When it moves toward it.”	“when move toward”	“when it eats the apple”
Q2. After the kodu eats a ball, it has to wait for the cycle to throw out a new ball. While it’s waiting, will the Kodu play a coin sound?	Yes	No	No	Yes
Q3. Explain your answer.	“Because there is a indent.”	“no because it will not move toward the ball.”	“because you don’t want them to.”	“because when it eats it it will play the sound” (drew arrow on diagram)

**Table 2:** Alternative programs used in the session 3 quiz.

<b>Answer A</b>	<b>Answer B</b>
[1] WHEN see fish DO move toward [2] WHEN bump fish DO grab it [3] WHEN DO score red 1 point	[1] WHEN see fish DO move toward [2] WHEN bump fish DO grab it → [3] WHEN DO score red 1 point

**Table 3:** Questions and responses about the code shown in Table 2.

<b>Question</b>	<b>Student 1</b>	<b>Student 2</b>	<b>Student 3</b>	<b>Student 4</b>
Q4. Suppose the kodu is pursuing fish. Every time it grabs a fish, it wants to count it. What rules should it use to grab and count fish: Answer A, or Answer B?	A	B	B	B
Q5. Why is your answer the correct choice?	“Because it chased the fish.”	“if I chose A the score will go really fast”	“Because it has an indent”	“because it is surpost [ <i>sic</i> ] to have an indent”
Q6. What would happen if we used the other set of rules instead?	“It would keep scoring points.”	“the score would go really fast”	“It wouldn’t do what you want it to do.”	“it will count a lot of fish.”

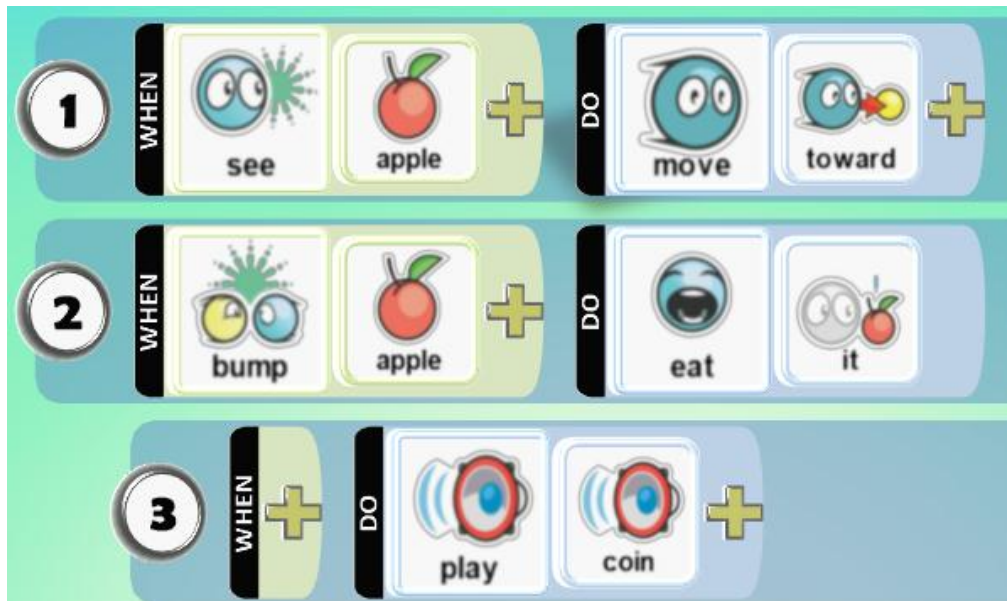


Figure 1: A program for eating apples using the “Pursue and Consume” idiom, displayed in the Kodu rule editor.



