

This is a self-archived version of an original article. This version may differ from the original in pagination and typographic details.

Author(s): Fagerlund, Janne; Vesisenaho, Mikko; Häkkinen, Päivi

Title: Fourth grade students' computational thinking in pair programming with scratch : A holistic case analysis

Year: 2022

Version: Accepted version (Final draft)

Copyright: © 2022 Published by Elsevier B.V.

Rights: CC BY-NC-ND 4.0

Rights url: <https://creativecommons.org/licenses/by-nc-nd/4.0/>

Please cite the original version:

Fagerlund, J., Vesisenaho, M., & Häkkinen, P. (2022). Fourth grade students' computational thinking in pair programming with scratch : A holistic case analysis. *International Journal of Child-Computer Interaction*, 33, Article 100511. <https://doi.org/10.1016/j.ijcci.2022.100511>

Fourth Grade Students' Computational Thinking in Pair Programming with Scratch: A Holistic Case Analysis

Janne Fagerlund^{ab*}, Mikko Vesisenaho^a, Päivi Häkkinen^b

^a*Department of Teacher Education, University of Jyväskylä, Jyväskylä, Finland;* ^b*Finnish Institute for Educational Research, University of Jyväskylä, Jyväskylä, Finland*

*Postal address: Ruusuipuisto, P.O. Box 35, 40014 University of Jyväskylä, Finland. *E-mail: janne.fagerlund@jyu.fi *Tel. +358408054711

Dr. Janne Fagerlund is a post-doctoral researcher at the Finnish Institute for Educational Research, University of Jyväskylä, Finland. He is currently working for the International Computer and Information Literacy Study (ICILS). He recently defended his doctoral dissertation, which focuses on computational thinking in graphical programming with Scratch at the primary school level. He also operates as a regional coordinator in the Innokas Network (<http://innokas.fi/en>). ORCID: <https://orcid.org/0000-0002-0717-5562> LinkedIn: <https://www.linkedin.com/in/jannefagerlund/>

Dr. Mikko Vesisenaho is an adjunct professor, and a senior lecturer at the Department of Teacher Education, University of Jyväskylä. His background is in education, contextual design and computer science education. He has 20 years' experience in multi-disciplinary education and research with national and international collaborators. His ambition is to innovatively reform learning with technology. ORCID: <http://orcid.org/0000-0003-1160-139X> Postal address: Ruusuipuisto, P.O. Box 35, 40014 University of Jyväskylä, Finland. E-mail: mikko.vesisenaho@jyu.fi Tel. +358400247686

Päivi Häkkinen is a Professor of educational technology and a vice-director at the Finnish Institute for Educational Research, University of Jyväskylä. Her research focuses on technology-enhanced learning, computer-supported collaborative learning and the progression of twenty-first-century skills (i.e., skills for problem solving and collaboration). ORCID: <https://orcid.org/0000-0001-6616-9114> LinkedIn: <https://linkedin.com/in/paivi-hakkinen-59132612> Postal address: Ruusuipuisto, P.O. Box 35, 40014 University of Jyväskylä, Finland. E-mail: paivi.hakkinen@jyu.fi Tel. +358405843325

Fourth Grade Students' Computational Thinking in Pair Programming with Scratch: A Holistic Case Analysis

Abstract

This article explores a new but expanding research topic: primary school students' computational thinking (CT) in the context of programming in pairs. The data comprises four fourth-grade student dyads using Scratch, a block-based programming tool, for two open-ended creative programming sessions. We sought insight into how the dyads put four intertwined CT dimensions—planning; iteration; collaboration, social interactions, and remixing; and debugging—into practice. To examine these dimensions, the data was analysed systematically in three overlapping layers: what happens on the computer screen (*design events*), who uses the computer (*computer control*), and what kind of talk is occurring (*talk*). The temporal viewpoint in the multi-layered analysis revealed and enriched holistic understanding of key computational and social factors, such as initial project planning methods, programming tendencies, and pair programming roles that essentially shaped the dyads' design processes. Next to opportunities for more focal research in CT education, the results provide especially evidence-based pedagogical knowledge for supporting students' open-ended programming in the classroom. In particular, concrete suggestions for supporting open-ended project planning, balancing between self-directed design and instructional support during programming, and promoting shared design processes in pair programming are provided based on the findings.

Keywords: computational thinking; Scratch; pair programming; primary education

1 Introduction

Programming by designing interactive media, robotics, and digital fabrication is increasingly promoted in schools across the world. One goal of these initiatives is to foster students' computational thinking (CT), which has several known educational benefits, including understanding the computational world and gaining skills in applying computational tools, methods, and models to solve real-life problems in different contexts [1]. CT is perceived to encompass skills and understanding in such cross-contextual concepts and practices as 'algorithms', 'data', and 'debugging' [2, 3, 4]. Pair programming, in which two programmers work together to achieve a common goal, has been emphasised as an efficient social constructionist way to engage students in learning such concepts and practices [5] as well as a pathway for learning collaboration—another key dimension in CT—for potentially improving the quality of the problem-solving process and learning [4].

In this expanding topic, educational scholars and practitioners continue searching for evidence-based practical ways to meaningfully assess and support students' learning of CT [6]. Examining students' CT-fostering programming activities, sometimes labelled distinctly as 'CT practices', is critical, along with analysing programmed artefacts [7], interviewing [8], conducting questionnaires [9], and testing [10]. The primary purposes of such examinations include attaining rich contextual insight into how CT is learnt while designing computational artefacts [6]. However, several significant issues in students' collaborative CT learning processes in Scratch—an especially popular programming tool—in the classroom are inadequately understood. These include students' programming tendencies, challenges they can face, potential effects of different kinds of challenges on artefact-related outcomes, shared design, and learning, assessment methods for classrooms, and various factors characteristic to school settings and instruction by non-expert programmer teachers.

This article seeks to expand the horizons of the computational and social aspects in CT education in Scratch in K–9 classrooms. A four-dimensional theoretical foundation of CT’s more practice-like qualities (planning; iteration; collaboration, social interactions, and remixing; and debugging) is utilised in this temporally focused empirical case study. We investigate how fourth-grade student dyads carry out CT-fostering programming activities depicted by the CT dimensions in Scratch in the classroom from video data. The dyads’ open-ended programming processes are explored and described through overlapping layers: what happens on the computer screen (design events), who uses the computer (computer control), and what kind of talk is occurring (talk). The goal of the analyses is to reveal and holistically understand key computational and social factors that can influence students’ open-ended pair programming processes and potentially regulate their artefact-related outcomes, shared design processes, and learning in this educational context. The results, discernible through a multi-layered rather than a restricted analytical lense, are intended especially for pragmatic utilisation (i.e. informing classroom pedagogy) and pinpointing focal additional research in this still rather meagerly understood research topic.

2 Computational thinking and programming in K–9

2.1 *Computational thinking in Scratch*

Computational thinking (CT) is a term that continues to puzzle educational stakeholders, partially due to inconsistent pedagogical emphases put on it. The question is whether CT promotes transferable cognitive skills [1], enhances abilities in computational problem-solving [11], or strengthens a critical perception of social and societal issues in the computational world [12]. Consequently, CT has been framed to encompass various cognitive, practical, and even attitudinal or perceptual dimensions for students to learn [2, 3].

Primary schools across the world have begun adopting an educational objective akin to CT, often by incorporating computer programming—a recognised pathway to foster especially the computational problem-solving aspect of CT—in curricula [13]. Among popular platforms to facilitate such learning is Scratch (Fig. 1), which is rooted in the interest-driven design of interactive media projects [5]. In Scratch, students can design visual ‘sprites’ and ‘backdrops’ and, with block-based coding, design computational sets of instructions as creative features that establish interactive games, stories, or animations.



Fig. 1. Creatively designed ‘sprites’ (penguin, fox) in a desert backdrop next to block-based code in Scratch (www.scratch.mit.edu).

Key skills and areas of understanding affiliated with CT can be concretised through specific core educational principles, such as ‘planning human-readable representations and models of an algorithmic design’ [4] and ‘isolating errors and fixing them’ [14]. In computational problem-solving with Scratch, students’ CT, as construed through such principles, can be examined through what they can do with the available programmatic affordances [15]. In particular, examining students’ programming processes can provide an in-depth and valid view of their CT [6]. This notion is encouraged by Blikstein et al. [16], who saw that examinations on programming processes rather than the final project uncovered ‘counterintuitive data’ and ‘patterns with better predictive power than exams’. Furthermore,

pragmatically, assessing students' programming processes potentially enables valuable timely learning support [17].

We view that CT can be meaningfully construed as a holistic competence, especially in the open-ended collaborative programming with Scratch at the introductory level of learning in schools. At least four conceptually different but practically intertwined CT dimensions, which depict programming activities that students can learn and effectuate when designing their Scratch projects, can thus play a role in shaping students' programming processes as a whole (e.g. at different times in the process or from different viewpoints):

- *Planning* involves planning and designing (human-readable) representations of the structure, appearance, and functionality of solutions with, for instance, algorithmic flowcharts, pseudo-code, drawings, and lists, for them to be subsequently designed [3, 14, 18].
- *Iteration* involves designing computer programs cyclically with phases of problem understanding, modelling, design and planning, scripting, testing, and debugging through various tasks, such as building and disassembling scripts, testing designed features, reading tutorials, creating or importing media, and accessing other people's work, to maintain a manageable design process and in-time discovery of faults [4, 5, 18, 19].
- *Collaboration, social interactions, and remixing* involve a set of different ways for seeking and receiving support, help, and resources for using tools and implementing solutions. They include being supported collaboratively by the partner at the computer and interacting with persons beyond it (e.g. peers, teachers) to avoid wild-goose chases, engage in self- and peer explanations for conceptual development, learn from explanations, and plan code rather than making random or trivial changes [4, 20, 21,

22]. They also include utilising external resources to reuse and build on existing concrete ideas, leading to the creation of more complex designs than one could create alone [4, 5].

- *Debugging* involves evaluating and verifying the designed solutions (scripts) effectively and fairly for accuracy and detecting potential flaws and fixing them [4, 5, 14].

2.2 *Pair programming*

The focal educational context of this study is the collaborative design of computational artefacts in Scratch. Pair programming, in which two programmers work together to achieve a common goal, has several recognised benefits, including building knowledge in CT, improving design quality, reducing defects in a project, and improving motivation, especially among younger and less experienced students [23, 24, 25]. The core of pair programming is that while the ‘driver’ controls the keyboard and/or the mouse, the ‘navigator’ assists in reviewing the process and verifying the design [26].

Learning CT through programming in pairs can be theorised through social constructionism [5], which is rooted in constructivist principles: collaborative artefact design in pairs with a shared goal involves social interactions, such as comparing viewpoints and negotiating, which are valuable for the construction of high-level knowledge [27] and important in CT [4]. Earlier research has shown that the negotiation of ideas while designing typically leads to more novelty in programmed artefacts [28]. Collaboration at the computer can, however, be seen to situate in a broader social context in which external social resources can be available as well. Specifically, to support learning, instructors (and potentially even peers in the classroom) can, for instance, model the process and provide scaffolds or help [29], such as direct explanations or indirect hints [22], which could be understood as forms of

social interactions initiated by the external persons or proactively by collaborating students to enhance learning processes. Additionally, with contemporary technologies, students can seek to enhance their design by interacting with socio-material resources beyond the classroom, such as through the Scratch online community, by searching and remixing existing materials to receive concrete ideas and designs from other programmers' work to build upon [4, 5].

However, pair programming among young learners is influenced by various factors, including skill level, previous learning history, learning strategies, attitude toward collaboration, personality traits, emotions, and the physical environment [30, 31, 32]. A shared programming process can include different kinds of talk at different times and when operating in different programming roles [21, 33, 34]. In particular, mutual talk can be more or less agreed or disagreed, critical or uncritical, or discussed or non-engaged [28, 35], potentially encompassing uptake, praise, conflicts, and even antagonisation [36]. Especially the 'driver', controlling the computer and the code, has been found to have more opportunities to be more dominant [35, 37].

Currently, there is scant knowledge how the versatile social factors can interconnect with the more design-focused aspect [31]—empirically an unexplored domain already by itself—in collaborative open-ended programming—and how students' CT learning could be supported in the holistic programming process. Despite the potential benefits of collaboration and social interactions at and beyond the computer, designing computations even with tools designed for young learners can be difficult [6]. Students can face various challenges while programming and need different kinds of help, varying from validation to knowledge on implementing an entire solution [38], and may or may not manage to receive it [22] from the partner or elsewhere [31]. Errors and bugs, which are often results of the difference between what students want the computer to do (drawing from their mental models) and what the computer actually does, are nearly always involved in programming [39]. Research on

debugging goes back several decades [40], but the emergence and wider spread of new pedagogies and technologies justify distinctly examining contemporary contexts.

2.3 *The current study*

This maturing research topic includes several emergent needs, especially concerning the identification of key trouble spots in learning CT through programming and finding appropriate evidence-based pedagogical ways to support students' learning meaningfully in classroom practice. Research is needed to investigate the ways in which primary school students carry out CT-fostering programming activities (here: 'CT activities') in pairs in classrooms and to holistically examine key computational and social factors surrounding them and potentially influencing artefact-related outcomes, shared design processes, and learning.

The purpose of this study is to expand knowledge of students' CT activities in collaborative programming settings in schools especially to inform classroom pedagogy and highlight topics for more focal research in CT education. Utilising a sample of four fourth-grade student dyads' programming processes from two open-ended Scratch programming sessions, this study sets out to answer an overarching research question (RQ): **How do 4th grade students carry out CT-fostering Scratch programming activities as dyads?** This RQ is answered in this article through four sub-questions corresponding the four focal CT dimensions: How did the dyads...

