Defining and Evaluating Conflictive Animations for Programming Education: The Case of Jeliot ConAn

Andrés Moreno
School of Computing
University of Eastern Finland, Finland
amoreno@cs.uef.fi

Erkki Sutinen
School of Computing
University of Eastern Finland, Finland
sutinen@cs.uef.fi

Mike Joy
Department of Computer Science
University of Warwick, UK
m.s.joy@warwick.ac.uk

ABSTRACT

A review of the practical uses of errors in education reveals three contexts where errors have been shown to help: teaching conceptual knowledge, changing students' attitudes and promoting learning skills. Conflictive animations form a novel approach to teaching programming that follows a long tradition on research and development on program animation tools. Conflictive animations link the benefits of errors with program animation tools and programming education. This approach involves presenting to the students conflictive animations that do not animate faithfully the programs or concepts taught. Conflictive animations are versatile enough to cover the fundamental building blocks of programs such as operators, expressions and statements. With conflictive animations a novel set of learning activities can be introduced to computer science classes. This conflictive dimension of activities augments an engagement taxonomy for animation tools at all levels. They are an example of activities that promote critical thinking. A particular implementation of conflictive animations has been empirically evaluated aiming for ecological validity rather than statistical significance. Results indicate that students using conflictive animations improve their metacognitive skills, and, when compared to a control group, their conceptual knowledge improves at a better rate.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer Science Education*

Keywords

CS1; animation; programming; conflictive animation; constructivism

1. INTRODUCTION

While constructivism has a place in computer science education, it is still accepted that students should reliably acquire the foundational knowledge of computers and programming, and put aside their misunderstandings and misconceptions [1]. As a consequence, computer science educators and researchers have been trying to improve students' learning by creating new ways to interact and engage with the material they produce. In an effort to make the material more relevant, researchers have explored and developed the possibilities of using visualizations as learning material to present the dynamic aspects of programs and algorithms. With these tools educators can propose different activities to the students in order to engage with the material. As with books, animation tools show the execution of programs and algorithms following the experts' understanding of the underlying concepts [9, 2], which we assume to be correct.

Unfortunately, and despite the efforts of computer science educators, visualization tools do not appear to significantly improve the programming knowledge of first year students [7]. A meta-study of algorithm animation evaluations pointed out that the content of the visualizations are not as decisive for learning as the intended use of the visualizations [7]. In other words, "what" they see is second in learning to "how" they interact with what they see.

If we take the idea that what students see is not the key factor for the students to learn with animations, we can propose that students could just see an erroneous animation: now, the animation does not match the expert's representation. Errors, or errata, appear in learning material or information sources, and have been explicitly used before to foster learning. In this article, the roles of errors for learning are explored. Moreover, the use of errors may address problems previously identified in programming education and program visualization in particular.

SIGCSE'14, March 3–8, 2014, Atlanta, GA, USA.

ACM 978-1-4503-2605-6/14/03 ...$15.00.
http://dx.doi.org/10.1145/2538862.2538888 .

2. ROLES OF ERRORS IN EDUCATION

Postman [18] proposed using errors as the basis for a new teaching and learning practice. He argued that making students aware of human limits, and hence the occurrence of errors, could raise a generation of critical learners and thinkers. Students' awareness of frequent scientific errors and historical misrepresentations would prevent them from making assumptions about the truth of information presented to them, and force them to check the facts and theories built upon that information.

In a similar manner, Borasi [5] used errors as *springboards for inquiry.* In her exploratory research, students took part in activities that centered around errors, including those made by the students themselves and by teachers and mathematicians. Students were asked to reflect on those errors

and discuss possible corrections, and as a result of the experiment she determined that working with errors empowered students to think mathematically.

Constructivism follows a similar principle, its focus is on students constructing their own knowledge, and the teacher's mission is to guide students' learning. Researchers in constructivism have promoted cognitive conflict as a way to correct students' misconceptions, and this has been successfully used in physics education [10]. Students are asked to explain an empirical observation, and their incorrect explanations are challenged with further empirical observations that they cannot explain or refute. This should cause a cognitive conflict that allows for conceptual change, as students are ready for alternative explanations after their own have been shown to be wrong.

