Tema

PROGRAMMING IN MOTION

Es ist nicht einfach, das im Lehrplan21 verankerte Schreiben von Programmen im zweiten Zyklus anders als oberflächlich zu vermitteln. Ein Grund liegt mitunter darin, dass die grundlegenden syntaktischen Elemente der Programmiersprache für Programmierer offensichtlich sind, nicht aber für Kinder. Es braucht mehr als Erklärung, um die Logik dahinter zu erkennen - es braucht Übung. Ein Ansatz kann darin liegen, Code-Elemente physisch darzustellen und so jene Vertrautheit in deren Umgang zu erlangen, die für komplexeres Programmieren unerlässlich ist. Der folgende Beitrag gibt eine Übersicht zu den verschiedenen Möglichkeiten, mit denen sich Codes durch Bewegung lernen lassen und sich gleichsam die theoretischen Grundlagen in anderen Bereichen erschliessen (Sprache und Mathematik).

Patrick Büchel | Zürich

Patrick Büchel is a primary teacher at Schule Auhof in Zurich.

The Swiss second cycle (upper primary) curricular aim of writing programs is hard to meet in more than just a superficial way. Part of the reason is that the basic syntactic elements of programming languages seem obvious to programmers but not to children. Seeing the logic behind it takes more than just explaining, it takes practice. A way of bridging this gap might be the physical acting out of code elements used to drill functional elements in order to gain the familiarity needed for more complex programming. The paper gives an overview of different ways of code enacting and looks for parallels to theoretical foundations in other areas (language and math).

The Swiss curriculum

The regional curriculum (LP21) provides aims for both foreign languages (here English) and programming (Medien und Informatik). The skills hopefully acquired through the following activities are described under having learners be able to model processes using loops, conditions and parameters and finally write functional code. In foreign languages, learners are expected to reach the A2 level by the end of primary school (reading, speaking and listening) and the language involved in programming (primarily English outside the classroom) is extremely functional in real-world, A2 level communicative situations. Thus, the context of programming in English can be seen in a CLIL context and the activities below lend themselves to an embodied approach to language learning.

I usually start the introduction to programming with simple motion commands steering some sprite (in the old days it was a turtle in Scratch) to draw a square:

Move, turn right, move, turn right, move, turn right, move, turn right

Fourth or fifth grade children (9-11 year olds) will immediately understand this, especially if acted out. The explicit form of “turn right” is “turn right by 90°”. At this stage however, this is what the children will intuitively understand. As a next step, the tedious repetition can be handed over to a loop:

```
Repeat 4 times:
        Move
        Turn right
```

This step also is teachable within the first

Thus, you have here an example of seemingly easy and familiar language (writing and reading if/then/repeat/ move) but the true comprehension of which is not always obvious to the learners!

few lessons as children are familiar with simple repetitions. For acting it out, a second child can play the loop, counting the cycles and ordering the cat to have another go.
The next step of complexity is to program a polygon:

```
Repeat n times:
        Move
        Turn right by 360° / n
```

Astonishingly, it is not so much the math of calculating the angle but more the introduction of a variable (or a parameter, as the curriculum puts it) which makes this step hard for many students. But we can go even further, introducing recursion:

```
To draw a recursive polygon (n):
        Move
        Turn right by 360° / n
        If (n > 2) then:
                Draw a recursive polygon (n-1)
```