1. ...plan their open-ended Scratch projects? (Planning)
2. ...cyclically design the projects? (Iteration)
3. ...mutually participate in the design, activate teachers and peers, and search for external materials? (Collaboration, social interactions, and remixing)
4. ...locate and fix bugs? (Debugging)

3 Methods

3.1 *Research design*

This multi-case study adopted exploratory and descriptive approaches with mixed methods. The studied cases involved dyads programming in Scratch. The priorities were to reveal and explain the characteristics of the studied phenomena and produce theoretical ideas, propositions, and hypotheses for further research. We aimed to compose rich accounts of the programming processes and attain novel information regarding them rather than make systematic comparisons and find trends for generalisation. The investigation leaned on the above-established theoretical underpinnings established through a diligent literature review [41].

3.2 *Participants and context*

The participants were fourth-grade students (10 to 11 years old) and their regular teachers from three classes in one average-sized Finnish primary school. The students participated in a Scratch programming course [42] organised and taught by the first author (the visiting teacher) in 2017 in the school's computer laboratory. The main goal of the course was to introduce students to creative, interest-driven graphical programming and basic programming contents and activities in Scratch based on the guidelines of the Finnish core curriculum. The course was implemented through thirteen 45-minute sessions, which addressed such contents and activities as 'sequences', 'looping', 'initialisation', and 'debugging' through tutorials, debugging challenges, remixing, and open-ended design projects. Informed consent to participate was received from the students' legal guardians. The participants are given pseudonyms to protect their anonymity.

All students designed open-ended interactive games, stories, or animations as a final assignment of the course. The only requirement was that the projects included interactive

elements. The students formed groups and planned their projects using pen and paper over one 45-minute session. The core idea in planning was ‘ideas-first’, but the visiting teacher attempted to provide more feasible alternatives to ideas that seemed ambitious with regard to practical affordances in Scratch or the allocated time. The students were provided an opportunity to use the computer while planning.

Subsequently, the students implemented their plans in Scratch over two 45-minute sessions. The visiting teacher instructed the students to adopt the roles of ‘driver’ and ‘navigator’ in turns while all the teachers guided the students’ work. Collaboration with peers and working at home were encouraged to promote sharing and informal learning.

We aimed to gather data from several dyads to gain extensive insight but were required to maintain a moderate workload in analysis. Twelve random dyads were selected for the study. Dyads with absentees and those with data losses from technical difficulties were omitted, resulting in four dyads with rich data for analysis.

3.3 *Data collection*

We adopted non-intrusive data collection methods to capture the dyads’ naturally occurring behaviour in authentic situations. Video technology enables capturing complex phenomena and ‘ex-post facto’ systematically observing and analysing different areas of interest [43].

The computer screens were recorded with a software called CamStudio, and a GoPro camera and a voice recorder were set next to each dyad (Fig. 2.). Video and audio files for each session were merged and time-synchronised for analysis. In total, eight videos ($M^{\text{duration}} = 38 \text{ min } 0 \text{ sec}$) – two sessions from the four dyads – were analysed. The dyads’ initial project plans and screen captures of their finished projects in Scratch were also collected as data.

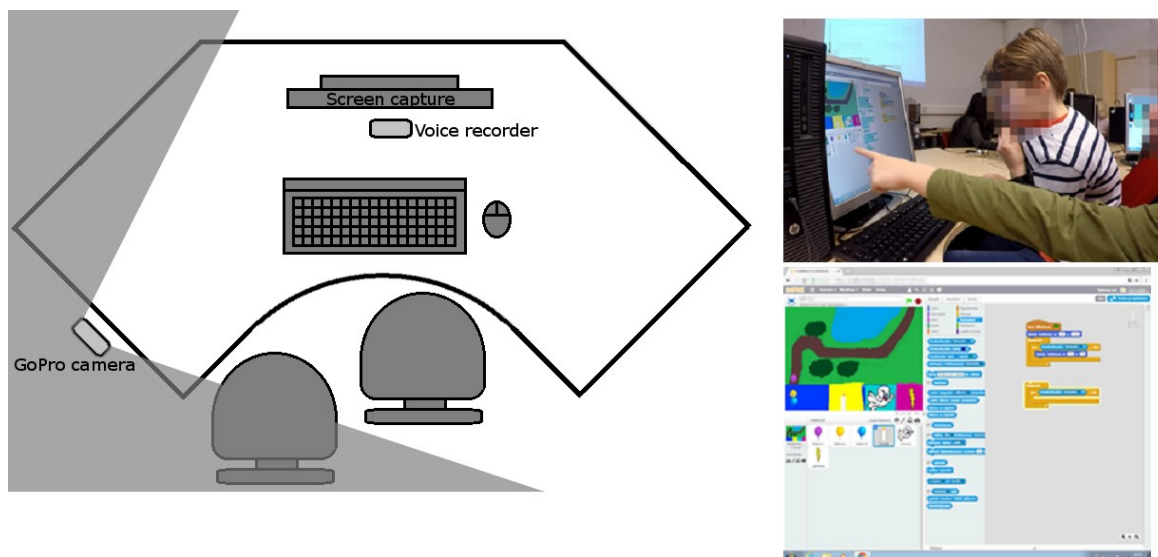


Fig. 2. The data collection setting and sample footage.

3.4 *Data analysis*

We found that the studied CT dimensions targeted by the four sub-RQs were not directly identifiable in the artefact and video data. Instead, we were required to analyse smaller, straightforwardly identifiable occurrences (introduced in the following subsections) that jointly denoted them in the data through select courses.

3.4.1 *Project plans and final projects*

The analysis of the project plans began by identifying what the students aimed to design (programming contents, graphics, audio) based on their initial project plans. Content analysis was performed to describe the planning methods (e.g. algorithmic flowcharts, pseudo-code) [18] and, with a framework adopted from Fagerlund et al. [42], to specify programming contents (coding patterns, code constructs) and their estimated difficulty as either basic (e.g. timed animations, sprite clicking, monologues) or advanced (e.g. custom variables, collision, state-sync) based on contents introduced previously during the course. The finished projects were compared to the initial plans in terms of quantity and quality of the features.

3.4.2 *The programming processes*

The aim of analysing the videos was to identify major events and patterns [44] in the students' CT activities in the four studied CT dimensions. We began by viewing the videos to familiarise ourselves with them and find an optimal fit between the units of analysis and the observations. One investigator developed an initial coding scheme based on previous research, and it was subsequently tested and revised iteratively with a blind coder.

The data was first coded on a surface level (low-inferent coding) and subsequently on a depth level (high-inferent coding) [44]. The sampling scheme for coding was selected as interval-based for conveniently examining sequences and transitions from one code to another to discover patterns. An optimal interval was set at five seconds to differentiate the most potentially significant events (e.g. a quick question amid making commands, a burst sequence of coding followed by a playtest). The videos were coded with Excel on three overlapping layers: *design events*, *computer control*, and *talk*. The coding was performed entirely by one investigator, and another investigator performed a blind 20% check including one entire video and random equal-length samples from each video using the same procedure. Cohen's kappa was .85 at the surface level, which was above the .75 threshold to indicate sufficient reliability, and .79 at the depth level, which was above the sufficient .60 threshold [45].

Design events (Table 1) were coded with a scheme adapted from Ke [19] with specifications: 'script analysis/design discussion' was omitted because it better suited the *talk* layer. 'Technical actions' was added because we wanted to distinguish specifically content-related design. Relevant events were specified as 'productive' or 'unproductive' based on whether a series of events for a specific purpose (e.g. a certain script) led to lasting changes or not. 'Testing play' was specified as 'validation' if no errors were revealed or 'bug reveal' when a programming error was apparently discovered.

Computer control (Table 2) was adapted from Höfer [26] to differentiate the programming roles.

Table 1. The coding of design events.

Design event	Surface level		Depth level specification
	Description		
<i>Off-task</i>	Actions on the computer irrelevant to project design.		
<i>Inactivity</i>	The cursor is not moving, and keys are not being pressed.		-
	The cursor is moving, but no clicks or selections are being made.		
<i>Technical actions</i>	Logging in. Naming the project. Navigating the Scratch interface (e.g. switching code block palettes, selecting editing tools). Modifying the project page.		
<i>Graphical design</i>	Sprites are being browsed in the library. Making graphical modifications in the backdrop editor or a separate graphics editor. Sprites are being dragged on the stage. Sprites are being enlarged with the growing tool. Image files are being explored from the computer. Sprites are being removed.		
<i>Audio design</i>	Sounds are being selected from the library. The microphone is being used to record sounds. Sound files are being explored on the computer.		<i>Productive / Unproductive</i>
<i>Coding</i>	Blocks are being added to the scripting area. Blocks are being moved in the scripts. Blocks' or sprites' property values are being adjusted.		
<i>Material searching</i>	Searching for other projects on the Scratch website. Search engines outside Scratch are being used.		
<i>Testing play</i>	Green flag is clicked. Keys are pressed on the keyboard to examine sprites' behaviours on the stage.		<i>Validation / Bug reveal</i>

Table 2. The coding of computer control.

Control type	Observation
<i>No-one</i>	No-one is using either the mouse or the keyboard.
<i>Conflict</i>	An attempt to take control of the mouse or the keyboard while the other one has it.
<i>Teacher</i>	
<i>Student A</i>	This person is using the mouse or the keyboard (or both).
<i>Student B</i>	

Talk was first coded on a relatively general level for ‘silence’, ‘teacher talk’, ‘mutual talk’, and ‘peer talk’ [31], for the audio data having become partially unclear due to classroom noise. In the depth level, teacher and peer talk was specified as information elicitation, the quality of sought help [38], the type of received help [22], or sharing solutions [5] (Table 3). Dyads’ mutual talk was specified based on its quality and the persons involved [31] (Table 4).

Table 3. The depth level coding of dyad talking with teachers or peers.

Talk type	Description and examples
<i>Other</i>	Talk unrelated to project design. Uninformative statements or exclamations. Unclear/unspecifiable. ‘Blah.’
<i>Eliciting information</i>	A teacher/peer asks for elaboration. The dyad explains. Teacher: ‘So, the mouse goes over the sprite, and it’s supposed to go up there?’
<i>Validation</i>	The dyad wants confirmation or a small piece of information. ‘In which studio do we put this?’
<i>‘Where’</i>	The dyad knows what tool they need to use, but wants help to find it. ‘Where is the UFO costume?’
<i>Help-seeking</i>	<i>‘What’</i> The dyad knows how they should proceed, but wants help to know what tool to use. ‘How can we save this [custom backdrop]?’
<i>‘How’</i>	The dyad knows what they should achieve, but wants help to find a way to achieve it. ‘It [a sprite] should go up there when the mouse touches it, but it doesn’t.’
<i>Help-receiving</i>	<i>Executive</i> A teacher/peer takes over and solves the task or tells directly what the dyad should do. ‘Click that script. Does it work if you take that “broadcast” out of there?’
<i>Instrumental</i>	A teacher/peer provides explanations, hints, or suggestions or thinks aloud about what the students could do. ‘I think that “broadcast” block should be somehow different.’
<i>Sharing solutions</i>	Students are sharing project ideas or giving other kinds of assistance to their peers. ‘I can show you now [how to draw a backdrop].’

Table 4. The depth level coding manual for dyads' mutual talk.

Talk type	Description and examples
<i>Disputing</i>	The students are engaging in opposing or unconstructive confrontation, disagreement, or talking out of turn. 'Why can't I do anything?' 'That's why.'
<i>Driver/navigator telling</i>	The driver/navigator alone is making commands, stating their intentions, or explaining what they are doing. 'Put that there.'
<i>Driver/navigator opening</i>	The driver/navigator alone is making propositions, questions, or planning aloud. 'Let's make a "game over" background, ok?'
<i>Negotiating</i>	The students are making constructive or relevant commands, explanations, propositions, questions, or planning aloud in turns. 'Should that be a bit bigger?' 'Yeah, change its size.'

3.4.3 Method of interpreting the findings

We specified main targets for interpreting relevant findings from the multi-layered coding of the design events, computer control, and talk (Table 5). For example, for understanding debugging, we examined 'bug reveals' and 'validation' (design events, depth level) further. For collaboration, social interactions, and remixing, we examined computer control and mutual talk as components of roles and participation (collaboration), all talk generally for potentially involving the receiving of support for the design (social interactions), and 'material searching' as indications of procedures in utilising external resources (remixing). The interpretations utilised qualitative and quantitative methods: for instance, interpreting iteration prompted examining the frequencies, temporal positions, transitions, description of content (e.g. what is being manipulated in Scratch), and evaluation of significance (e.g. perceived impact on the project) of all the coded design events representing the entire design process. Additional remarks were driven exploratorily from the data, such as from gestures. Resultantly, the gained information was manifold, and, structured based on the studied CT dimensions, it was presented selectively on a substantiality basis as intact portrayals of the

dyads' programming processes.

Table 5. Main targets for interpreting the dyads' CT activities from the coded data.

CT dimension	Analysis layer	Main targets for interpretation
Planning (RQ1)	<i>Project plans and final projects</i>	Identified programming contents, audio, and graphics <ul style="list-style-type: none"> • Type • Estimated difficulty • Method of planning (plans only)
	<i>Talk</i>	All talk codes: talk related to planning (*) <ul style="list-style-type: none"> • Description of content • Overlapping and consecutive codes
Iteration (RQ2)	<i>Design events</i>	All design events codes <ul style="list-style-type: none"> • Frequencies, temporal positions, transitions • Description of content • Evaluated significance • Overlapping and consecutive codes
	<i>Design events</i>	'Material searching' <ul style="list-style-type: none"> • Description of content • Evaluated significance • Overlapping and consecutive codes
Collaboration, social interactions, and remixing (RQ3)	<i>Computer control</i>	All computer control codes <ul style="list-style-type: none"> • Frequencies, temporal positions, transitions • Overlapping and consecutive codes
	<i>Talk</i>	All talk codes <ul style="list-style-type: none"> • Frequencies, temporal positions, transitions All talk codes: talk related to collaboration, social interactions, and remixing (*) <ul style="list-style-type: none"> • Description of content • Overlapping and consecutive codes
Debugging (RQ4)	<i>Design events</i>	'Testing play' <ul style="list-style-type: none"> • Description of content • Evaluated significance • Overlapping and consecutive codes

(*) Systematic analysis of the exact contents of talk was not suitable with the partially unclear audio data (see reliability and limitations).