Große and Renkl [6] proposed and studied the use of worked examples — detailed descriptions about how a particular problem is solved — which contained errors, in order to improve understanding and problem solving skills. In their experiments, groups had to find errors in worked examples and correct them, and they concluded that correct and incorrect worked examples allowed for a more comprehensive understanding (*far transfer*), but only to students with prior knowledge. Students with little prior knowledge performed better only by using correct examples.

To summarize, errors have been used in education for improving three key aspects of learning: *conceptual knowledge* [19, 10, 6], *student skills* [6] and *student attitudes* [18, 5].

Roles of errors in learning programming.

Coping with uncertainty, errors and unexpected results is an important competence of all IT professionals throughout their careers: programmers correct their own errors while debugging, security researchers look for errors in other people's programs. When it comes to learning, studies have listed errors students make while programming, debugging or understanding programs [8]. However, the active use of errors in lectures and programming exercises has rarely been applied or researched in depth.

Cognitive conflict is one of the exceptions, as it has been applied to programming education. Ma *et al.* [11] proposed and evaluated an instructional design where cognitive conflict and animation take a central role in explaining value assignment and reference assignment. Their results indicated that cognitive conflict combined with correct animations leads to conceptual change. This effect was more evident with easier concepts, like simple assignment, than harder ones such as reference assignment.

Regarding programming skills, it is acknowledged that the practice of programming is non-linear. A programming solution requires substantial and frequent error correcting before reaching an acceptable state. Bennedsen and Caspersen [4] argue for the need to expose the programming process to students, highlighting how errors occur in the process and showing the way experts deal with them.

Finally, debugging is a fundamental skill that students are expected to acquire, and in which errors have a prominent role. Specific activities and tools have been designed to help them start debugging, but in most cases students develop the skill while working on other tasks. According to Perkins *et al.*et al. [16], tracing is a demanding task, and students often fail to follow the correct execution of the program and consequently do not diagnose the error.

3. PROGRAM VISUALIZATION AND ITS LIMITATIONS

Program visualization tools depict the steps taken by programs during execution. They illustrate and animate programming concepts, from basic programming constructs to object oriented design issues [2]. It is postulated that the use of multimodal presentation of information increases the level of comprehension by students [12]. In program visualization, animations present the abstract source code or algorithm in a graphical way, providing an alternative representation to a standard text document, and this presentation should help students to construct their own knowledge [1].

However, when working in new or complex fields such as programming, understanding the graphical clues provided by the visualizations requires a level of expertise not always found in learners [17]. To fully benefit from the animations or visualizations, students should make the effort to understand the graphical notation used. Thus, rather than easing the student's learning process, visualizations and animations can initially impose a new extraneous cognitive load. Vainio and Sajaniemi [21] even recommend programming teachers not to present diagrams to students, or ask them to draw their own, before they understand the concepts and the mapping between concepts and diagrams, lest the diagrams introduce more confusion to the students. Both Petre [17] and Vainio and Sajaniemi [21] seem to indicate that the use of graphical representations, animated or not, are not recommended in the case of novice programmers. This results in a vicious circle: educators would like to use graphical representations to help student understand programming concepts, but educators cannot because students do not understand yet those concepts.

However, empirical studies of algorithm animation and program visualization conclude that the educational tools can be beneficial for learning *under certain conditions* [7]. For example, in order to be beneficial, visualizations and animations should be used consistently during a course, and animations should increase in complexity as the students progress[3].