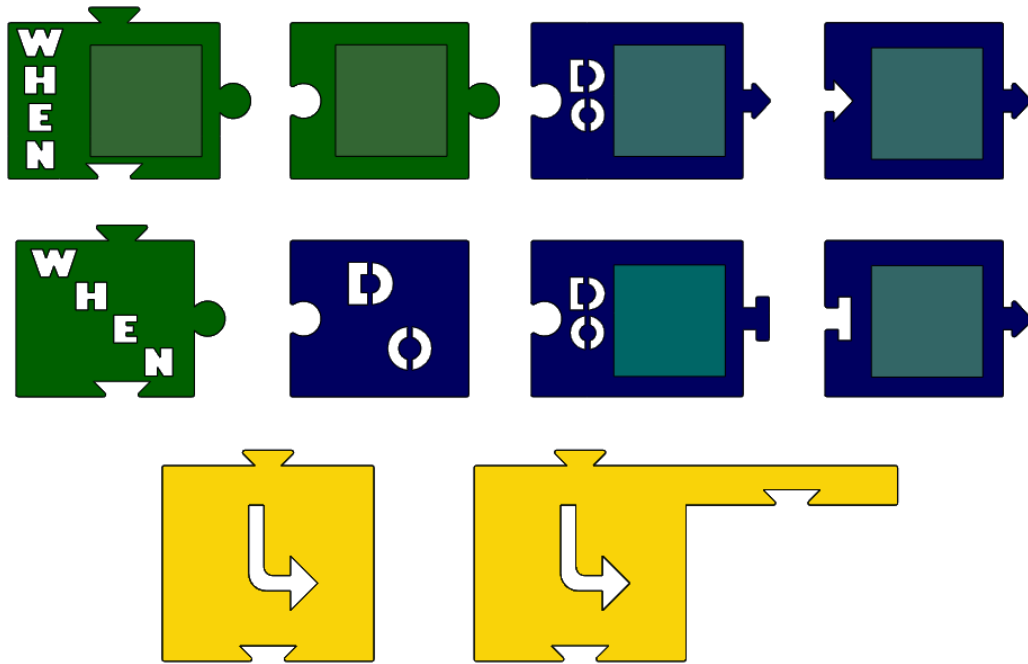


Figure 2: The ten tile shapes comprising the Kodu manipulatives design. Top row: WHEN-Predicate, Predicate-Argument, DO-Action, and Action-Argument tiles. Middle row: Empty-WHEN, Empty-DO, DO-Create, and Create-Argument tiles. Bottom row: More-Indent and Indent tiles.

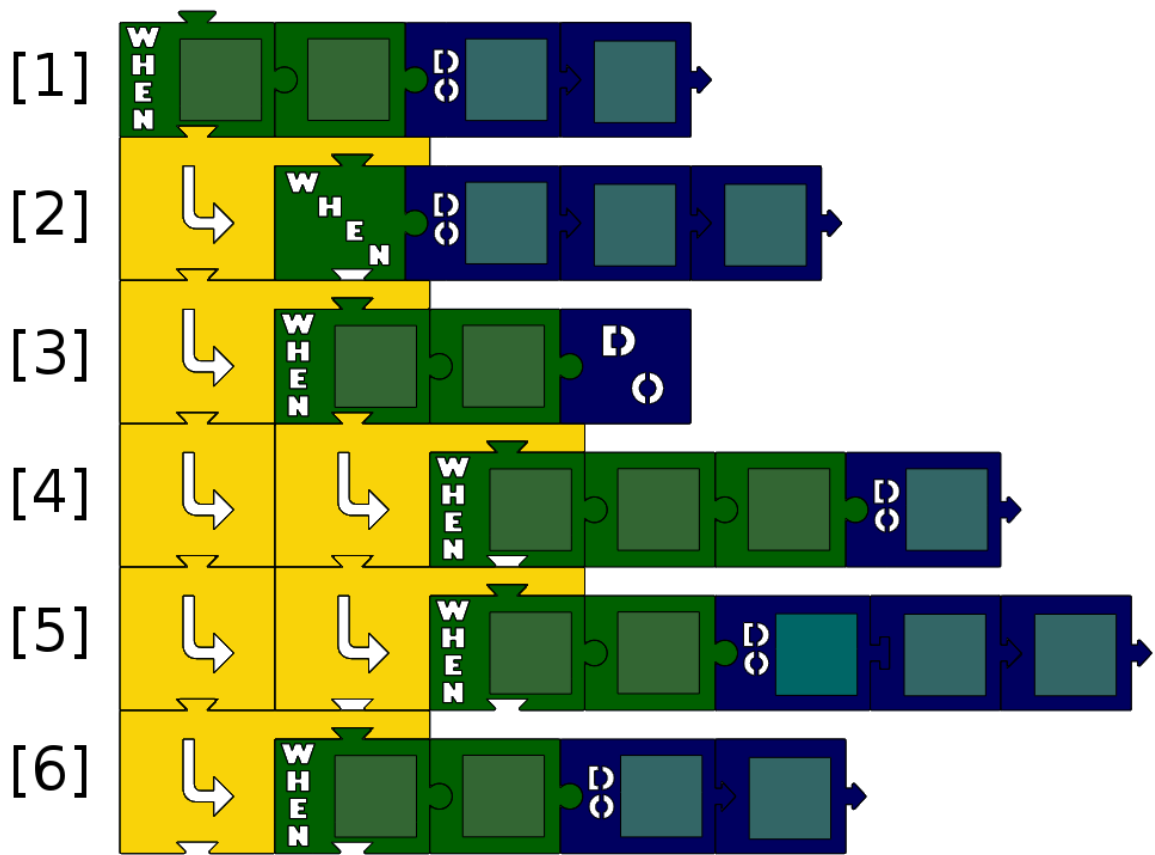


Figure 3: A schematic page of Kodu rules using all ten tile shapes.