4 Results and discussion

4.1 Overview of the dyads' projects

The dyads designed assorted creative Scratch projects (Table 6). The projects included two more game-like projects, a balloon tower defence (D1) and a princess rescue (D3), and two more story-like projects, a Harry potter story (D2) and a beach story (D4).

Table 6. Overview of the dyads' projects.

Dyad	Students	Project	Brief description
D1	Sami and Pete	Balloon tower defence	Balloons move automatically along a path. Using in-game currency, the player purchases defenders that destroy the balloons before they can reach the end.
D2	Johanna and Mari	Harry Potter story	The player must do as the story instructs to help Harry Potter and Severus Snape slay Voldemort.
D3	Anne and Marja	Princess rescue	The player controls a prince to move, avoid obstacles, and reach a princess to save her from an evil sorcerer.
D4	Tinja and Saana	Beach story	The player must do as the story instructs to return a beached whale back to the sea.

4.2 The programming processes

4.2.1 Sami and Pete (D1): Balloon tower defence

Planning. D1 had planned a tower defence by sketching such features as motion animation, sprite clicking, and in-game currency described in a pseudo-code-like manner (Fig. 3). We perceived the animations (F1, F6, F7) and sprite clicking (F2) as basic level features and the currency (F3), timer (F4), and losing condition (F5) more advanced for involving uninstructed programming contents (e.g. 'custom variables' [42]). The students had also drawn a play area (G1–G3) and the sprites and their animations (G4, G5) by hand.

As expected for discovery learning [5], further planning occurred on the computer by explaining and subsequently implementing spontaneous ideas (e.g. Sami 'opening': 'Let's

make a game over background, ok?'). As speculated by Campe et al. [31], we found that these findings co-occurred especially with 'inactivity', confirming how apparent passivity can encompass relevant design.

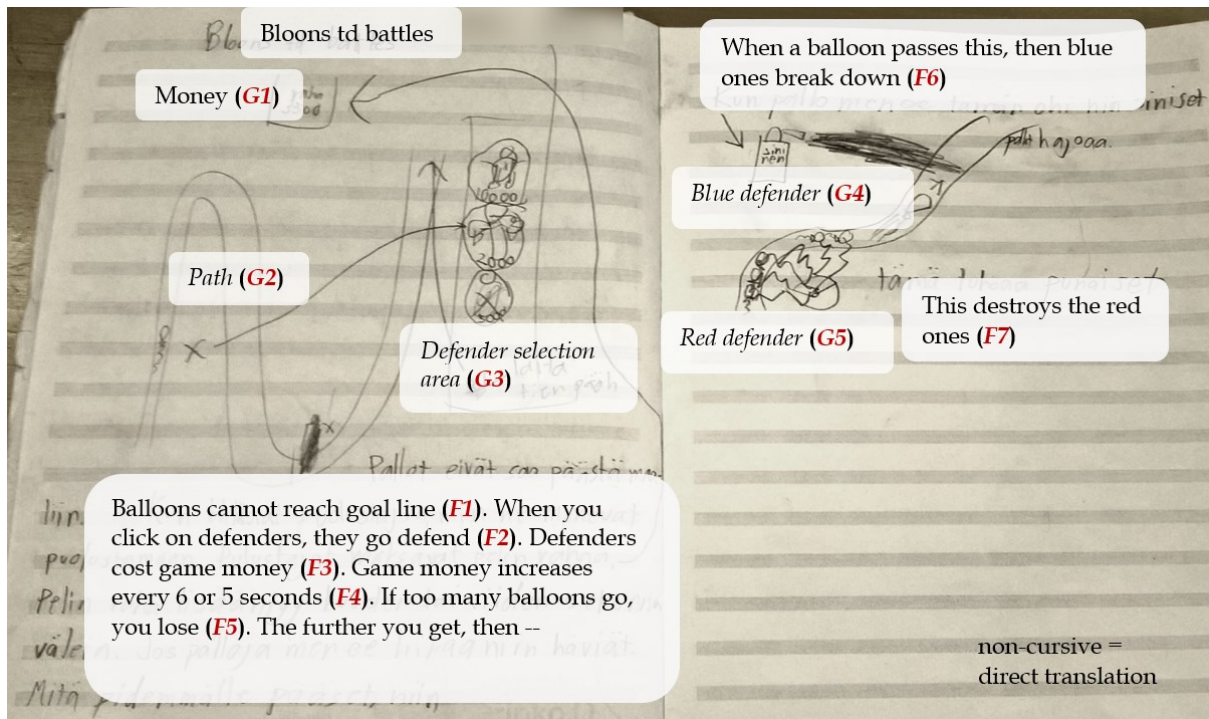


Fig. 3. D1's project plan (G=graphic, F=feature).

Iteration. Session one began by Sami mentioning having designed the main backdrop (G3), sprites (G4, G5), and animations (F1) at home. This effectively allowed D1 to begin with 'coding' (Fig. 4), focusing on the defenders' selection (F2). D1 'tested play' often within a minute of 'coding' (49 transitions) rather than effectuating other design events (9) apart from 'inactivity' (62), which involved on-topic discussion, and 'technical actions' (39), which was necessary between tasks (e.g. navigating the interface). D1 also typically 'coded' after 'bug reveals' (28) rather than continuing to, for instance, 'graphical design' (3). We calculated that the ratio of 'productive' to 'unproductive' design was 5:1 timewise, implying that the self-directed work was mainly efficient.

However, D1 transitioned eleven times to ‘unproductive’ design, which was mainly rather insignificant (involving e.g. cancelling inadequate designs) except for when ‘unproductive graphical design’ clustered over five minutes at the beginning of session two. The students, presumably unaware of the possibility of importing premade backdrops (or ‘remixing’ [5]), drew a spontaneously planned ‘game over’ backdrop and experienced multiple successive and evidently frustrating mishaps with the graphical editing tools. Although particular challenges could be considered small and predictable in discovery learning, the students may have saved time and trouble with instructional support [46], such as a demonstration [6] for ‘material searching’.

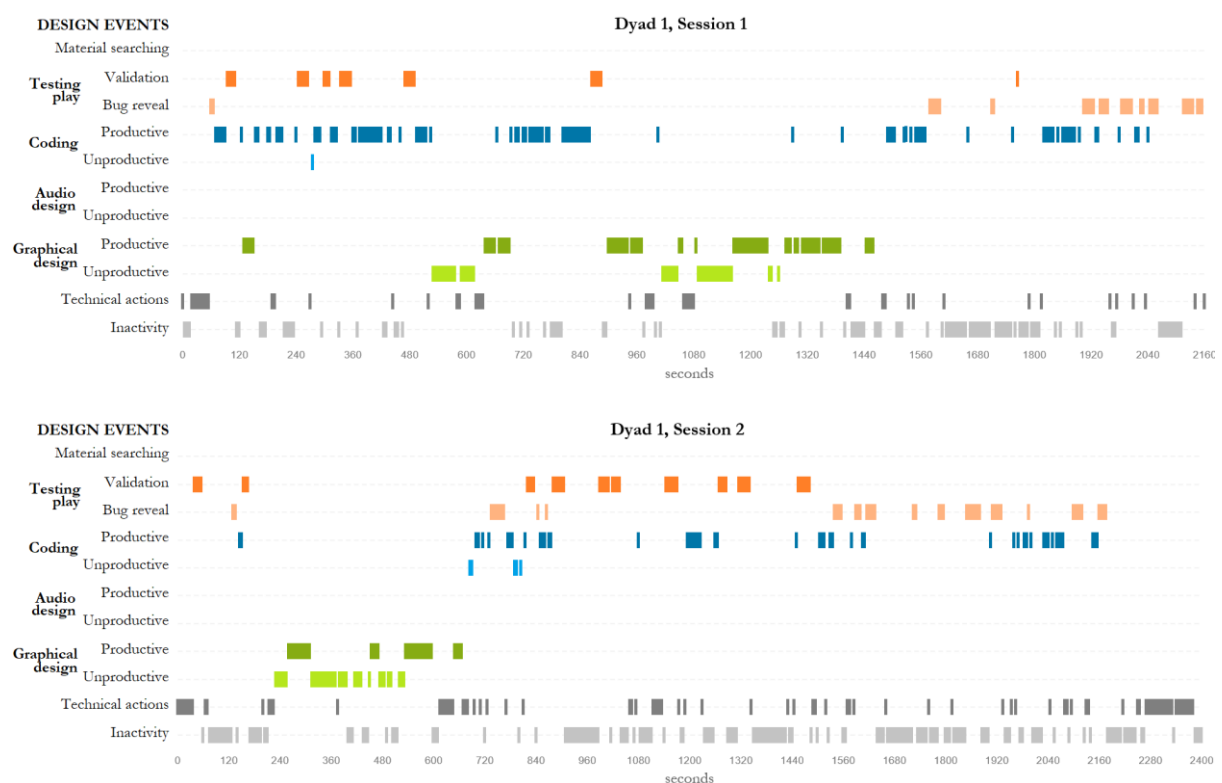


Fig. 4. D1’s design events over the two sessions.

Collaboration, social interactions, and remixing. Sami and Pete had exceptionally imbalanced roles: computer control among the students was virtually one-sided (Sami: 78%,

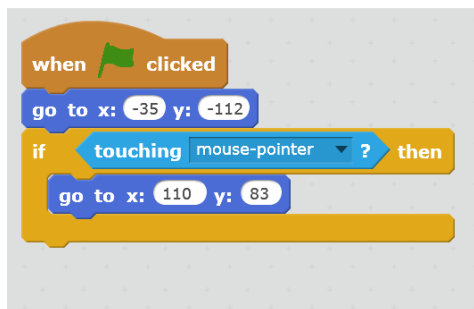
Pete: <1%). As evidenced previously [35, 37], the driver was also generally more verbal, accounting for five times more one-sided talk ('telling' or 'opening') than the navigator. Sami was 'coding' in 'silence' for 53% of all 'coding', and 51% of the talk during 'coding' encompassed him 'telling'. Furthermore, Sami fixed four of the dyad's seven bugs ('coding' after 'bug reveal') rather straightforwardly, primarily during 'silence' or while talking alone. He appeared more cognisant of the program's logic and likely felt more ownership [30] to the project he started at home. Pete objected only through soft questioning (e.g. 'Why can't I do anything?') and two attempts to gain computer control ('conflict'), which Sami warded off saying, for example, 'You don't know how to do this'. Pete may have lost track of the design, especially for the 'coding' aspect. Constructionist principles [5] imply that he may have also learnt less from the process.

Debugging. Pete nevertheless complemented Sami's skills and showed how two heads can be better than one [23]: first, he rightfully suspected a dysfunctional backdrop initialisation by proactively proposing ('opening') to 'test play' immediately after a 'validation'. Second, Sami was unable to fix a combination of incorrect events (when I receive-block) and coordinates in the defender selection (F2) alone. The students 'negotiated' for 10 minutes while 'coding' and 'testing play', attempting to find solutions. Although they modified mainly irrelevant code—a kind of trial-and-error or 'try-it-and-see-what-happens' behaviour [39]—the bug emerged as an opportunity to mend the lacking collaboration.

Another bug faced by D1 highlighted the importance of instructional support. The students sought help for 'how' to make the defenders move when touched by the mouse-pointer (F2) (Fig. 5). The regular teacher was unable to help, presumably for lacking programming expertise. Eventually, the visiting teacher explicated the issue and modified the scripts ('executive help'). However, the bug recurred only five minutes later, questioning whether the explication promoted learning. Guiding the students to solve the problem more

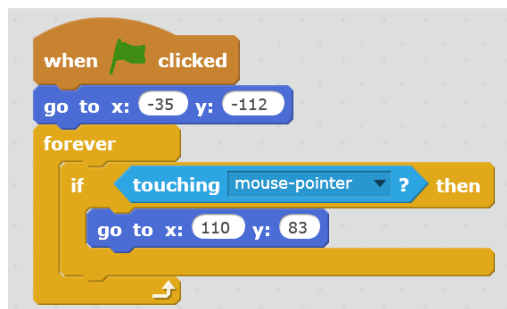
actively by themselves [45] could have been a better strategy. On another note, the entire data encompassed more ‘how’ help-seeking (9% of all talk) than the other kinds combined (6%), indicating that the dyads lacked understanding of several contents required by their plans. Moreover, moments of ‘help-seeking’ and subsequent ‘eliciting information’ to understand the problem better occasionally took several minutes, suggesting that communicative capabilities [2] were crucial. Otherwise, the entire data did not present substantial findings regarding help-seeking.

Bug: defenders are not moving when touched by the mouse-pointer.



```
when clicked
  go to x: -35 y: -112
  if touching mouse-pointer ? then
    go to x: 110 y: 83
```

A solution alternative.



```
when clicked
  go to x: -35 y: -112
  forever
    if touching mouse-pointer ? then
      go to x: 110 y: 83
```

Fig. 5. Left: a reconstruction of a bug that D1 and the regular teacher could not solve. Right: the visiting teacher’s implementation.

D1’s final project (Fig. 6) comprised approximately half of the planned features. The more advanced features (F3, F4) and the animations (F6, F7) were not implemented. The plan may have been slightly too complex, further justifying the suitability of game design for more experienced programmers [47]. However, this level of progress may be expected for the two 45-minute sessions, which were likely insufficient for the project.