4. CONFLICTIVE ANIMATIONS

Moreno et al. [14] defined conflictive animations as those that deliberately may not reproduce what the animated code or algorithm is programmed to perform. An extreme example would be an animation of a different algorithm than the one presented; for example, a bubblesort algorithm that animates a heapsort algorithm instead. In contrast to the usage of animations proposed by Ma *et al.* [11] – solving a cognitive conflict – conflictive animations continually *create* a cognitive conflict, and it is the student's task to resolve it by consulting lecture materials, watching correct animations, or asking teachers and peers. The learning is now centered on the student.

Theoretically, conflictive animations serve the three key aspects of errors in learning identified in Section 2. The next subsections illustrate the possibilities that conflictive animations offer to conceptual knowledge of programming (Sec. 4.1), higher order thinking skills (Sec. 4.2), and attitudinal and motivational factors promoted by the use of animations (Sec. 4.3).

4.1 Programming Concepts in Conflictive Animations

A program is the largest construct that compilers and interpreters deal with, and can be the source for a conflict in an animation. However, smaller program components provide more opportunities to create conflictive animations that should help the student focus on fundamental programming concepts. Moreno *et al.* described the categories of concepts which can generate conflictive animations [13]. The categories were deduced from the main production rules of the Python grammar[1]. The chosen categories were: literal, operator, function call, expression, statement and module. For example, in the *statement* category, which determines the execution path and data flow, a conflictive animation of an *if-statement* can, for example, animate the *then-block* when the condition of the *if-statement* is *false*.

4.2 Promoting Programming Skills with Conflictive Animations

Programming skills cover a range of skills that are useful when practicing with the conceptual knowledge acquired. Conflictive animations may increase the student's metacognition, and reveal the student's lack of knowledge. The possibility of an error changes the way the student processes the graphical information, and forces the student to concentrate and to *decipher* the animation. This active attempt at deciphering the animation may increase the student's metacognition, and reveal the student's lack of knowledge.

Other programming skills which potentially benefit from conflictive animations are debugging and quality awareness. As the animation executes the program step by step, the student can learn how to trace the execution better than with normal program animation, in which the student can visualize the animation without stopping at each step.

4.3 Attitude Change with Conflictive Animations

Identifying an error can serve as a means for assessment, and a good one, as it requires proper understanding, but — as in Postman's vision [18] — it is important that students change their attitude towards the animation and their learning. They cannot be passive when they are told that the tool is cheating them, and they need to become in control of the animation. In turn, the barrier between the tool that produces the animation and the student is lowered. Now the tool can fail, and the student is in position to make a contribution.

This change of attitude may push students to make the effort to understand the animation and the concepts represented. This way we can break the vicious circle hinted by Petre [17] and elicited by Vainio and Sajaniemi [21]. Ironically, it would be the wrong information displayed that would make graphical representations and animations useful.

5. PEDAGOGICAL USES OF CONFLICTIVE ANIMATIONS

Naps *et al.* [15] introduced a taxonomy of engagement or interaction with algorithm animations. This taxonomy lists six levels of increasing engagement, 1)*no viewing*, 2)*viewing*, 3)*responding*, 4) *changing*, 5) *constructing*, and 6)*presenting*. However, the increasing engagement of the student does not always lead to increased grades [20]. This result may indicate that classical grading does not evaluate all the benefits of engaging educational tools. Moreno *et al.*[14] explained how conflictive animations add a new dimension to the engagement taxonomy as they trigger different activities not considered in the original taxonomy. For example, in the *responding* category students are asked to spot the error or errors in the animation. Identifying the error does not necessarily mean that the student has wholly grasped the concept, but at least that they have a functional mental model of the program execution or algorithm.

6. CREATING CONFLICTIVE ANIMATIONS

With the previous description of levels and pedagogical issues, specific applications of conflictive animations can be imagined or prepared in practical settings. Teachers can manually create animations with existing tools (e.g. [9]) or modify existing ones from publicly available repositories[2] to introduce errors in them and use them in their lectures.

Using conflictive animations at the responding and changing levels can especially improve students' attitudes. Activities at those levels are similar to puzzles where students have to find all the pieces or correct the errors. The construction and presenting levels can be extended further to allow for other gaming activities, either collaborative or competitive.