Adding just a few lines has contributed to the concepts of procedures, logical or Boolean operators, conditions and recursive calls as well as a complexity impossible to grasp for most in the class. This is not a program you can enact either – it would take hours and the actors would inevitably get confused and lose control. The resulting drawing however makes the process a bit more understandable:
For a teacher who was so proud of his fourth graders after they wrote their first square drawing program within a lesson, this is frustrating. When the power of complexity starts getting interesting you lose most students, and the gap to close seems huge. A few lines which are quite basic to a seasoned programmer seem almost unteachable. What is the problem? The problem, of course, is that seasoned programmers think that the functional details and their interaction are self-evident, "logical". Explain the working of a loop, a condition and a variable and everything should be clear, right? We know from math teaching (and also from language teaching) that this is not the case. 2 * 4 is easy but (2 * (3 + (6 * 2))) is not. You can solve it step by step, but in order to find an algorithm for a complex problem, your focus needs to go beyond the single steps. As in every complex action (e.g. walking), the basic parts have to be automatized. What programmers call "logical" should better be called "familiar" - like bending your foot the right angle after each step. In order to understand the interaction of a condition (an if statement) and a Boolean operation each has to be automatized, and to understand a condition within a larger algorithm, this interaction itself has to be automatized. Just like in math, part of acquiring this familiarity

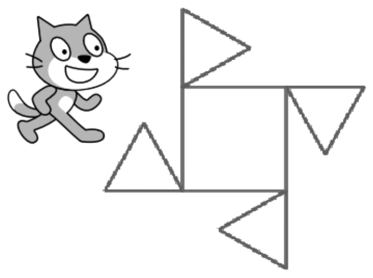


Figure 1: Recursive polygon (4)

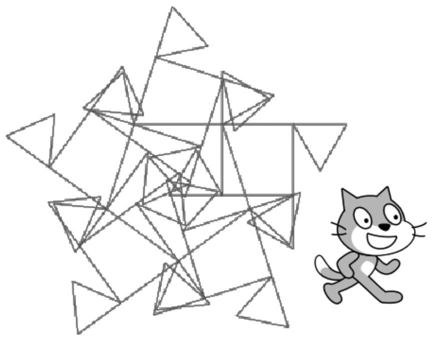


Figure 2: Recursive polygon (5)

(Those who will try to code this in Scratch will find that I made a few changes to make the graphics more intuitive.)

which goes beyond understanding it in principle is switching between different forms of representation, i.e. writing code, analyzing code, running code (seeing it in action, seeing its results) and acting out code.
Thus, you have here an example of seemingly easy and familiar language (writing and reading if/then/repeat/move) but the true comprehension of which is not always obvious to the learners!

Ways of acting out code

In language teaching, acting out seems obvious. In math or IT teaching, it comes in just as handy and does about the same thing. It shifts from language to another form of representation which (just like music) often allows for more complexity to be grasped at once. As opposed to when you explain a piece of code, when you act it out you can do so at an increasing speed which forces students to internalize the code parts, enabling the learner to use them on a higher level. Accelerating acting out (e.g. in a competition) will force students to control "the logic" behind it without full executive control – to automatize it.

Simulation

When a child acts like the cat drawing a square, he or she is merely a convenient simulation of the code. The advantages over making Scratch do this is that the acting child has to demonstrate some comprehension and the peers observing will mostly do the same in empathy (or in order to detect mistakes). It also allows for a slow, flexible execution the teacher can comment on at will.

Class drill

In a similar manner, in an entire class, each student individually can act out a small element of code, e.g. a condition:
IF I raise my hand THEN you clap your hands ELSE (if I stick my hand any other direction) you snap your fingers.
Here your goal is to demonstrate the logic, assess understanding, and drill, all in one go. As in language repetition drills, you need to flexibly alter your commands and sometimes single out students who you observe lacking understanding or who you suspect of just copying the others.