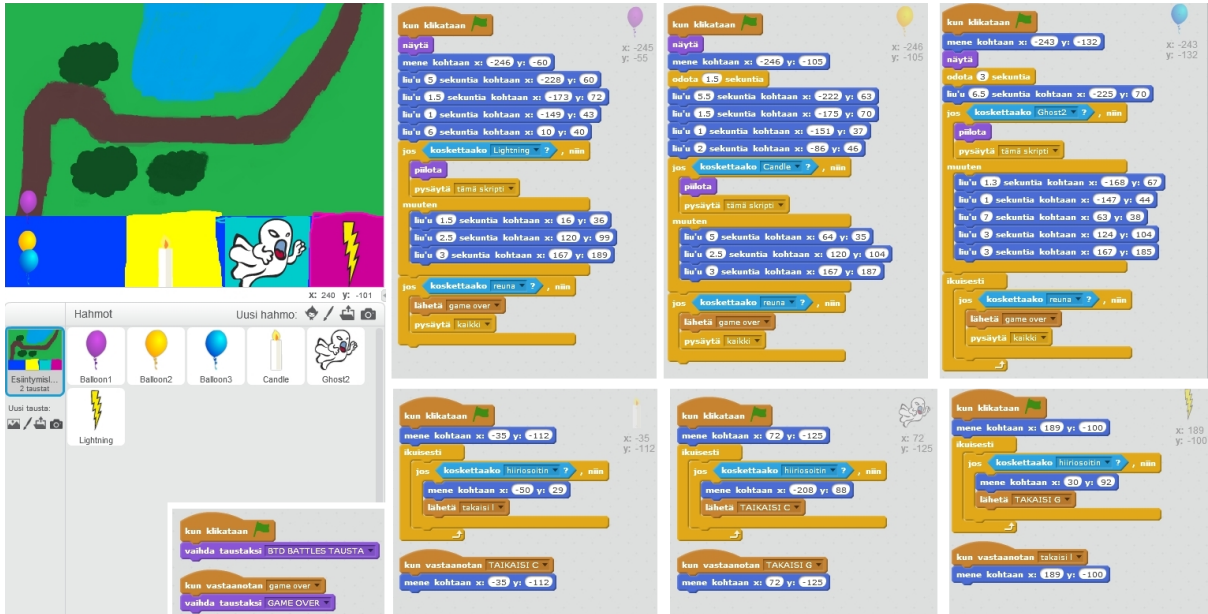


Fig. 6. D1's finished project.

4.2.2 Johanna and Mari (D2): Harry Potter story

Planning. D2 had planned an interactive Harry-Potter story entirely with text (Fig. 7). The plan included pseudo-code-like depictions of basic level contents introduced during the course: animations and user interaction (F1–F4). The plan also included text descriptions of backdrops and sprites (G1–G6) and audio-based dialogue (A1, A2). The students also planned during the sessions by sporadically explaining or implementing spontaneous ideas (e.g. Johanna: ‘Then this could say, like...’, before implementing a say-block unilaterally).

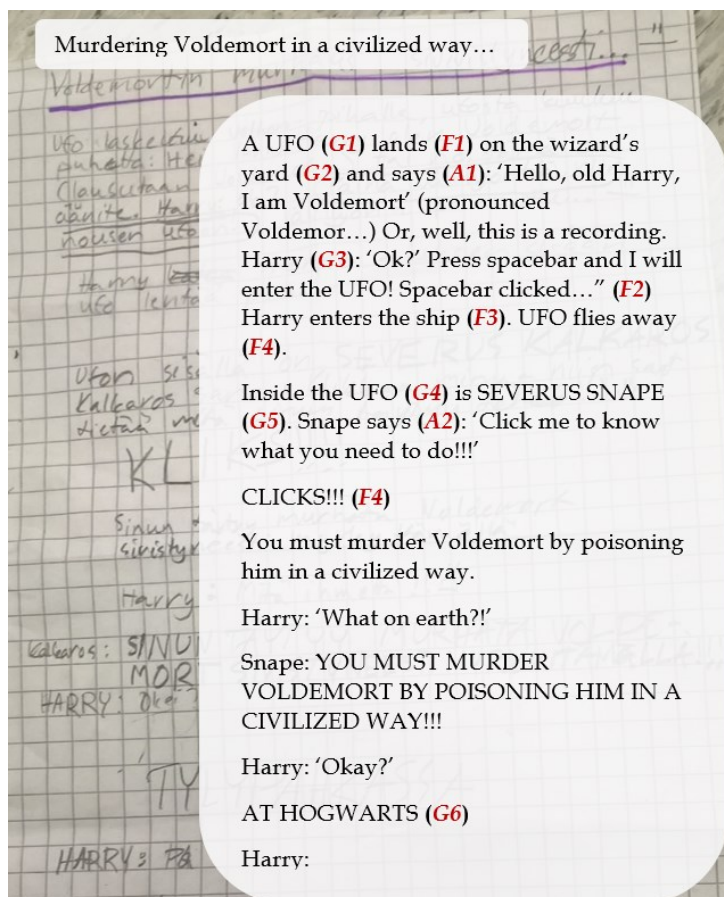


Fig. 7. D2's project plan (G=graphic, A=audio, F=feature).

Iteration. D2 began session one with approximately 21 minutes of 'graphical design' (Fig. 8), focusing on the main sprites and backdrops, and continuing to 'coding' the planned features only at the final third of the session. They 'tested play' often within a minute of 'coding' (52 transitions) rather than effectuating other design events (15) apart from the perhaps expected 'inactivity' (38) and 'technical actions' (25). They also often 'coded' after a 'bug reveal' (60) rather than doing something else (13). No 'unproductive coding' transpired, which may have been the benefit of the basic level plan. Otherwise, the ratio of 'productive' and 'unproductive' design was 3:2 timewise, which was explained by the fifteen transitions to 'unproductive graphical design'. Although this design involved insignificant matters (e.g. misclicking), Johanna and Mari were thrice stressfully hampered by an inability to find ways

to cancel accidental effects while lacking success to draw the planned UFO sprite (G1). After a prolonged struggle, they discouragingly selected a spaceship sprite.

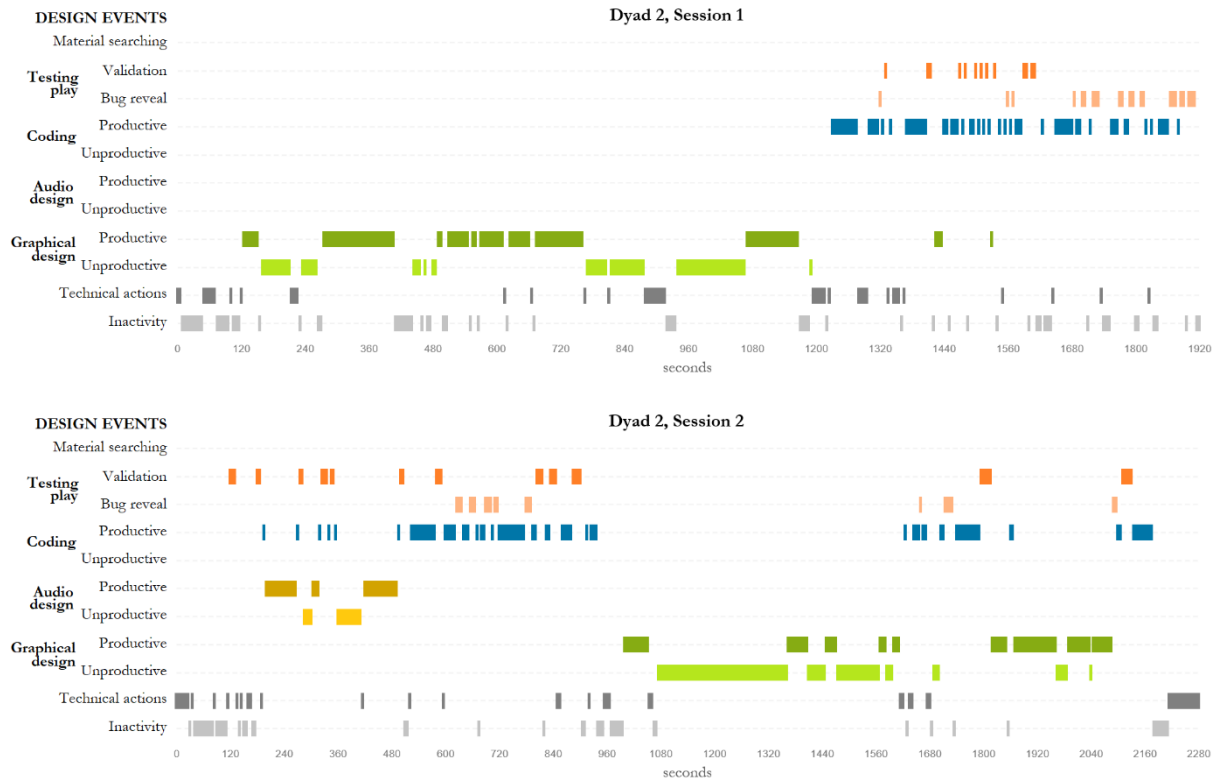


Fig. 8. D2’s design events over the two sessions.

Collaboration, social interactions, and remixing. Participation within D2 was highly irregular: Johanna ‘drove’ for 78% of the time (Mari: 10%). Additionally, although the dyad ‘negotiated’ for 47% of all mutual talk, Johanna accounted for 70% of all one-sided talk. Furthermore, major issues amidst particular design events surfaced: first, after a ‘bug reveal’ in a dialogue (A1), Mari (navigating) suggested using a broadcast-block to synchronise say-blocks. Johanna disregarded the idea (‘telling’) without expressed justification and, opting to use wait-blocks instead, continued to exhibit trial-and-error-like behaviour [39] by guessing rather than calculating appropriate time parameters (seconds) in the blocks. Similar

dominative behaviors [35, 37] potentially inhibiting new solutions [28] (and presumably learning) recurred twice more.

Second, the students switched computer control and even seats occasionally when Johanna explicitly asked Mari to take over, such as when she failed to colorise the backdrop (G4) precisely ('unproductive graphical design'). However, Mari got to 'drive' only during 'graphical design', and in 'silence' or while Johanna was talking. She appeared to want to capitalise on her opportunities and materialise ideas single-mindedly when receiving the chance to control the design. Subsequent 'disputing' and 'computer control conflicts' occurred, involving apparent dissatisfaction with observing something non-negotiated being designed and indicating a mismatch of interests [30]. Reflecting alternate viewpoints and reconciling differences could have benefitted from mediation and teacher modelling [3, 27]. However, such practices could have been regarded disadvantageous in the time-constrained context: a feeling of urgency and one-sided decision-making seemed to increase during session two, when Johanna repeatedly expressed her concerns about time and scolded Mari for making editing mistakes. Mari looked to dissociate further as the amount of 'negotiation' lessened from 45% to 21% between the sessions.

Debugging. D2 encountered altogether ten bugs in their 'bug reveals', four of which appeared substantial for involving the same construct: initialisation. At first, an uninitialised position animation (G3) led the students seek help ('how') and the visiting teacher recap the cause of the bug ('instrumental help'). The students subsequently encountered three uninitialised attributes (backdrop, position, visibility) when 'testing play', suggesting that the computational abstraction did not generalise well [4, 5]. A similar shortcoming was implied when the students sought help ('how') for implementing two algorithmically similar coding patterns, visibility animation and position animation, for the UFO's landing animation (F1). The visiting teacher articulated the entire solution in pseudo-code ('instrumental help'). The

students successfully ‘coded’ the visibility animation (step 1 in Fig. 9) and then continued ‘coding productively’ albeit inefficiently, dismantling the previous functional pattern (step 2) for an unknown reason, and seeking and receiving similar help again. Exceptionally, however, the navigator stepped in to complement the driver [23], whose understanding manifestly lacked yet dictated the process [35, 37]: she talked one-sidedly (e.g. stating or proposing what the driver should do) through a ‘coding’ sequence to a functional end result (step 3).

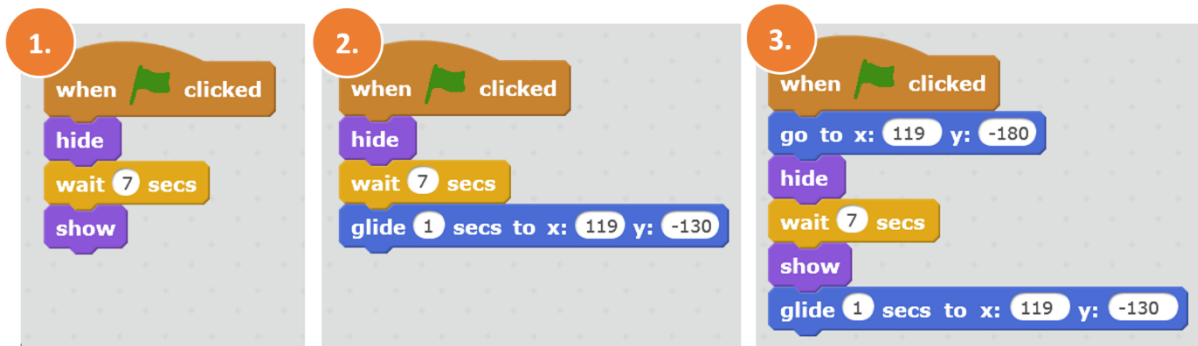


Fig. 9. D2’s process of implementing visibility and position animation patterns.

D2’s final project (Fig. 10) comprised a majority of the planned features. Only few motion animations (F3, F4) and the final, unplanned part of the story (G6) were not implemented, however, based on their general efficiency, a third session would have likely led to full completion. The plan representing contents introduced in the course may have sustained previous learning and led to the productive design, but whether the students discovered many new contents remained questionable.