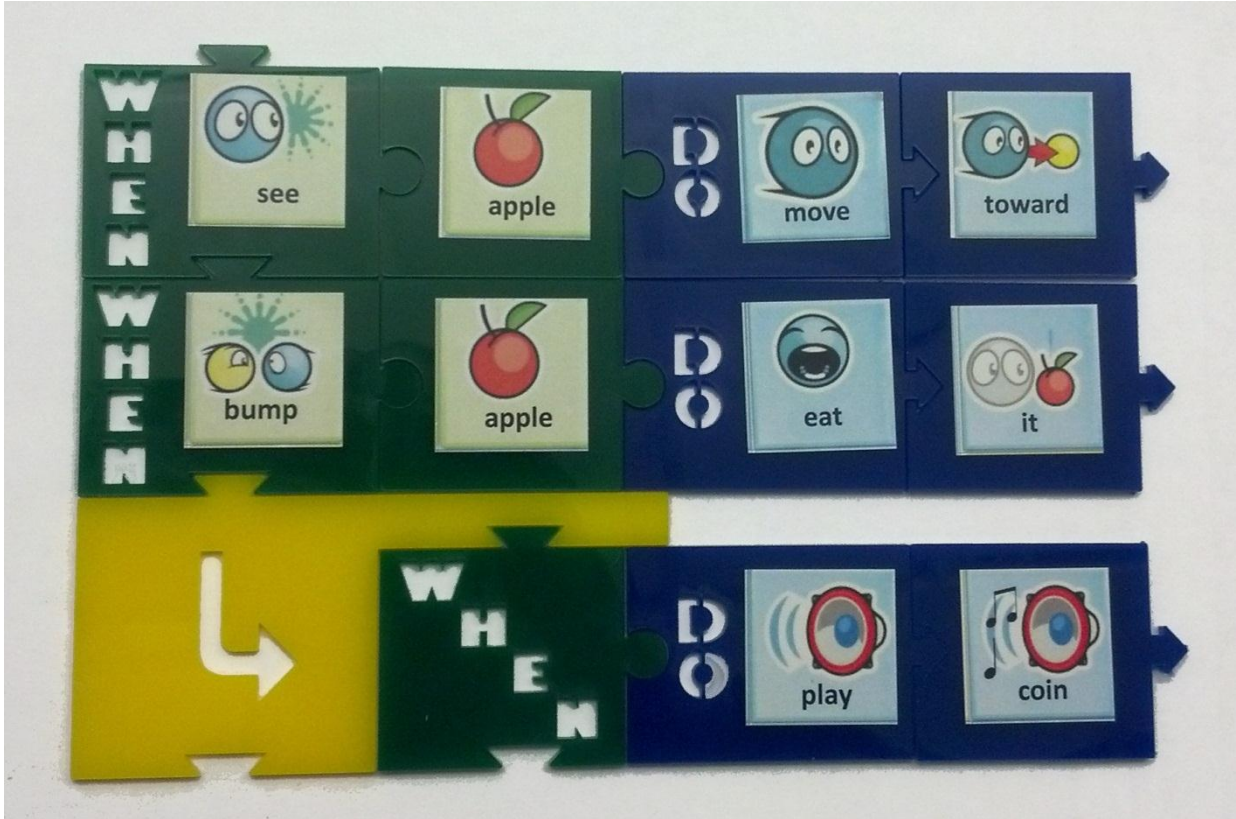




Figure 4: The physical realization of the apple eating program from Figure 1.


### Do Two Things

Make the Kodu take two actions with one rule.

WHEN *something* ... DO **this** 

↳ *and also* → DO **that** 

### Do Two Things



General Form:  
 WHEN *something* DO *action1*  
 → WHEN DO *action2*

Indenting the second rule makes it dependent on the WHEN part of the rule above.

Figure 5: The “Do Two Things” flash card, one of three such cards that were supplied to the students. The card is designed to be folded in half at the dashed line and then laminated.

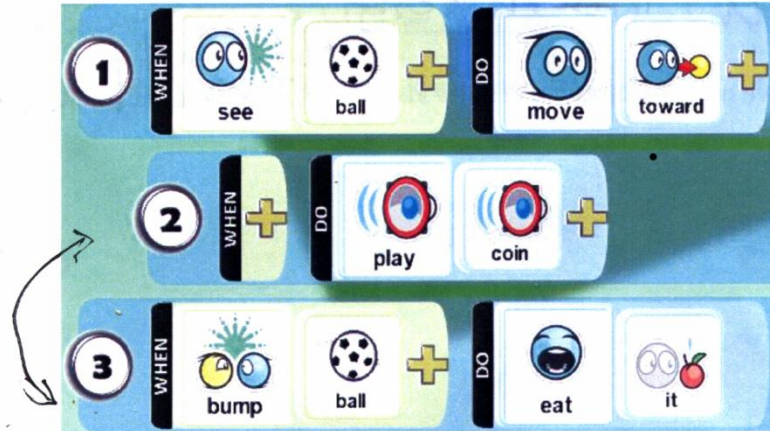


Figure 6: A buggy program used to test students' understanding of rule indentation. The arrow connecting rules 2 and 3 was drawn in by a student and is discussed in the main text.