As described by Moreno *et al.*, automatic animation tools like Jeliot 3 [2] produce conflictive animations when new rules are introduced that modify the interpretation of programming constructs, i.e., creating a *conflictive interpreter* [13].

7. EVALUATION

To compare the effectiveness of conflictive animations with the normal animations in learning, a new version of Jeliot 3, called Jeliot ConAn, has been modified to produce them automatically [13]. The Jeliot family of animation tools is a successful lineage that has resulted in a wide range of solutions [2] to support learners in different contexts, e.g., at high schools, and universities, in e-learning. The evaluations carried out in those contexts have led the further development of the Jeliot family.

Jeliot 3 animates in its right-hand pane the execution of the code entered by the student. This way students can observe the steps taken by their programs, as illustrated in Figure 1. Jeliot ConAn shares the same user interface and animation graphics. It has been modified to misinterpret certain statements that result in conflictive animations pertaining to object-oriented concepts. As well, students can indicate the moment they think a conflict has happened by pressing a button. Thus, Jeliot ConAn engages students at the *responding* level with conflictive animations of *statements* and *function calls*.

To evaluate Jeliot ConAn and the impact of conflictive animations in programming, an empirical study within a programming course was performed. In the study, a subset of the conflictive animation characteristics was evaluated: the conflictive animations related to the *function call* conceptual level (see Section 4.1) and the *responding level* (Section 5) of the engagement taxonomy. Eighteen students (11 male,

[1]http://docs.python.org/2/reference/grammar.html

[2]http://www.animal.ahrgr.de/Anims/animations.php3

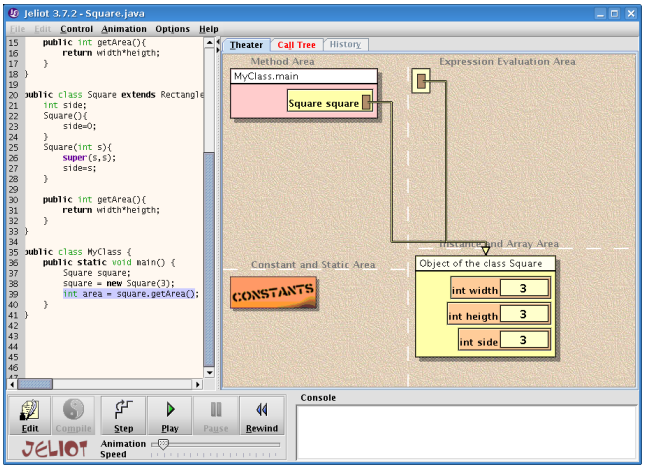

Figure 1: Screenshot of Jeliot. Showing one step in the object allocation process.

7 female) from an introductory programming course in Java volunteered to take part in the study. The students were taking part in an international Master's program in information technology and had very different backgrounds and programming knowledge. To ensure the ecological validity of the experiment, the experiment was performed during one of the practical sessions of the course, and after the lecture and practical session on the topic of object-oriented inheritance. Thus, the experiment would measure the knowledge, skills, and attitude of actual programming students in a realistic setting.

Students were randomly assigned to the control group (9 subjects) and to the experimental group (9 subjects), and the procedure was similar for both of them except for the visualization task. In the visualization task, the control group used Jeliot 3 to debug two programs, and the experimental group used Jeliot ConAn to find the conflict in two different animations. The programs dealt with the concept of inheritance and shared most of the code apart from the error students had to debug: thus, they were of similar difficulty. The visualization task duration was limited to 40 min.

Before the visualization task, students completed a questionnaire about their previous programming experience, and a pre-test with multiple-choice questions about inheritance. Afterwards, they completed several evaluation forms: 1) a post-test identical to the pre-test, 2) a graphical questionnaire with screenshots of the animations they had seen to assess their understanding of the animations and the concepts represented by them, and 3) a feedback form about the animation tool to assess their attitude regarding the tool. As well, the first author, who facilitated the experiment, took notes during the sessions regarding students' activities and interactions with the computer.