Program execution

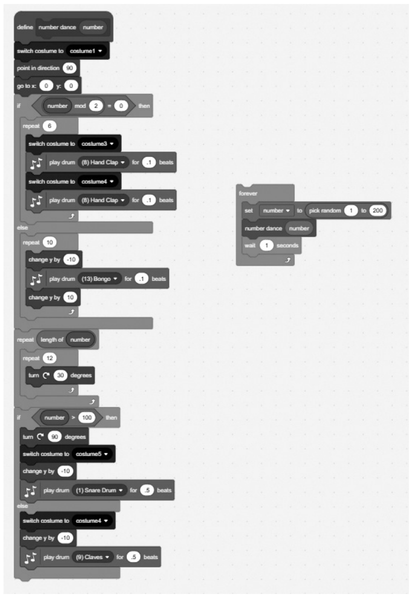
Instead of just simulating a command-driven drawing cat (where syntax understanding is of minimal importance), single students can perform a parameterized program of some complexity (cf. the example of a number-analyzing dancing program below). The goal is that students become fluent in analyzing code. As with the cat simulation, peers will closely observe the student acting out. If you carefully choose the sequence of students acting out and the order of parameter values, you can control the learning curve of the entire class. Once you let classmates choose values for their acting peers, they will try to predict program execution, an important programmer skill.

Program parts as theater roles

```
The number dance (number)
    If (number is even)
        Clap your hands
    Else
        Stomp your feet

    Repeat for (length of the number)
        Do one turn of a pirouette

    If (number is bigger than 100)
        Land on your belly
    Else
        Land on your bum
```



```
define number dance (number)
switch costume to costume1
point in direction 90
go to x: 0 y: 0
if (number mod 2 = 0) then
    repeat 6
        switch costume to costume3
        play drum (8) Hand Clap for .1 beats
        switch costume to costume4
        play drum (8) Hand Clap for .1 beats
else
    repeat 10
        change y by -10
        play drum (13) Bongo for .1 beats
        change y by 10
repeat (length of (number))
    repeat 12
        turn ↻ 30 degrees
if (number > 100) then
    turn ↻ 90 degrees
    switch costume to costume5
    change y by -10
    play drum (1) Snare Drum for .5 beats
else
    switch costume to costume4
    change y by -10
    play drum (9) Claves for .5 beats

forever
    set number to (pick random 1 to 200)
    number dance (number)
    wait 1 seconds
```

Figure 3: The number dance for students (left) and in Scratch (right)

As in language repetition drills, you need to flexibly alter your commands and sometimes single out students who you observe lacking understanding or who you suspect of just copying the others.

A group of students can also each act out a part of a larger program and thus together run it in interaction. Someone will play a loop, someone the Boolean operator within, someone a variable and someone else the drawing cat receiving orders, let's say from the loop. As any teacher who has put together a theater production with his or her class will tell you, one main challenge is to manage all the actors who are not currently saying their line into nevertheless participating in some way or at least keeping track of the play's progress in order not to miss their line. Very much the same holds true for acting out code. In addition and in contrast to theater acting, it is anything but obvious to define the roles. In my class, we had the following discussion in English:

Is "move 10 steps, turn 90°" one role, the cat who knows what to do when? Or is it two roles, the cat commander and the cat. Can the cat commander be the loop who at the same time has to count the number of loop executions or do we need to have a loop, a cat commander and a cat? Or do we even need a loop counter (some kind of variable), a loop (saying something like "one more time!", a cat commander specifying what exactly to do and the cat actually doing it?

If you carry this kind of discussion with your students, it might lead them to analyze code execution more closely. For beginners and with initial understanding and drilling in mind, this kind of acting out – as fun as it sounds – carries a high risk for confusion (the funniest if not most educational part being when the directing teacher starts getting confused).