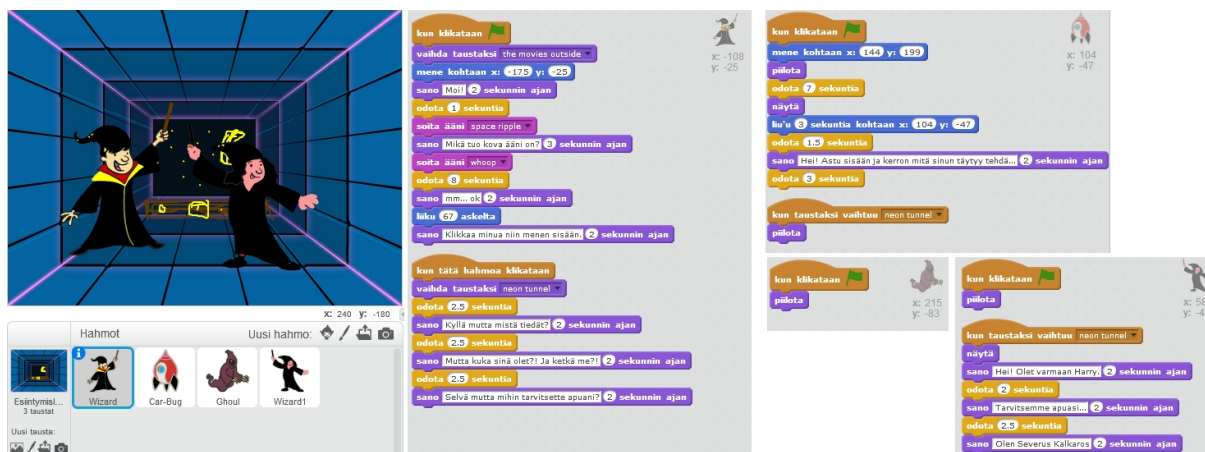


Fig. 10. D2's finished project.

4.2.3 Marja and Anne (D3): Princess rescue

Planning. D3's plan (Fig. 11) for a princess rescue project involved several hand-drawn pictures of the sprites and backdrops and ideas of programmable features and audio. We perceived that there were roughly as many basic (F1, F5, F6, F7) and advanced features (F2, F3, F4). The features were planned in generic human language (e.g. 'jump over hills') and, especially with the question-asking (F1), with more concrete pseudo-code resembling Scratch blocks. Additionally, the students specified their plans by explaining spontaneous ideas (e.g. Marja: 'We could do it [F3] like in the Mario game'), suggesting that the feature implementations were not necessarily conceived in full detail at first.

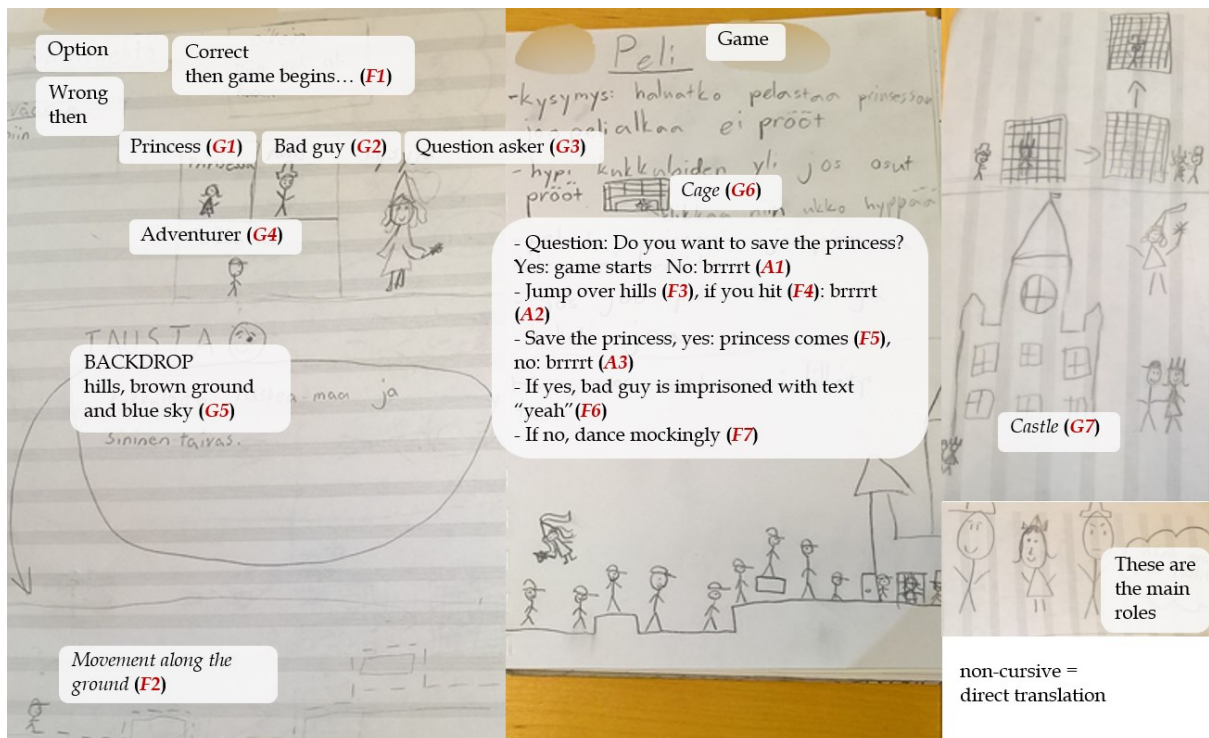


Fig. 11. D3's project plan (G=graphic, A=audio, F=feature).

Iteration. D3 dedicated session one predominantly to 'productive graphical design' (Fig. 12), namely for selecting premade sprites and drawing the backdrop. The ratio between 'productive' and 'unproductive' design was 6:1 timewise, implying mainly productive design. A switch from 'graphical design' to 'coding' occurred before the end of session one. Surprised at the passage of time, Anne presented the project to a peer, saying: 'You're already done? We haven't even properly started!' Relatedly, there was no substantial project-related talk with peers among any dyad, proposing that opportunities for spontaneous or deliberate sharing should perhaps have been facilitated [27].

44% of Marja and Anne's design was 'inactivity', which, however, encompassed more often on-topic 'negotiation' (37% of the time) than other talk types (e.g. silence: 31%, all one-sided talk: 17%). Nonetheless, it transpired mainly in session two, which also comprised much interlaced 'bug revealing' (84% of all playtests). D3 also continued more

often to ‘graphical design’ (21 transitions) than ‘coding’ (15) after ‘bug reveals’, signalling challenging debugging (see below).

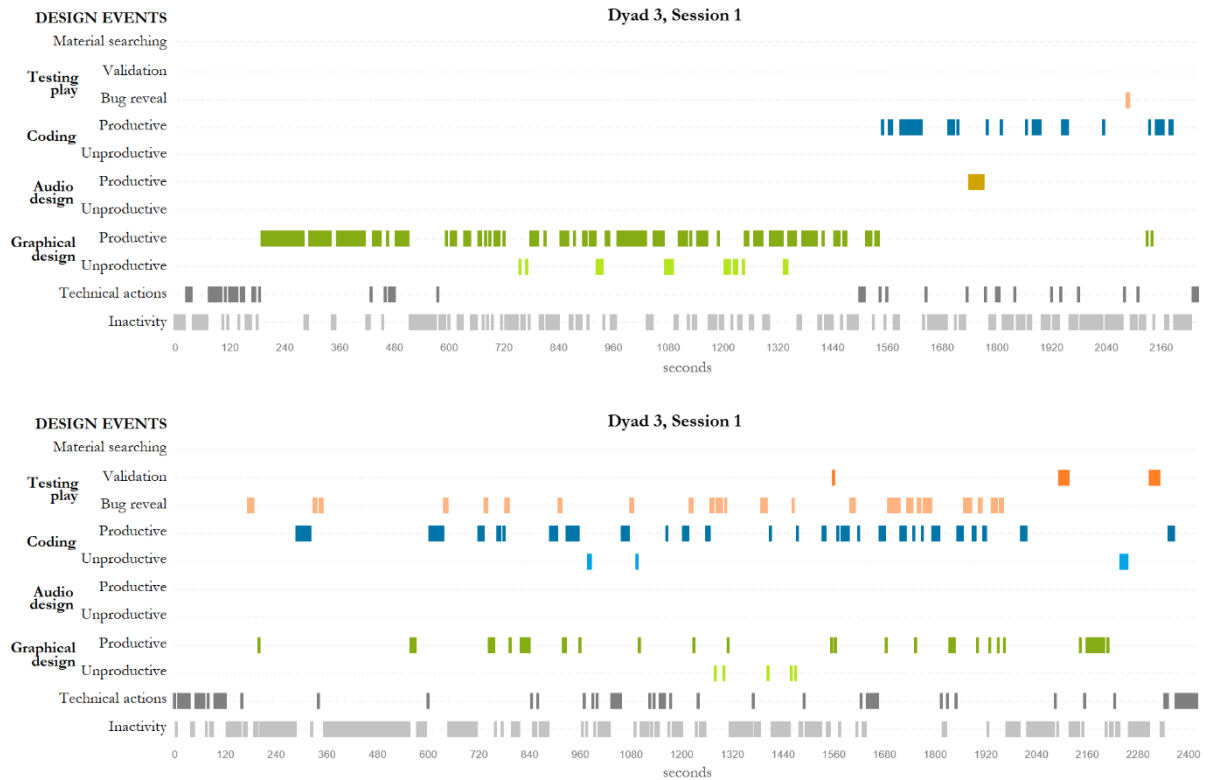


Fig. 12. D3’s design events over the two sessions.

Collaboration, social interactions, and remixing. There was an uneven aspect in D3’s participation: Marja ‘drove’ for 77% of the time (Anne: 11%). Otherwise, the process encompassed much ‘negotiation’ (70% of all mutual talk), fairly balanced one-sided talk (Marja: 12%, Anne: 17%), and harmonious switches in ‘driving’. Most importantly, Marja demonstrated ways of seeking help from the partner [34] by frequently asking for ideas and validation (‘opening’, accounting for 63% of the time while ‘driving’). She seemed keen to share responsibility [30] and pool knowledge [23], looking hesitant to work alone (‘inactivity’ while ‘driving’) especially when her partner was occupied elsewhere.

Illustratively, after a ‘bug reveal’, Marja heard a concrete solution from her partner and stated, ‘I could’ve never done this without you’.

Debugging. ‘Bug reveals’ influenced D3’s design process pivotally. Most crucially, an enduring collection of unresolved bugs emerged when the students could not implement their plans for player movement (F2), jumping (F4), and collision (F5). The visiting teacher first compared the intricate plan with the existing nascent scripts (‘instrumental help’) (step 1 in Fig. 13), presumably intending to highlight the missing contents and facilitate self-directed problem-solving [50]. Marja and Anne continued to demonstrate trial-and-error-like approaches [39] in their subsequent ‘coding’; adjusting irrelevant scripts (e.g. fine-adjusting the sprite’s initial position) and making apparent guesses (e.g. selecting the edge bounce-block). Although the features in question were introduced earlier during the course, the students appeared unable to translate their ideas into code or generalise abstractions in the new situation [4, 5].

The regular teacher arrived to ponder prospective solutions, providing ineffective support, such as suggesting modifying other irrelevant blocks. The visiting teacher then returned to reduce but not fully remove autonomy [50] by decomposing the pattern for jumping (F4) with pen and paper (‘instrumental help’) and ‘coding’ a half-complete script for the students (‘executive help’) (step 2). Making only gentle progress for nearly 10 minutes, D3 required constant ‘instrumental’ (e.g. validation) and ‘executive help’ (e.g. telling what blocks to implement) while struggling to find required blocks (‘technical actions’) and facing several new ‘bug reveals’ (e.g. step 3: mutually cancelling parallel animations). The dyad may have been led to managing too heavy information loads [6] with their plan, which was further enhanced by the requirement to experiment and evaluate design decisions [5]. Although the bugs could be interpreted as valuable expressions of gaps in the students’

knowledge [39], the requirement for constant help may have been unwarranted in the busy classroom.

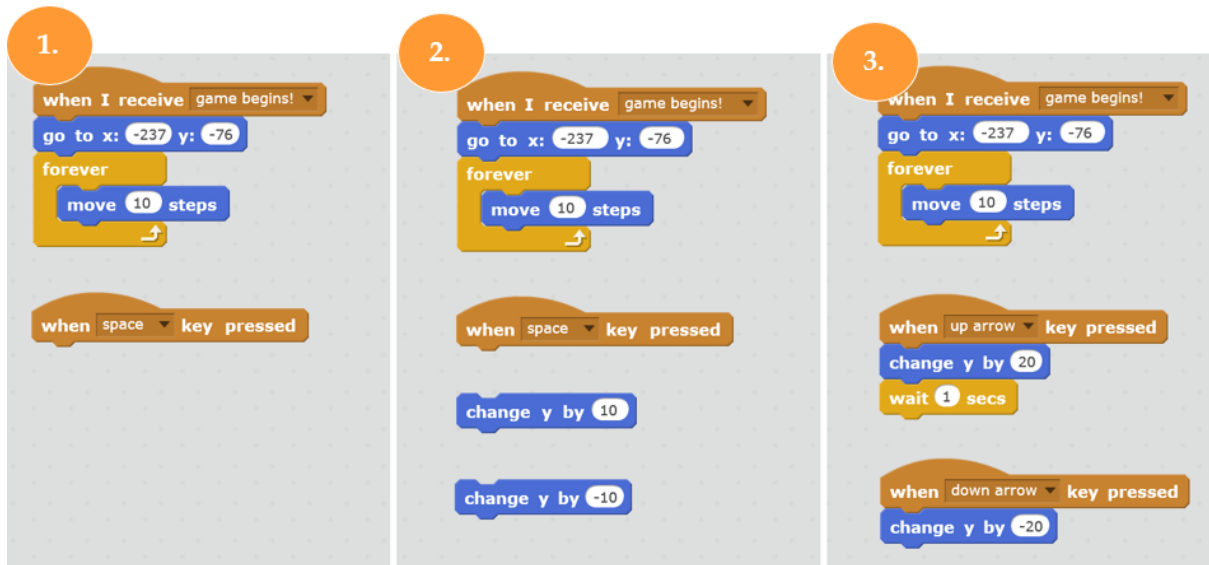


Fig. 13. D3's process of attempting to implement a jumping feature (F3).

