A Review of Generic Program Visualization Systems for Introductory Programming Education

JUHA SORVA, VILLE KARAVIRTA, and LAURI MALMI, Aalto University

This article is a survey of program visualization systems intended for teaching beginners about the runtime behavior of computer programs. Our focus is on generic systems that are capable of illustrating many kinds of programs and behaviors. We inclusively describe such systems from the last three decades and review findings from their empirical evaluations. A comparable review on the topic does not previously exist; ours is intended to serve as a reference for the creators, evaluators, and users of educational program visualization systems. Moreover, we revisit the issue of learner engagement which has been identified as a potentially key factor in the success of educational software visualization and summarize what little is known about engagement in the context of the generic program visualization systems for beginners that we have reviewed; a proposed refinement of the frameworks previously used by computing education researchers to rank types of learner engagement is a side product of this effort. Overall, our review illustrates that program visualization systems for beginners are often short-lived research prototypes that support the user-controlled viewing of program animations; a recent trend is to support more engaging modes of user interaction. The results of evaluations largely support the use of program visualization in introductory programming education, but research to date is insufficient for drawing more nuanced conclusions with respect to learner engagement. On the basis of our review, we identify interesting questions to answer for future research in relation to themes such as engagement, the authenticity of learning tasks, cognitive load, and the integration of program visualization into introductory programming pedagogy.

Categories and Subject Descriptors: K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms: Human Factors

Additional Key Words and Phrases: Introductory programming education, CS1, software visualization, program visualization, program dynamics, notional machine, engagement taxonomy, literature review

ACM Reference Format:
Sorva, J., Karavirta, V., and Malmi, L. 2013. A review of generic program visualization systems for introductory programming education. *ACM Trans. Comput. Educ.* 13, 4, Article 15 (November 2013), 64 pages.
DOI:http://dx.doi.org/10.1145/2490822

1. INTRODUCTION

Over the past three decades, dozens of software systems have been developed whose purpose is to illustrate the runtime behavior of computer programs to beginner programmers. In this article, we review the computing education research (CER) literature in order to describe these systems and to survey what is known about them through empirical evaluations. To our knowledge, a comparable review on this topic does not previously exist; earlier reviews (e.g., Maletic et al. [2002]; Myers [1990]; Price et al. [1993]) have explored either algorithm visualization or software visualization more generally, and have sought only to provide illustrative examples of different

Authors' address: J. Sorva, V. Karavirta, and L. Malmi, Department of Computer Science and Engineering, Aalto University, P.O. Box 15400, FI-00076 AALTO, Finland; email: juha.sorva@aalto.fi.

kinds of systems rather than to present a more thorough exposition of a subfield as we do. Our review is intended to serve as a reference for the creators, evaluators, and users of educational program visualization systems.

We have structured the article as follows. Section 2 starts with some background on the challenges faced by beginners taking an introductory programming course (CS1) and on the use of visualization as a pedagogical tool in such courses. Section 3 positions our review within the field of software visualization and explicates the criteria we have used to include systems in our review and to exclude others. To provide additional context and structure for what comes later, Section 4 surveys a current debate in the CER community that concerns the importance of learners actively engaging with visualizations; we also present a framework that we have used for classifying modes of engagement. The lengthy Section 5 contains the review proper: descriptions of specific visualization systems and their empirical evaluations. In Section 6, we discuss the systems and the evaluations in overall terms and point at opportunities for future work. Section 7 briefly concludes the article.

This article has been adapted and extended from a chapter in the first author's doctoral thesis [Sorva 2012].

2. MOTIVATION

The creators of many program visualization systems have been motivated by the difficulties that beginners have with program dynamics and fundamental programming concepts. We will briefly outline some of these difficulties and then comment, in general terms, on the use of visualization to address them.

2.1. The Learning Challenge

Static perceptions of programming. Novices sometimes see particular programming concepts (e.g., objects, recursion) merely as pieces of code rather than as active components of a dynamic process that occurs at runtime [e.g., Eckerdal and Thuné 2005]. More generally, the entire endeavor of learning to program is sometimes perceived primarily as learning to write code in a particular language, and less, if at all, as learning to design behaviors to be executed by computers [Booth 1992; Thuné and Eckerdal 2010]. Learning to relate program code to the dynamics of program execution transforms the way novices reason about programs and consequently the way they practice programming [Sorva 2010].

Difficulties understanding the computer. A major challenge for the programming beginner is to come to grips with the so-called *notional machine* [du Boulay 1986], an abstraction of the computer in the role of executor of programs of a particular kind. Different programming languages and paradigms can be associated with different notional machines; the execution of a particular program can also be expressed in terms of different notional machines at different levels of abstraction.