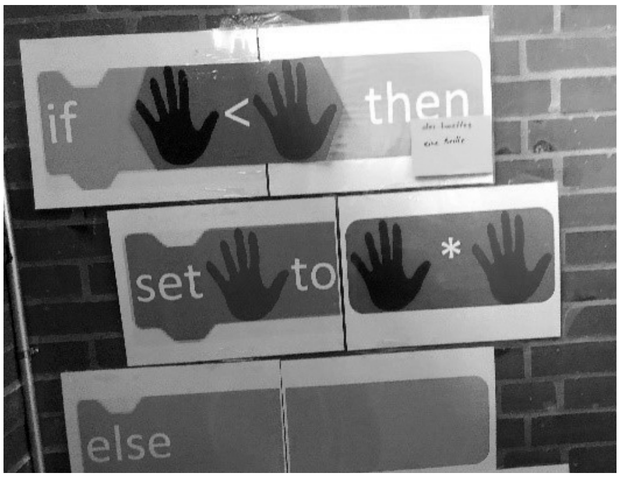
Learning environment

Finally, you have what I like to call a code-elements learning environment. You need a preferably large space (classroom, gym, outside area) with various visual code pieces, each with relatively simple rules for their correct use by a group of students. As variables play a powerful role as parameters or in the storage and manipulation of information, the students enter the space as variables. Variables can receive an initial value. They can have their value changed and they need at all times to remember their current value and tell it to anyone asking (before receiving their initial value they actually don't know their value and will say so if asked):

```
Set x to 1              // initial value
Set x to (x + 1)        // value change
```

This is a special example of value change in which the variable assumes at once the role of receiving a new value and of a parameter offering its previous value.

Visual code pieces



The following visual pieces are used (I am using ❑ where a variable can be inserted to receive a new value or to give its value as a parameter):

Set a variable's value:		
Set ☐ to 1 Give a variable an initial value.	**Set ☐ to (☐ + ☐)** Give a variable a value calculated by operation from other variables.	**Set ☐ to (☐ + 1)** Give a variable a value calculated by operation from a variable and a value.
Math operators		
(☐ + ☐) Addition	**(☐ - ☐)** Subtraction	**(☐ * ☐)** Multiplication
Loops		
Repeat ☐ Set ☐ to (☐ + 1) Repeat a number of times.	**Repeat until (☐ > ☐)** Set ☐ to (☐ + 1) Repeat as long as a logical operator remains false.	
Logical operators		
(☐ > ☐) bigger than	**(☐ < ☐)** smaller than	
Conditions		
if (☐ > ☐) then Set ☐ to (☐ + 1) **else** Set ☐ to (☐ * 2)	If a logical operator is true, do something else do something else	

Notice that this is a small choice of the existing Scratch commands for the sole purposes of setting variable values and controlling program flow. And even from that subset anything too complicated (division) or too boring (Boolean = operator) has been omitted.

Usage

Elements are introduced one by one, building up complexity.

- Set with value
- Set with operator
- If condition with logical operator
- Repeat a number of times
- Replacement of "Repeat a number of times" by "Repeat until"

It is a good idea to start challenges after the introduction of "sets" and conditions. Loops require some familiarity with the simpler concepts first. Here are some competition task examples, also in order of increasing complexity:

In teams, single values as goals

Values in a sequence	(each member aims for one of the values: 10, 11, 12, 13, 14)
The biggest the quickest	(reach the greatest sum of all member values quicker than the other teams) *Strategy: Find the most powerful multipliers, rotate among members.*
All 7	(all members aim for the same value) *Strategy: How does the last member become 7? E.g. become 14 and subtract by 7.*
One big, others small	(one member needs to become as big as possible while the others remain as close to 0 as possible)

Everybody (social challenge)

My birth month	(everybody tries to reach his or her month of birth, participants are allowed to murmur: "I'm 3, I want to be 7" *Strategy: Help others! Be transparent!*

In teams, value sequences as goals

Multiplication table	(each member has to become the numbers of a multiplication table sequence: 2, 4, 6, 8 ….; 3, 6, 9, 12 …; 4, 8, 12, 16 …) *Strategy: What value do others need to have in order to help me reach my values? Can I use mathematical relationships between sequences (e.g. 2 = 2 * 4)?*

General strategy: Discuss your goals in your team before trying out.

These examples show how programming can be carried out entirely in English, with useful English embedded throughout the lessons (if…then…else…find…set….).