D3's final project was relatively small (Fig. 14), but lacked only the more advanced features, such as the collision (F4) and the final animations (F5–F7). The game plan mismatched with the students' skills, the available support, and the allocated time. The teachers should have perhaps provided more constraints [50] during planning. Nevertheless, despite necessitating much individual support, the plan appeared to have led to learning through discovery [5] and even taught such meta-skills as dealing with complexity and uncertainty [2].

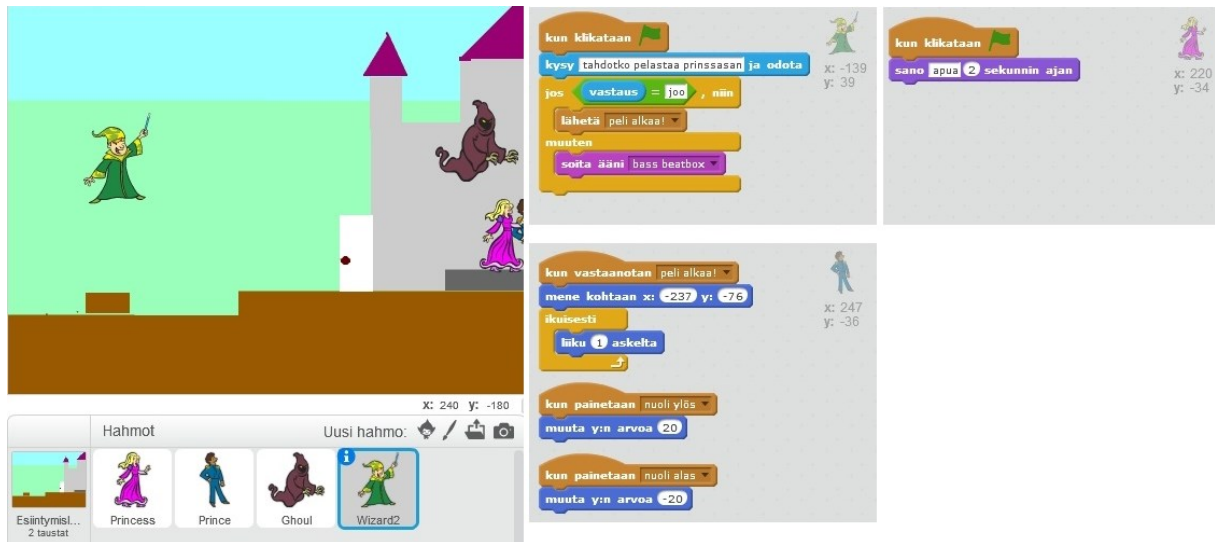


Fig. 14. D3's finished project.

4.2.4 Tinja and Saana (D4): Beach story

Planning. D4 had planned a beach-themed story entirely with text (Fig. 15). The plan encompassed human-language descriptions of graphics, which mainly resembled those in the Scratch library (G1, G2, G4), and a narrative progressing through exclusively basic level features: animations (e.g. F1) and interactive elements (e.g. F4).

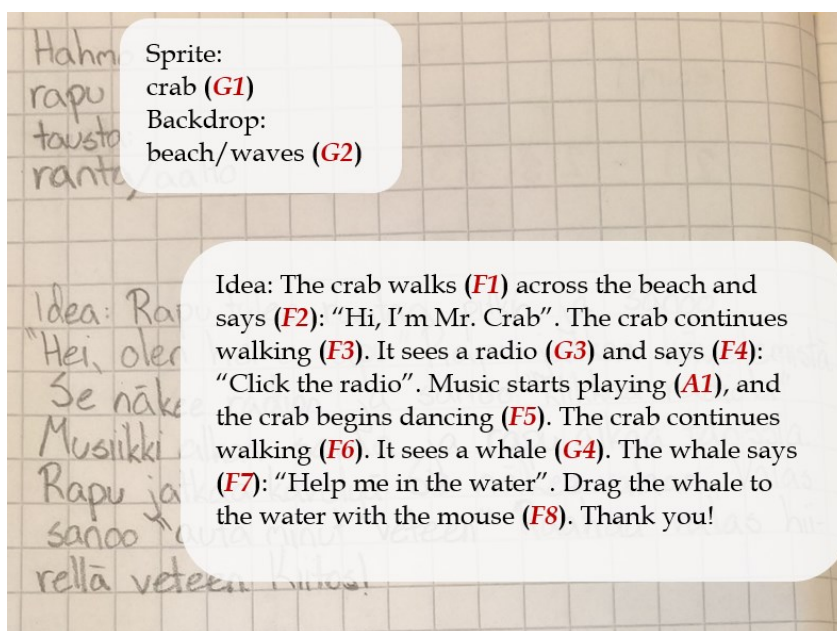


Fig. 15. D4's project plan (G=graphic, A=audio, F=feature).

Iteration. Tinja and Saana began session one mainly ‘productively’ with ‘graphical design’ (Fig. 16), in particular, by incorporating premade graphics and customising one sprite (G3), and ‘coding’ the first features in the story (e.g. F1). They effectuated altogether mostly ‘coding’ (27% of the time), an expected consequence of using mainly premade graphics. The plan appeared to have led to efficient design: the ratio between ‘productive’ and ‘unproductive’ design was 9:1 timewise, and all ‘unproductive’ design (e.g. quickly cancelling made changes) was unsubstantial. The students also typically ‘tested play’ within a minute of ‘coding’ (67 transitions) rather than effectuating other design events (16) besides ‘inactivity’ (72) or ‘technical actions’ (64). They also typically ‘coded’ after a ‘bug reveal’ (19) instead of effectuating other design events (3). A unique moment of ‘raising the ceiling of learning’ [5] even occurred: the dyad had implemented a looped costume animation (F5: crab dance) when the visiting teacher encouraged, ‘Now you have costume change, but there could be motion too’ (‘instrumental help’), successfully guiding the implementation of a parallel direction animation.

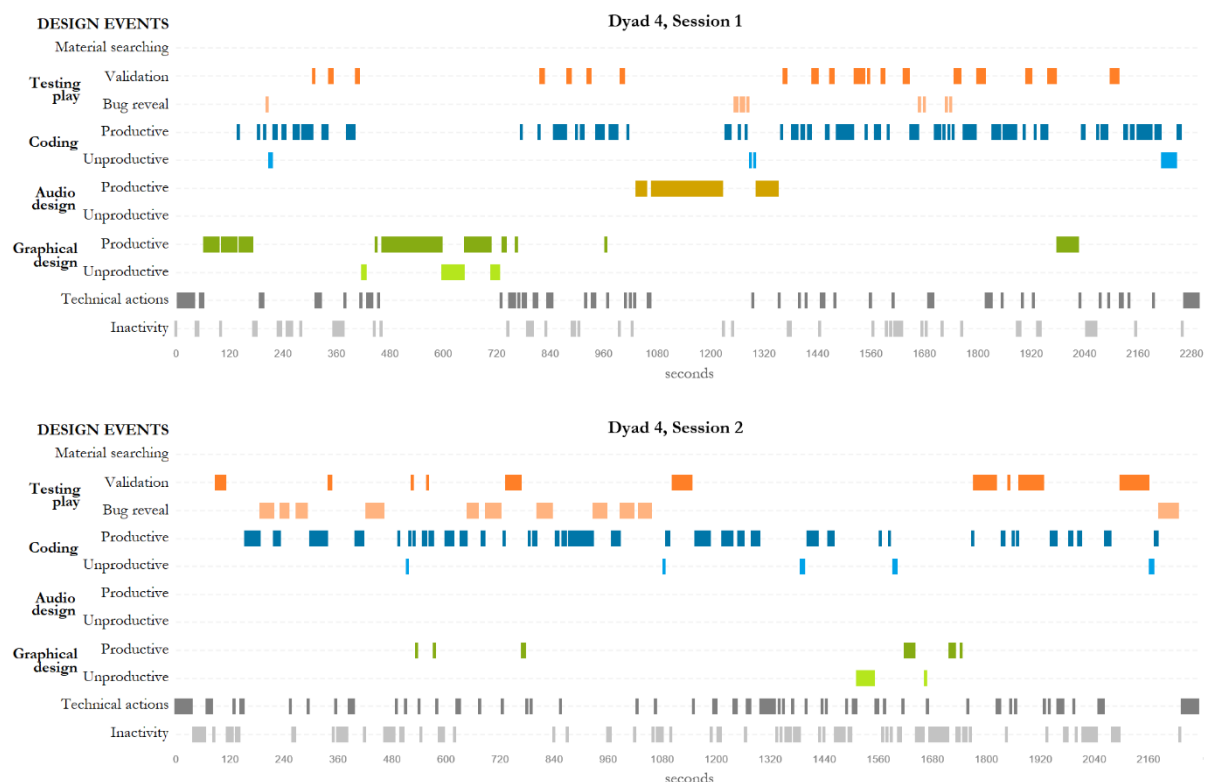


Fig. 16. D4's design events over the two sessions.

Collaboration, social interactions, and remixing. Demonstrating shared participation, Tinja and Saana 'negotiated' during all design events and for 66% of their mutual talk. However, computer control was highly unbalanced (Tinja: 95% 'driving', Mari: 2%). Although Tinja expressed a willingness to exchange roles twice during session one, she chased away Saana three times when her partner approached the input devices ('conflict'). Saana no longer attempted to acquire control in session two. The amount of 'negotiation' also lessened by two thirds between the sessions, and Tinja's 'telling' increased approximately as much. Accounting for 83% of all one-sided talk, Tinja appeared to govern of the design, especially toward the end. She may have lacked trust or been resistant to sharing ideas, especially after having began driving as the more outspoken personality. In turn, the navigator may have become frustrated at the lack of involvement [30] and dissociated from the process (and potentially learning) [5].

Debugging. Tinja and Saana's eleven bugs revealed through their 'bug reveals' highlighted three major issues. First, the dyad worked around bugs instead of solving them: two consecutively scripted play sound-blocks did not play in succession (A1) for lacking coordination (e.g. timing) [42]. The students included a longer sound clip instead of resolving the issue programmatically, highlighting a missed opportunity for timely support for timely discovery [5]. Another workaround concerned F8, which was implemented as a size animation. Tinja unilaterally 'coded' the pattern to encompass a relative (change size by) instead of absolute modification (set size to) as supposed initialisation [42]. Several initialisations existed in the project, and Saana (the navigator) even indirectly suggested ('opening'), 'At first, size must be at certain value'. The navigator seemed aware of the driver's mistake but possibly uncertain to indicate it [30].

Relatedly, the second major issue surfaced when D4 sought help for ‘how’ to implement final celebrations, a spontaneously planned feature. A backdrop animation was missing initialisation (set backdrop to) [42], which existed for other parameters. Information load [6] may have been influential here: the students may not have identified the required blocks from the abundant selection. They may have also been unsuccessful in generalising the abstraction with new parameters [38].

The third issue surfaced when say-blocks in a spontaneously planned dialogue ran simultaneously. Despite adding wait-blocks to relevant places to synchronise the say-blocks, Tinja began merely guessing the required seconds instead of calculating them [39]. A similar approach surfaced in F8 (the size animation), which was initialising erratically due to the relative initialisation. Not realising the misused blocks, the students effectuated ‘productive’ yet inefficient ‘coding’ and ‘testing play’ for 8 minutes by guessing appropriate parameter values in the change size by-blocks.

D4’s final project (Fig. 17) encompassed more than what they had planned. The rudimentary plan likely sustained previous learning and even facilitated the discovery of new contents [5]; however, Tinja’s dominative actions raised questions regarding Saana’s potential learning and enjoyment in the process.

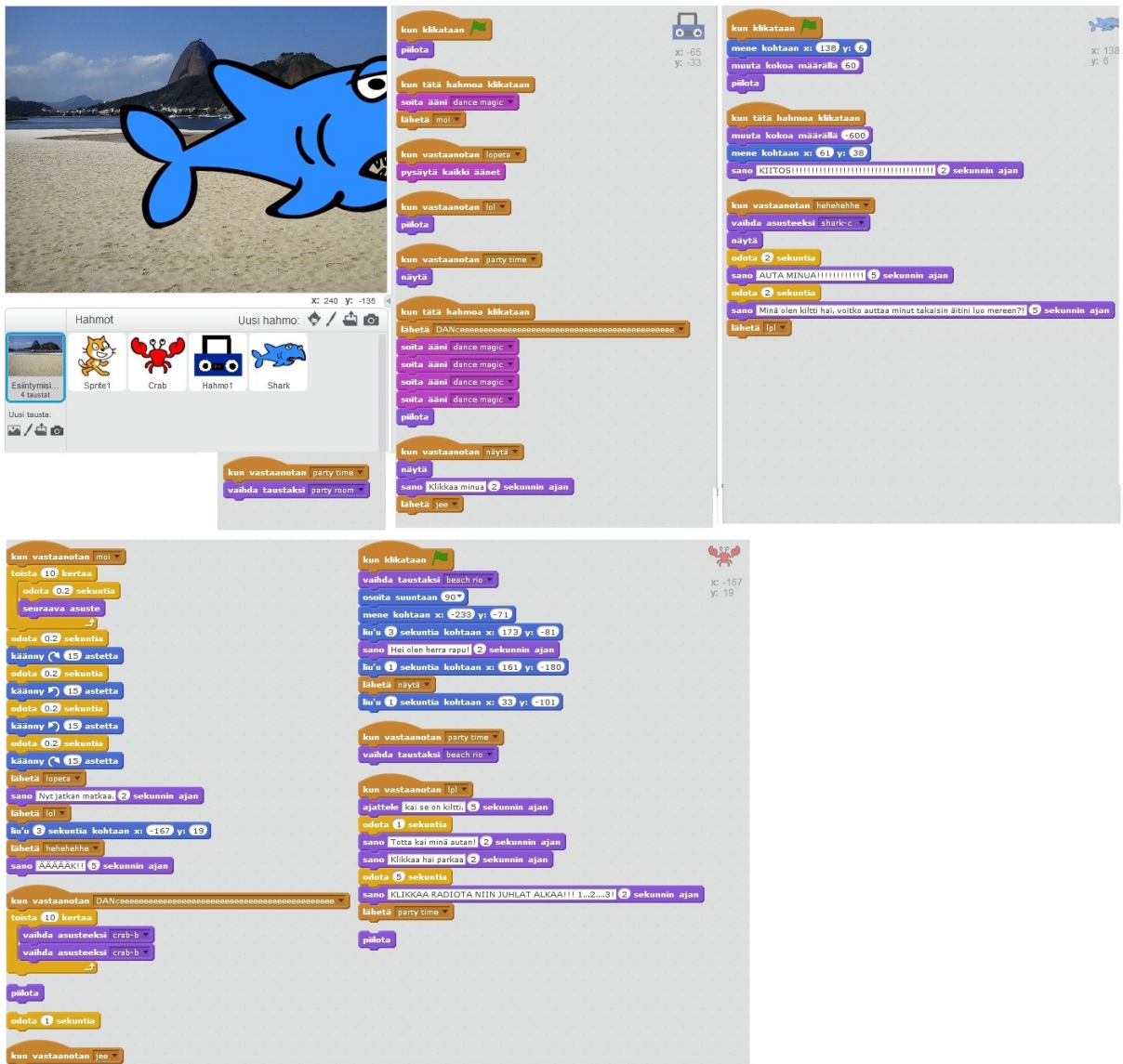


Fig. 17. D4's finished project.