7.1 Results

Students were divided into five categories according to their reported experience with programming languages. The self-reported Java programming experience previous to the Java course was similar in both groups, with a majority having none or low previous experience in Java (8 out of 9 in both groups). For other programming languages the Jeliot 3 control group had a majority of students with medium to high experience (6 out of 9), while in the Jeliot ConAn experiment group a majority had low experience (7 out of 9) in other programming languages.

In the following results, non-parametric statistical tests for significance were run. However, due to the small size of the groups (N=18) and small differences, the results were not statistically significant.

Knowledge tests. The pre-test and post-test consisted of 10 multiple-choice questions. The test grade was obtained by giving one point per right answer and deducting half a point per wrong answer, in order to discourage students to randomly answer. Table 1 shows the average results for these tests in both groups.

The Jeliot ConAn group did improve the grade in the post-test (Wilcoxon Signed-rank test, p-value= 0.1). The Jeliot 3 group did not improve at all.

The grading test, the same for pre-test and post-test, contained questions related to the concepts that were the cause of the conflictive animations in the group using Jeliot ConAn: method overloading (2 questions) and constructors (3), and general object-oriented questions (5). To find out if Jeliot ConAn had any effect in the learning of those concepts for the experimental group, the pre-test and post-test were divided according to those three categories (method overloading, constructor, and general object-oriented knowledge). Results of this division are presented in Table 4.1. In the table, the third column, *maximum gained points possible*, is calculated by subtracting the total of points that all students can get for a given concept. The fourth column adds the increase in points for each student in the Jeliot ConAn group. The fifth column shows the p-value of the Wilcoxon Signed-rank test.

For the method overloading concept, there was no change in score after the intervention. Two out of nine students found the conflict present in the animation. For the constructor concept, three students increased their score, and three students found the conflict. Only one student increased their score and found the conflict for the constructor concept. In the general object-oriented knowledge group two students improved their score and one worsened.

Graphical questionnaires. The graphical questionnaire consisted of 11 questions related to the animation of an object allocation. Two points were awarded for each right answer, one for each answer that had some truth in it, and 0 if the answer was empty. One point was deducted for each wrong answer. The first five questions asked the students to describe the screenshots depicting the five main steps in the execution, and the other six questions focused on components of the animation. The average score result for this questionnaire was higher in the control group, 6.82 points, than in the experimental group, 4.82.

Previous programming experience shows a correlation with the graphical questionnaire results in both groups (Spearman's test: Jeliot 3 group p-value= 0.03; Jeliot ConAn group p-value=0.12). Expert students described better the animation than non-expert students. There is no correlation between the gain and the experience in any of the groups.

Feedback. The feedback form consisted of nine questions, five questions used a Likert scale to record the answers and

Table 1: Average and standard deviation of previous programming experience, pre-test, post-test, and the difference between those two (gain).

Group	Prog. experience	Pre-test	Post-test	Gain
Jeliot 3 (N=9)	2.73 (σ 1.56)	3.00 (σ 2.52)	2.89 (σ 2.31)	-0.11 (σ 1.34)
Jeliot ConAn (N=9)	3.11 (σ 1.36)	1.38 (σ 2.91)	2.16 (σ 2.54)	0.78 (σ 1.28)

Table 2: Results from the Jeliot ConAn group are analyzed regarding their improvement in the concepts demonstrated with conflictive animations.

Concept	Number of questions	Maximum gained points possible	Accumulated gained points	Number of students finding the conflict	p-value
Method Overloading	2	18	0	2	NA
Constructor	3	22	4	3	0.25
General object-oriented	5	20	1	NA	1

four were open questions. According to the responses, many more students from the control group felt that the animation tool had helped them to understand Java and inheritance, 89% from the control group and 44% from the experimental group. However, students from both groups disagreed with the fact that the tool had not had an influence in their Java knowledge (67% both groups), i.e., students thought the tool had a positive influence in their knowledge. Both groups of students from the experimental group felt that the animation tool was not hard to use (56% and 67%), but only 55% would like to have had more exercises using the tool. On the contrary, 89% of the students from the control group would have liked more debugging exercises with the tool.