Discussion of the "learning environment"

The second part of this article discusses what I call the "learning environment" (because there is space for moment around rich concepts) which I tried out for 1-3 lessons (for each topic) of two classes of 5th graders in the canton of Zurich in settings of full (18 students) and half class (10 students) as well as with adults with a language teaching background in a workshop. This is very meager data for conclusions, so let's just call it first impressions.

First of all, most children are not very motivated to use the material. The ones motivated are the ones that already show a good understanding of the mechanisms. What motivates them are the strategical challenges (partially also the prospect of winning), not the drill. Drilling situations (quick and frequent use) are rare and only occur with the simple "set" elements.

Furthermore, as opposed to – especially older learners – adults, children will accept a short introduction and mostly understand a quick demonstration of the elements. Corrections are made among peers or by the instructor as they try it out. Adults

Findings from language, math and programming suggest that motion plays an important part in building, understanding and dealing with abstract concepts.

refuse to try before fully understanding the how and also the why.

From this first impression, one could conclude that the intended effect of the drilling does not occur as quickly as one might have expected. In this instance, it could be that the pace of introduction was too fast, and we shifted to tasks needing too many strategic considerations too early. Also, in the tight classroom environment, the motivating factor of really moving (running and screaming) was missing – doing it in the gym would have been better!

It can also be asked if the motion and sensory input in this learning environment fit the internal representations we want to build and strengthen (or the areas in our brain we want to trigger). No studies were found on brain areas affected by programming but based on questionnaires, Petre (1999) does find that programmers use mental imagery different from the actual form of implementation (code). These descriptions of seasoned programmers are however highly individual and on too high a level of abstraction to point to brain areas involved.

More promising are Núñez' (2008) efforts to show the foundations of mathematics in embodiment. To prove his point, he analyzes language and gestures used by mathematicians. He finds movement (in the sense of "to go somewhere") to be a common way of not only expressing, but also thinking of, static mathematical concepts. One of Núñez' examples is an equation in the form of a function. Considering many programming languages call their pieces of algorithm "functions" and considering programming algorithms are always some action in time, this seems like a good fit.

Another well researched domain is language. Besides obvious action words like move or turn, programming languages use what linguists call function words as opposed to content words or lexical words. Content words refer to things, their properties and actions, whereas function words modify content words or express relations between them. Programming language examples would be *if … then … else*, *not*, *and*, *or*, *until*, *while*. Repeat, used for loops, at first seems like an action verb, i.e. a content word. However, without specifying what to repeat, it does not command any action, so it is actually a modifier (specifically a multiplier) for action verbs, i.e. a function word. Building on research showing that words referring to action like *kick* activate the same neurons the action itself does but also words related to actions like words for tools for actions do so (cf. overview in Pulvermüller 2005), Gosselke (2010) suggests that function words have roots in content words which are weakened and complemented by connections to the language areas, not lost during grammaticalization (Gosselke 2005: 49). Gosselke shows this grammaticalization process with the word *going to*, used in several languages to express future tense.

Back to programming: Eckerdal & Berglund (2005) interviewed engineering students in their first year about learning object-oriented programming, a part of their curriculum. They found that the students who got into programming more easily could switch between an object and a process-understanding of concepts. Let's take as an example an ordered list. The static object would be a list, where for any pair x_i and x_{i+1}, we would have x_i is smaller than x_{i+1}. The process description would be a sorting algorithm, e.g. to repeatedly loop through the list and switch any pair where the first item is larger than the second. The first, abstract view is necessary to think of a larger program where the sorted list is just a detail, whereas the process view assures the concept is understood in detail (this reminds me of the teacher's operationalizing of learning aims). Eckerdal & Berglund base their findings on Hazzan (2003) who comes to a similar conclusion and Sfard (1991) who postulates such a duality in mathematics, using similar examples as Núñez.

I have come to the conclusion that the very tools I am using (Scratch and Snap) in and of themselves are designed to support an embodied learning environment.