5 Main findings and their implications

The analysis of the four dyads' design events, computer control, and talk together highlighted various computational and social factors that shaped their programming processes in different ways. We saw especially how the dyads succeeded and struggled when proceeding from their initial project plans to implementing them, while programming amongst themselves and while receiving support, and while carrying out social interactions during different design events. By way of explanation, we saw the importance of scrutinising different CT

dimensions to understand each dimension and the entire programming processes more fully. Below we recap the main findings from examining the processes on a more general level and discuss especially their pedagogical implications from three viewpoints that we found to meaningfully capsule key factors that influenced the dyads' design processes: planning open-ended projects, self-directed programming vs. instructional support, and promoting shared design processes. Notions on reliability and limitations of this study are also specified.

5.1 *Planning open-ended projects*

The dyads' initial project plans fundamentally guided their subsequent implementation processes. A more free-form (i.e. untechnical) approach in planning of sprites and their behaviours [18] appeared natural and characteristic to the students. However, with some dyads, translating plans into even familiar programming contents necessitated reducing autonomy [50] by designing half-complete scripts and demonstration [6] by pointing out similar abstractions in other scripts. Although such needs can be predictable at the introductory level of learning programming, they can be inappropriate to meet fully in classroom settings with several active learners. More critically, however, plans implying unawareness of operable designs, such as very human-language-like feature descriptions involving unintroduced programming contents, led to strenuous debugging, discouraging creative compromises, inefficient trial-and-error, and unfinished projects, demanding even intense support from knowledgeable teachers [49]. In contrast, though, it is important to note that such plans occasionally facilitated discovering new contents in a constructionist manner [5], as seen through spontaneous planning and refinement through on-screen actions [19] (e.g. browsing graphics) and discourse [31] (e.g. suggesting new ideas) during programming.

On another note, the dyads' processes of materialising their plans revealed a typical progression from graphical design, an integral yet potentially highly time-consuming phase,

to coding. This sequence may not come as a surprise in the design of creatively planned media artefacts in Scratch, but it appeared to have consequences in the classroom setting burdened by time: the teachers had to guide the dyads firmly with time management (e.g. telling to switch tasks). Indeed, the limited session time (2 x 45 minutes) appeared to pressurise and hinder both design events and several important tasks related to them, such as discovery, discussion, reflection, and reconciliation.

The main implications for planning open-ended Scratch projects in classrooms of this study stress the importance of instructionally guided initial planning, which should lead dyads toward design that is suitably challenging for them and toward instructional demand that is moderate for teachers. Although free-form planning appears feasible (and more technical computational modeling could be practiced in other contexts), planning may benefit from guidelines that orient students' toward planning basic core features (based on their earlier knowledge) and regarding more ambitious features as extraneous learning opportunities [46]. More story-like rather than game-like projects [47] seem more suitable for students becoming familiar with affordances and constraints in open-ended programming. More exact learning goals for students to integrate in their plans could be translated from programming content rubrics [42] and matched to students' individual capabilities [29]. Moreover, a priority between the more computational or algorithmic [1] and the more creative or self-expressive [5] emphasis may be required for a given open-ended task, for example, by restricting students' use of graphics to premade ones. Students' reflection of their design (e.g. time spent in different design events) as potentially facilitated by emerging visualisation methods for programming processes [48] could also be beneficial for (e.g. time-wise) self-regulated design.

5.2 *Self-directed programming vs. instructional support*

The novice programmer dyads' self-directed open-ended design raised expectations for an inherent occurrence of several generally profitable tendencies; namely, staying on-task, testing play regularly, attempting to fix revealed bugs, and making mainly productive design modifications. These tendencies emerged habitually likely for the students wanting to ensure functionality for their motivating creations. It is important to note, though, that interpreting the exact quality of such tendencies was not entirely unambiguous, as seen through such examples as 'productive coding' that led to new but computationally unorthodox or dysfunctional contents [42]. That said, programming challenges occurred especially through demanding or unresolved bugs and ineffectual practices, such as developing workarounds, adjusting or implementing irrelevant code, and guessing.

Although challenges carry a negative connotation (e.g. bugs are mistakes that must be corrected), they valuably demonstrated the students' programming incapacities [39] and appeared to lead learning organically to fruitful places. Additionally, the bugs that the students encountered may have partially represented misconceptions; however, in contrast to previous studies in Scratch [51], they seemed more of the basic kind [42], relating especially to control and coordination [40] as well as initialisation; an undoubtedly common bug (and thus fundamental content area) [38]. The bugs also suggested difficulties with managing the high loads of information present [6] and inability to generalise computational abstractions regularly [4, 5], burdening efficient designing even if previous knowledge existed. Such challenges can perhaps be expected when using an open-ended design tool with a range of features and design opportunities.

The results suggest altogether that learning CT through open-ended pair programming is generally viable once students have grasped (but can still be reinforcing their learning of) the basics of programming with Scratch. The several more autonomously occurring profitable

programming tendencies can require lesser instructional support. However, the sufficiency of pair programming and the Scratch environment (e.g. visual feedback from the simulation [39]) may not ensure efficient, high-quality design and learning without appropriate instructional support. Especially during coding, there can be key areas for deliberate support (e.g. for debugging more basic or advanced contents, depending on the educational level and prior learning). To welcome the pedagogical potential of challenges in programming, for instance, naturally occurring bugs can display instructional support needs and be used to guide learning by organising further drill-like practice or providing impromptu guidance. However, teacher knowledge [49], especially of effective programming solutions (e.g. coding patterns [42]) and good programming practices (e.g. generalising solutions) [15], may be necessary to provide effective support. To reduce teachers' instructional demand, automated script analysis tools, such as Dr. Scratch [7], could be alternately revisited to provide formative support, for instance, by having them provide ad-hoc guidance, such as directing attention to key places in the scripts (e.g. previously implemented similar abstractions).

5.3 Promoting shared design processes

Control of the purportedly shared design process emerged as a key issue in the pair programming. Most notably, the navigators were commonly left defending their ideas, especially for the coding, whereas the drivers were afforded with seemingly structural privilege [20, 35, 37], allowing them to dictate how they activate the navigator regardless of the quality of their partners' contributions (e.g. bug solutions, contents to implement).

Contrasting with the ideals of pair programming [26], this setup led to missed opportunities to implement new contents [28], thus also possibly learning [5], dissociation from the design [20], and altogether dissimilar knowledge acquisition and even interest or enjoyment.

Although the drivers' behaviour can be hypothesised through such factors as unwillingness to

receive criticism, lack of trust, unilaterally gained ownership, feeling of superiority, and mismatching interests [23, 31], no rationale emerged for the programming roles, as if they were pre-existing or were formed quite immediately without explicit discussion. The pressure of the limited session time may have also been reflected in this issue.

On a separate note, the examination of information sharing both in the classroom (i.e. sharing ideas with peers, help-seeking) and beyond it provided unfortunately scant information. There was altogether an unpredictable absence of meaningful design-related peer interactions during the design processes. However, such interactions were not necessarily feasible during the timewise pressurised sessions, which could be considered as kinds of design sprints that occurred between sessions of more facilitated sharing and feedback. The lack of utilisation of web resources, such as premade backdrops, in turn, could have prevented particular cumbersome challenges while editing the graphics and benefited learning in ways characteristic to ‘collaborating’ [5].

In conclusion, a risk of strong participatory imbalance and consequent many negative effects can reside in pair programming, and especially the better activation of navigators can require careful pedagogical consideration to enhance shared design processes. Potential solutions could include teacher mediation and modelling or even intelligent systems to reflect viewpoints and reconcile differences [27, 35, 36]. They could also include obligatory role switches during the processes [20] and intentional distribution of design tasks (e.g. coding and graphical design), pairing like-minded students [31], and even revising Scratch to allow two-computer design to promote shared agency [35, 37]. In turn, remixing practices may need to be facilitated by explicitly demonstrating their possibilities for the students. Design-related peer interactions (e.g. sharing, attaining feedback) may be appropriately facilitated in separate sessions to ensure that students are using their (often limited) time on the computers design-wise efficiently.

5.4 *Reliability and limitations of the study*

We omitted analysis of two CT dimensions: ‘patterns and generalisation’ and ‘problem decomposition’ [42]. We perceived that studying these cognitively deep-lying activities necessitates tailored approaches utilising, for instance, think-aloud protocols or artefact analyses that could have jeopardised the authenticity of the data or complicated the analyses. In turn, participants not outsourcing their thoughts constantly produced uncertainty in observing what they were thinking. Relatedly, the partially unclear audio data prevented analysing the exact contents of talk. Also, all relevant events in the classroom may not have been captured. The findings therefore may not represent reality in full detail. A similar limitation applies for the research design: reaching saturation would require examining more cases and adopting experimental designs.

The data collection may have affected the students’ behaviour. Although the cameras had been in the classroom for weeks, several students paid attention to them. We attempted to manage the issue by emphasising that the devices were used to record naturally occurring events rather than evaluation.

6 Concluding remarks

Educators need evidence-based pedagogical knowledge increasingly to support students’ CT learning through programming in general primary school classrooms. Focusing on open-ended creative programming with Scratch, this study sought to contribute to this need by answering an overarching RQ: How do 4th grade students carry out CT-fostering Scratch programming activities as dyads? This question was answered through four sub-questions targeting four conceptually different but practically intertwined CT dimensions: planning; iteration; collaboration, social interactions, and remixing; and debugging. The multi-layered analysis revealed that dyads’ open-ended programming processes can be shaped especially by

their initial project planning, autonomous programming tendencies, and habits of controlling the purportedly shared design process. The results implied specifically that, first, guiding novice programmer dyads' initial project planning is important to avoid looming design pitfalls (e.g. inability to effectuate the plans) and altogether steer toward suitable learning opportunities. In turn, more spontaneous discovery can evidently guide students' open-ended design, but it may require intense and time-consuming support from skilled instructors capable of guiding students to grasp the new opportunities. Second, dyads' self-directed programming can include many habitual effective tendencies, but their capability to proceed autonomously (e.g. fix bugs and generalise solutions effectively) can vary greatly and require timely support. Third—and perhaps most crucially—promoting pair programming processes that are shared rather than overly unilateral may require putting suitable pedagogical constraints (e.g. clear role assignments) in place.

Based on the insight gained in this study, more research is still required to thoroughly examine especially collaboration and social interactions in pair programming, for instance, through the interplay of particular design events and specific dialogue movements, such as planning [31], pooling knowledge constructively [23], and uncritical utterings [28, 35, 36]. Research could aim to better understand the robust role dynamics highlighting especially the navigators' incapability to effectuate important navigating tasks during specific design events and potentially learn as well as the driver. Additional research is also needed for better understanding the thinking and interaction processes involved in materialising creative plans into computational abstractions with previous knowledge and external support. In particular, pedagogical knowledge on debugging could be deepened, for instance, by further investigating key bugs [40] in Scratch to further explore novice programmers' common misconceptions [51] to inform appropriate forms of feedback and pedagogical planning in introductory open-ended programming. Altogether, appropriate methods for studying lengthy

temporal programming processes combining complex elements from both social and computational dimensions could also be further developed in the future.

Acknowledgements

We gratefully acknowledge the [organization blinded] for facilitation, [name blinded] for assistance with coding, and [name blinded] for the practical insights.