When answering the open questions, students from both groups emphasized the need for textual information to complement the animation, and the need to simplify certain aspects of the visualization like the multiple references to the objects.

From the notes taken during the sessions, it was observed that two students did refer to the online resources that were available for them in case of need. Both students belonged to the experimental group.

7.2 Discussion

Conceptual knowledge. The results regarding the improvement of students' conceptual knowledge of the topics presented in the experiment point to a higher impact of the conflictive animations, even if not statistically significant. Students from the experimental group had improved their knowledge, if only by half a point, after interacting with the conflictive animations for 40 minutes.

When the results obtained by the experiment group were scrutinized more closely, the conflictive animations, which were at the conceptual level of "function call" (Sec 4.1), did not significantly improve students' post-test grades on the topics that were addressed, namely method overloading and inheritance. The slight improvement in inheritance understanding may be due to the fact that both conflictive animations tasks dealt with inheritance, although only one had one error related specifically to it, and students' exposure was double.

It is worth mentioning that the experiment tried to shed light on students' understanding of very subtle and complex execution steps of object-oriented programs. Future research should evaluate the impact of conflictive animations on different conceptual levels that are also prone to misconceptions, e.g., "statements" level concepts such as assignments. Even though Úrquiza-Fuentes and Velázquez-Iturbide [20] found that the impact of animations is higher in complex topics, this has not been the case, and there may be a barrier for *very* complex topics.

Surprisingly, the slight improvement in inheritance understanding from the experimental group did not correlate with a good understanding of the graphical metaphor that was evaluated by the graphical questionnaire: the control group understood better the static depictions of the animation according to their descriptions. The ability to describe in their own words the animation is correlated to the previous programming experience of the participants, no matter the intervention. This finding is consistent with Petre's research [17], in which novices have more problems understanding graphical notations. Also, the students that found the conflict did not automatically improve their grades.

Programming skills. On one hand, the de-learning effect of the control group after debugging, negative learning gain, contrasts with their high confidence in the beneficial effects of the tool in their learning. It highlights the problems with traditional program animation where learners do not assess their knowledge, and we can say that their meta-cognition is poor.

On the other hand, two students from the experimental group checked the lecture notes while doing the task. This indicates that the conflictive animations improved their metacognitive skills. They had been aware of something they did not understand and decided that they needed to correct that. No one from the control group stopped to check the learning material.

Attitude change. This short intervention cannot capture all the potential of conflictive animations for attitudinal change. In any case, the students showed a negative attitude towards the conflictive animation tool and the activities it promotes, 63% of experimental group versus 89% of thecontrol group asking for more activities with their version of the tool. This possibly reflects the discomfort produced by the new conflictive task, and it is an unexpected result, as we had imag-

ined the engagement and motivation would have been larger with the conflictive activities. A way to solve this would be to advance through the conflictive engagement taxonomy, and have students *viewing* some conflictive animations during the lectures before attempting to work on the *responding* level. As well, the Jeliot ConAn focuses too much on finding the error, i.e., being critical. A more constructive approach could benefit the attitude change towards conflictive animations and thus balance the critical thinking. Future research will evaluate the impact of working at the *constructive* level with conflictive animations.

8. CONCLUSION

The paper is a first step to understanding the consequences of using errors in computer science education. Conflictive animations form a flexible resource that can change the way computer science is taught. The conflictive animations in the *responding* category evaluated here have had a small impact in the students using them. However, results point to students improving their meta-cognitive skills and knowledge when compared to using normal animations for debugging. In any case, more research is needed to confirm those suggested effects. As well, more activities should be designed and evaluated to fully address the impact of conflictive animations in education.