Maybe: Concluding thoughts

Findings from language, math and programming suggest that motion plays an important part in building, understanding and dealing with abstract concepts. Using motion to strengthen the understanding of code elements may be a good idea. However, if most programming concepts are based on motion, we can't just always walk around to strengthen their understanding. There has to be some differentiation like a typical motion for *repeat* (I would imagine some circular motion) and another one for *if … then … else* (maybe some forking trail).

Let's close the loop with someone who could with some right be called the grandfather of Scratch. Seymour Papert, greatly influenced by Piaget, used the LOGO programming language to empower children to construct their own understanding of abstract concepts (Papert 1980, 1993). LOGO was not yet a block or even visual language like Scratch where children can graphically connect code elements. LOGO was text-based, but with simple elements, and it featured "turtle graphics", a graphically less fancy way of doing exactly what my students do with the square-drawing cat (for a feel of turtle graphics, try Snap, a more complex and extendible successor of Scratch, www.snap.berkeley.edu). What Papert intended was not mainly that children learn to program, but that they learn the thinking behind programming and that their general cognitive (including mathematical and linguistic among others) skills would be improved through programming, and Scratch and Snap, used in today's classrooms, continue Papert's legacy. The activities described in this article were an attempt to replace mere verbal explanations of code by having learners physically experience code before using it in a more complex way on the computer. Because I wanted to understand why drilling in the "learning environment" didn't work as well as I had intended, I delved deeper into embodied representations of what I was trying to teach. The articles referenced in this text, describe how various concepts have their roots in embodiment. Finally, I have come to the conclusion that the very tools I am using (Scratch and Snap) in and of themselves are designed to support an embodied learning environment.

Literature

Eckerdal, A., Thuné, M., & Berglund, A. (2005, October). What does it take to learn 'programming thinking'?. In *Proceedings of the first international workshop on Computing education research* (pp. 135-142). ACM.A. Eckerdal & A. Berglund 2005. What Does It Take to Learn 'Programming Thinking'?

Gosselke, S. (2011). The Neural Representations of Function Words: Neurolinguistic Beliefs Reconsidered in the Light of Grammaticalisation Theory. S. Gosselke 2010. The Neural Representations of Function Words. Neurolinguistic Beliefs Reconsidered in the Light of Grammaticalisation Theory.

Hazzan, O. (2003). How students attempt to reduce abstraction in the learning of mathematics and in the learning of computer science. *Computer Science Education*, 13(2), 95-122.

Horn, M. S., & Jacob, R. J. (2007, February). Designing tangible programming languages for classroom use. In *Proceedings of the 1st international conference on Tangible and embedded interaction* (pp. 159-162). ACM.: Designing Tangible Programming Languages for Classroom Use

Núñez, R. (2008). A fresh look at the foundations of mathematics: Gesture and the psychological reality of conceptual metaphor. *Metaphor and gesture*, 93-114.R. Núñez: A Fresh Look at the Foundations of Mathematics: Gesture and the Psychological Reality of Conceptual Metaphor

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc..

Papert, S. (1993). **The children's machine: Rethinking school in the age of the computer.** BasicBooks, 10 East 53rd St., New York, NY 10022-5299. S. Papert 1993. The Children's Machine

Petre, M., & Blackwell, A. F. (1999). Mental imagery in program design and visual programming. *International Journal of Human-Computer Studies*, 51(1), 7-30.

D-EDK/Deutschschweizer Erziehungsdirektoren-Konferenz 2010–2014, *Lehrplan 21*, D-EDK, Luzern.

Pulvermüller, F. (2005). Brain mechanisms linking language and action. Nature reviews neuroscience, 6(7), 576.F. Pulvermüller 2005. Brain mechanisms linking language and action.

Sfard, A. (1991). On the dual nature of mathematical conceptions: Reflections on processes and objects as different sides of the same coin. *Educational studies in mathematics*, 22(1), 1-36. A. Sfard 1991. On the Dual Nature of Mathematical Conceptions: Reflections on Processes and Objects as Different >>Sides of the Same Coin.