References

- [1] P. Denning, M. Tedre, Computational Thinking, MIT Press Ltd, 2019.
- [2] V. Barr, C. Stephenson, Bringing computational thinking to K–12: What is involved and what is the role of the computer science education community?, *ACM Inroads* 2:1 (2011) 48–54. <https://doi.org/10.1145/1929887.1929905>
- [3] T.-C. Hsu, S.-C. Chang, Y.-T. Hung, How to learn and how to teach computational thinking: Suggestions based on a review of the literature, *Computers and Education* 126 (2018) 296–310. <https://doi.org/10.1016/j.compedu.2018.07.004>
- [4] S. Grover, R. Pea, Computational thinking: A competency whose time has come, in: S. Sentance, E. Barendsen, C. Schulte (Eds.), *Computer Science Education: Perspectives on teaching and learning in school*, Bloomsbury Academic, London, 2018, pp. 19–37.
- [5] K. Brennan, M. Resnick, New frameworks for studying and assessing the development of computational thinking, Paper presented at the meeting of AERA 2012 (2012).
- [6] S.Y. Lye, J.H.L. Koh, Review on teaching and learning of computational thinking through programming: What is next for K–12? *Computers in Human Behavior* 41 (2014) 51–61. <https://doi.org/10.1016/j.chb.2014.09.012>
- [7] J. Moreno-León, G. Robles, M. Román-González, Dr. Scratch: Automatic analysis of Scratch projects to assess and foster computational thinking, *RED-Revista de Educación a Distancia* 15:46 (2015) 1–23. <http://doi.org/10.6018/red/46/10>
- [8] Y. Allsop, Assessing computational thinking process using a multiple evaluation approach. *International Journal of Child-Computer Interaction* 19 (2018) 30–55. <https://doi.org/10.1016/j.ijcci.2018.10.004>
- [9] C. Chalmers, Robotics and computational thinking in primary school, *International Journal of Child-Computer Interaction* 17 (2018) 93–100. <https://doi.org/10.1016/j.ijcci.2018.06.005>

- [10] M. Román-González, J.-C. Pérez-González, C. Jiménez-Fernández, Which cognitive abilities underlie computational thinking? Criterion validity of the Computational Thinking Test, *Computers in Human Behavior* 72 (2017) 678–691.
<https://doi.org/10.1016/j.chb.2016.08.047>
- [11] P. Denning, Remaining trouble spots with computational thinking, *Communications of the ACM* 60:6 (2017) 33–39. <https://doi.org/10.1145/2998438>
- [12] M. Høholt, D. Graungaard, N.O. Bouvin, M.G. Petersen, E. Eriksson, Towards a model of progression in computational empowerment in education, *International Journal of Child-Computer Interaction* 29 (2021) 100302.
<https://doi.org/10.1016/j.ijcci.2021.100302>
- [13] F. Heintz, L. Mannila, L. T. Färnqvist, A review of models for introducing computational thinking, computer science and computing in K–12 education, in: *Proceedings of the 2016 IEEE Frontiers in Education Conference (FIE)*, IEEE, 2016, pp. 1–9. <https://doi.org/10.1109/FIE.2016.7757410>.
- [14] A. Csizmadia, P. Curzon, M. Dorling, S. Humphreys, T. Ng, C. Selby, J. Woollard, *Computational thinking - A guide for teachers*, Computing at School, 2015.
- [15] J. Fagerlund, P. Häkkinen, M. Vesisenaho, J. Viiri, Computational thinking in programming with Scratch in primary schools: A systematic review, *Computer Applications in Engineering Education* 29:1 (2021) 1–17.
<https://doi.org/10.1002/cae.22255>
- [16] P. Blikstein, M. Worsley, C. Piech, M. Sahami, S. Cooper, D. Koller, Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming, *Journal of the Learning Sciences* 23:4 (2014) 561–599.
<https://doi.org/10.1080/10508406.2014.954750>
- [17] Q. Hao, D.H. Smith IV, L. Ding, A. Ko, C. Ottaway, J. Wilson, K.H. Arakawa, A. Turcan, T. Poehlman, T. Greer, Towards understanding the effective design of automated formative feedback for programming assignments, *Computer Science Education*, published online (2021). <https://doi.org/10.1080/08993408.2020.1860408>
- [18] Q. Burke, The markings of a new pencil: Introducing programming-as-writing in the middle school classroom, *Journal of Media Literacy Education* 4:2 (2012) 121–135.
- [19] F. Ke, An implementation of design-based learning through creating educational computer games: A case study on mathematics learning during design and computing,

Computers & Education 73 (2013) 26–39.

<https://doi.org/10.1016/j.compedu.2013.12.010>

- [20] C. Lewis, N. Shah, How equity and inequity can emerge in pair programming, in: B. Dorn (Ed.), Proceedings of the Eleventh Annual International Conference on International Computing Education Research (ICER '15), ACM, New York, 2015, pp. 41–50. <https://doi.org/10.1145/2787622.2787716>
- [21] C. M. Lewis, Is pair programming more effective than other forms of collaboration for young students? Computer Science Education 21:2 (2011) 105–134. <https://doi.org/10.1080/08993408.2011.579805>
- [22] K. Mäkitalo, C. Kohnle, F. Fischer, Computer-supported collaborative inquiry learning and classroom scripts: Effects on help-seeking processes and learning outcomes, Learning & Instruction 21:2 (2011) 257–266. <https://doi.org/10.1016/j.learninstruc.2010.07.001>
- [23] E. Arisholm, H. Gallis, T. Dybå, D.I.K. Sjøberg, Evaluating pair programming with respect to system complexity and programmer expertise, IEEE Transactions on Software Engineering 33:2 (2007) 65–86. <https://doi.org/10.1109/TSE.2007.17>
- [24] D. Preston, Pair programming as a model of collaborative learning: A review of the research, Journal of Computing Sciences in Colleges 4 (2005) 39–45.
- [25] X. Wei, L. Lin, N. Meng, W. Tan, S.-C. Kong, Kinshuk, 2021. The effectiveness of partial pair programming on elementary school students' computational thinking skills and self-efficacy. Computers & Education. 160, 104023. <https://doi.org/10.1016/J.COMPEDU.2020.104023>
- [26] A. Höfer, Video Analysis of Pair Programming, in: P. Kruchten, S. Adolph (Eds.), Proceedings of the 2008 International Workshop on Scrutinizing Agile Practices or Shoot-Out at the Agile Corral (APOS '08), ACM, New York, 2008, pp. 37–41. <https://doi.org/10.1145/1370143.1370151>
- [27] J. Roschelle, S.D. Teasley, The construction of shared knowledge in collaborative problem solving, in: C.E. O'Malley (Ed.), Computer-Supported Collaborative Learning, Springer, Berlin, Germany, 1995, pp. 69–197. https://doi.org/10.1007/978-3-642-85098-1_5
- [28] I. Deitrick, B. O'Connell, R.B. Shapiro, The Discourse of Creative Problem Solving in Childhood Engineering Education, in: J.L. Polman, E.A. Kyza, D.K. O'Neill, I. Tabak, W.R. Penuel, A.S. Jurow, K. O'Connor, T. Lee, L. D'Amico (Eds.), Learning

and becoming in practice: The International Conference of the Learning Sciences (ICLS) 2014, Volume 2, International Society of the Learning Sciences, Boulder, Colorado, 2014, pp. 591–598. <https://repository.isls.org/handle/1/1168>

- [29] A. Collins, J.S. Brown, S.E. Newman, Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics, in: L.B. Resnick (Ed.), *Knowing, Learning, and Instruction*, first ed., Routledge, New York, 2018, 42 pages. <https://doi.org/10.4324/9781315044408-14>
- [30] M. Ally, F. Darroch, M. Toleman, A framework for understanding the factors influencing pair programming success, in: H. Baumeister, M. Marchesi, M. Holcombe (Eds.), *Extreme Programming and Agile Processes in Software Engineering. XP 2005. Lecture Notes in Computer Science vol 3556*, Springer, Berlin, Heidelberg, 2005, pp. 82–91. https://doi.org/10.1007/11499053_10
- [31] S. Campe, J. Denner, E. Green, D. Torres, D., Pair programming in middle school: variations in interactions and behaviors, *Computer Science Education* 30:1 (2020) 22–46. <https://doi.org/10.1080/08993408.2019.1648119>
- [32] R. Scherer, F. Siddiq, B. Sánchez Viveros, Learning to Code – Does it Help Students to Improve Their Thinking Skills?, in: S.C. Kong, D. Andone, G. Biswas, T. Crick, H.U. Hoppe, T.C. Hsu, R.H. Huang, K.Y. Li, C K. Looi, M. Milrad, J. Sheldon, J.L. Shih, K.F. Sin, M. Tissenbaum, J. Vahrenhold (Eds.), *Proceedings of the International Conference on Computational Thinking Education 2018 (CTE2018)*, The Education University of Hong Kong, Hong Kong, 2018, pp. 37–40.
- [33] N. Shah, C. Lewis, R. Caires, R., Analyzing equity in collaborative learning situations: a comparative case study in elementary computer science, in: J.L. Polman, E.A. Kyza, D. Kevin O’Neill, I. Tabak, W.R. Penuel, A.S. Jurow, K. O’Connor, T. Lee, L. D’Amico (Eds.), *Learning and Becoming in Practice: The International Conference of the Learning Sciences (ICLS) 2014 Volume 1*, International Society of the Learning Sciences, Colorado, 2014, pp. 495–502. <https://repository.isls.org/handle/1/1155>
- [34] J. Tsan, C.F. Lynch, K.E. Boyer, ‘Alright, what do we need?’: A study of young coders’ collaborative dialogue, *International Journal of Child-Computer Interaction* 17 (2018) 61–71. <https://doi.org/10.1016/j.ijcci.2018.03.001>
- [35] Z. Zakaria, D. Boulden, J. Vandenberg, J. Tsan, C.F. Lynch, E.N. Wiebe, Collaborative talk across two pair-programming configurations, in: K. Lund, G. Niccolai, E. Lavoué, C. Hmelo-Silver, G. Gweon, M. Baker (Eds.), *A Wide Lens: Combining*

- Embodied, Enactive, Extended, and Embedded Learning in Collaborative Settings, 13th International Conference on Computer Supported Collaborative Learning (CSCL) 2019, International Society of the Learning Sciences, Lyon, France 2019, pp. 224–231. <https://repository.isls.org/handle/1/1571>
- [36] J. Tsan, J. Vandenberg, Z. Zakaria, D.C. Boulden, C. Lynch, E. Wiebe, Collaborative Dialogue and Types of Conflict: An Analysis of Pair Programming Interactions between Upper Elementary Students, in: M. Sherriff, L.D. Merkle (Eds.), Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21), ACM, New York, 2021, pp. 1184–1190. <https://doi.org/10.1145/3408877.3432406>
- [37] J. Tsan, J. Vandenberg, Z. Zakaria, J.B. Wiggins, A.R. Webber, A. Bradbury, C. Lynch, E. Wiebe, K.E. Boyer, A Comparison of Two Pair Programming Configurations for Upper Elementary Students, in: J. Zhang, M. Sherriff (Eds.), Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20), ACM, New York, 2020, pp. 346–352. <https://doi.org/10.1145/3328778.3366941>
- [38] D. Franklin, P. Conrad, B. Boe, K. Nilsen, C. Hill, M. Len, G. Dreschler, G. Aldana, P. Almeida-Tanaka, B. Kiefer, C. Laird, F. Lopez, C. Pham, J. Suarez, R. Waite, Assessment of computer science learning in a scratch-based outreach program, in: T. Camp, P. Tymann (Eds.), Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13), ACM, New York, 2013, pp. 371–376. <https://doi.org/10.1145/2445196.2445304>
- [39] M. Ben-Ari, Constructivism in computer science education, SIGCSE Bulletin 30:1 (1998) 257–261. <https://doi.org/10.1145/274790.274308>
- [40] D.J. Gilmore, Models of debugging, Acta Psychologica 78:1–3 (1991) 151–172. [https://doi.org/10.1016/0001-6918\(91\)90009-O](https://doi.org/10.1016/0001-6918(91)90009-O)
- [41] R.K. Yin, Case Study Research Design and Methods, fifth ed., Sage, Thousand Oaks, 2012.
- [42] J. Fagerlund, P. Häkkinen, M. Vesisenaho, J. Viiri, Assessing 4th Grade Students' Computational Thinking through Scratch Programming Projects, Informatics in Education 19:4 (2020) 611–640. <https://doi.org/10.15388/infedu.2020.27>
- [43] T. Janík, T. Seidel, P. Najvar, Introduction: On the power of video studies in investigating teaching and learning, in: T. Janík, T. Seidel (Eds.), The Power of Video

Studies in Investigating Teaching and Learning in the Classroom, Waxmann Publishing, Münster, 2009, pp. 7–19.

- [44] S.J. Derry, R.D. Pea, B. Barron, R.A. Engle, F. Erickson, R. Goldman, R. Hall, T. Koschmann, J.L. Lemke, M.G. Sherin, B.L. Sherin, Conducting video research in the learning sciences: Guidance on selection, analysis, technology, and ethics, *The Journal of the Learning Sciences* 19:1 (2010) 3–53.
<https://doi.org/10.1080/10508400903452884>
- [45] H.E. Fischer, K. Neumann, Video analysis as a tool for understanding science instruction, in: D. Jorde, J. Dillon (Eds.), *Science Education Research and Practice in Europe. Cultural Perspectives in Science Education*, vol. 5, SensePublishers, Rotterdam, 2012, pp. 115–139. https://doi.org/10.1007/978-94-6091-900-8_6
- [46] R.E. Mayer, should there be a three-strikes rule against pure discovery learning? The case for guided methods of instruction, *American Psychologist* 59:1 (2004) 14-19.
<https://doi.org/10.1037/0003-066X.59.1.14>
- [47] J. Moreno-León, G. Robles, M. Román-González, Towards data-driven learning paths to develop computational thinking with Scratch, *IEEE Transactions on Emerging Topics in Computing* 8:1 (2017) pp. 193–205. <https://doi.org/10.1109/TETC.2017.2734818>
- [48] A. Funke, K. Geldreich, Measurement and visualization of programming processes of primary school students in Scratch, in: E. Barendsen, P. Hubwieser (Eds.), *Proceedings of the 12th Workshop on Primary and Secondary Computing Education (WiPSCE '17)*, ACM, New York, 2017, pp. 101–102.
<https://doi.org/10.1145/3137065.3137086>
- [49] S.-C. Kong, D. Lai, D. Sun, 2020. Teacher development in computational thinking: Design and learning outcomes of programming concepts, practices and pedagogy. *Computers & Education*. 151, 103872.
<https://doi.org/10.1016/j.compedu.2020.103872>
- [50] N. Carlborg, M. Tyrén, C. Heath, E. Eriksson, The scope of autonomy when teaching computational thinking in primary school, *International Journal of Child-Computer Interaction* 21 (2019) 130–139. <https://doi.org/10.1016/j.ijcci.2019.06.005>
- [51] A. Swidan, F. Hermans, M. Smit, Programming Misconceptions for School Students, in: L. Malmi, A. Korhonen, R. McCartney, A. Petersen (Eds.), *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18)*, ACM, New York, 2018, pp. 151–159. <https://doi.org/10.1145/3230977.3230995>