Conflictive animations do not solve all the factors that partially explain the low retention rate and problematic mental models of students. Conflictive animations try to put within a computer science context ideas already used in other disciplines. Indirectly, conflictive animations can address the problems that current visualization systems and practices have by providing a different reason for students to look at the visualizations.

9. REFERENCES

[1] M. Ben-Ari. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–73, 2001.

[2] M. Ben-Ari, R. Bednarik, R. Ben-Bassat Levy, G. Ebel, A. Moreno, N. Myller, and E. Sutinen. A decade of research and development on program animation: The jeliot experience. *Journal of Visual Languages & Computing*, 22(5):375 – 384, 2011.

[3] R. Ben-Bassat Levy, M. Ben-Ari, and P. A. Uronen. The Jeliot 2000 program animation system. *Computers & Education*, 40(1):1 – 15, 2003.

[4] J. Bennedsen and M. E. Caspersen. Exposing the programming process. In J. Bennedsen, M. E. Caspersen, and M. Kölling, editors, *Reflection on the Teaching of Programming*. Springer, 2008.

[5] R. Borasi. Capitalizing on errors as "springboards for inquiry": A teaching experiment. *Journal for Research in Mathematics Education*, 25(2):166–208, 1994.

[6] C. S. Große and A. Renkl. Finding and fixing errors in worked examples: Can this foster learning outcomes? *Learning and Instruction*, 17(6):612–634, 2007.

[7] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290, 2002.

[8] J. Jackson, M. Cobb, and C. Carver. Identifying top java errors for novice programmers. In *Proceedings of the 35th ASEE/IEEE Frontiers in Education Conference*, 2005.

[9] V. Karavirta, A. Korhonen, L. Malmi, and K. Stalnacke. Matrixpro - a tool for demonstrating data structures and algorithms ex tempore. In *Proceedings of ICALT 2004*, pages 892–893, 2004.

[10] M. Limón. On the cognitive conflict as an instructional strategy for conceptual change: a critical appraisal. *Learning and Instruction*, 11(4–5):357–380, 2001.

[11] L. Ma, J. D. Ferguson, M. Roper, I. Ross, and M. Wood. Using cognitive conflict and visualisation to improve mental models held by novice programmers. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 342–346, New York, NY, USA, 2008. ACM.

[12] R. E. Mayer. *Multimedia Learning*. Cambridge University Press, 2001.

[13] A. Moreno, M. Joy, N. Myller, and E. Sutinen. Layered architecture for automatic generation of conflictive animations in programming education. *Learning Technologies, IEEE Transactions on*, 3(2):139–151, 2010.

[14] A. Moreno, E. Sutinen, R. Bednarik, and N. Myller. Conflictive animations as engaging learning tools. In R. Lister and Simon, editors, *Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *CRPIT*, pages 203–206, Koli National Park, Finland, 2007. ACS.

[15] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. Á. Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. In *ITiCSE-WGR '02: Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 131–152, New York, NY, USA, 2002. ACM Press.

[16] D. N. Perkins, C. Hancock, R. Hobbs, F. Martin, and R. Simmons. Conditions of learning in novice programmers. In J. C. Spohrer and E. I. Soloway, editors, *Studying the Novice Programmer*, pages 261–279. Ablex Publishing Company, 1989.

[17] M. Petre. Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, 1995.

[18] N. Postman. *The End of Education*, chapter The Fallen Angel. Vintage, 1996.

[19] J. P. Smith, III, A. A. diSessa, and J. Roschelle. Misconceptions reconceived: A constructivist analysis of knowledge in transition. *The Journal of the Learning Sciences*, 3(2):115–163, 1993-1994.

[20] J. Urquiza-Fuentes and J. Velázquez-Iturbide. Toward the effective use of educational program animations: The roles of student's engagement and topic complexity. *Computers & Education*, 67(0):178 – 192, 2013.

[21] V. Vainio and J. Sajaniemi. Factors in novice programmers' poor tracing skills. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education)*, 39(3):236–240, 2007.