Various authors have discussed the importance of the notional machine in terms of mental model theory [e.g., Ben-Ari 2001; Bruce-Lockhart and Norvell 2007; Jones 1992; Sorva 2013]. Ideally, mental models of the machine would be internally consistent and robustly reflect the principles that govern each machine component. However, as is typical of people's mental models of systems, novice programmers' intuitive understandings of the notional machine tend to be incomplete, unscientific, deficient, and lacking in firm boundaries. Often they are based on mere guesswork. Beginners are unsure of the capabilities of the computer, sometimes attributing human-like reasoning capabilities to it [Pea 1986; Ragonis and Ben-Ari 2005]. The way the computer needs each eventuality to be specified in detail (including, for instance, "obvious" `else` branches) is something that does not come naturally to many people [Miller 1981;

Pane et al. 2001]; novice programmers need to learn about what the computer takes care of and what the job of the programmer entails.

Misconceptions about fundamental programming constructs. The computing education research literature is rife with reports of the misconceptions that novice programmers harbor about fundamental programming constructs and concepts [see, e.g., Clancy 2004; Sorva 2012]. Many of these misconceptions concern the hidden runtime world of the notional machine and concepts that are not clearly visible in program code: references and pointers, object state, recursion, automatic updates to loop control variables, and so forth. Many of these topics also top various polls of students and teachers on difficult and/or important introductory programming topics [Goldman et al. 2008; Lahtinen et al. 2005; Milne and Rowe 2002; Schulte and Bennedsen 2006].

Struggles with tracing and program state. The difficulties that novice programmers have with tracing programs step by step have been reported in a much-cited multinational study [Lister et al. 2004] and elsewhere in the literature. Worryingly many students fail to understand statement sequencing to the extent that they cannot grasp a simple three-line swap of variable values [Simon 2011]. Tracing involves keeping track of program state, which is challenging to novices, perhaps in part because state is an everyday concept whose centrality to computer science is alien to non-programmers [Shinners-Kennedy 2008]. Studies of program comprehension have suggested that understanding state is an area that is particularly challenging to novices [Corritore and Wiedenbeck 1991].

In order to reason about changes in state—to "run" one's mental model of a notional machine—one needs to select the right "moving parts" to keep track of at an appropriate level of abstraction, but novices' inexperience leads them to make poor choices and to fail as a result of excessive cognitive load [Fitzgerald et al. 2008; Vainio and Sajaniemi 2007]. Novices, like expert programmers, can benefit from the external status representations provided by debuggers, but regular debuggers are not especially beginner-friendly (see Section 5.3.1).

In part because they have difficulty in doing so, and in part because they fail to see the reason for it, novices are often little inclined to trace their programs [see, e.g., Perkins et al. 1986]. Instead, novices may simply try to compare programs to familiar exemplars or simply guess—one article reports that students "often decided what the program would do on the basis of a few key statements" [Sleeman et al. 1986]. Novices also sometimes write template-based code that they cannot themselves trace [Thomas et al. 2004].

The reader can find a more detailed review of the beginners' difficulties with program dynamics and the notional machine in a companion article [Sorva 2013].

2.2. Visualizing a Notional Machine

Using pictures as clarification is common in textbooks and classrooms around the world, both within and outside computing education. According to one survey, most attendees of a computing education conference use visualizations of some sort in their teaching almost every day [Naps et al. 2003]. Various scholars within computing education research have advocated the visualization of the machine to support beginner programmers since the 1970s [see, e.g., Mayer 1975, 1976 and Figure 1]. A visualization of a notional machine can make focal, explicit, and controllable runtime processes that are hidden, implicit, and automatic when one's focus is on the program at the code level. Such visualizations can serve as a *conceptual models* that help learners construct viable programming knowledge. Depending on the learning goals (the target notional machine), program execution may be visualized at different levels of abstraction; Gries and Gries, for instance, used a visual memory model "that rises

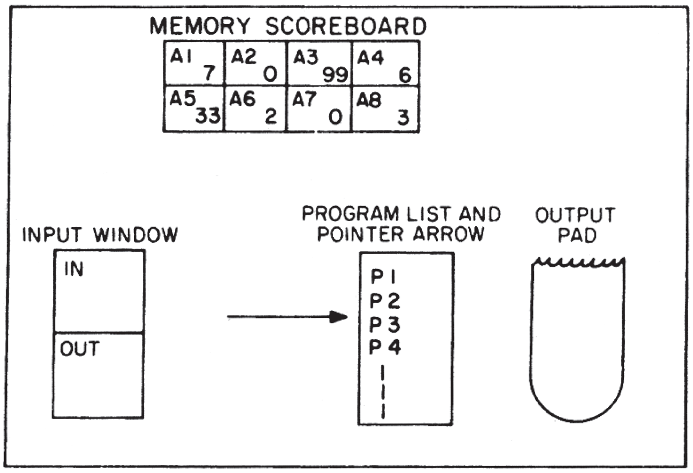


Fig. 1. A drawing of a notional machine presented to beginner programmers [Mayer 1976, 1981].

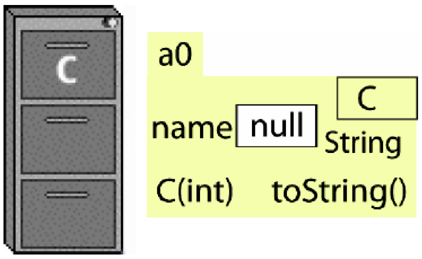
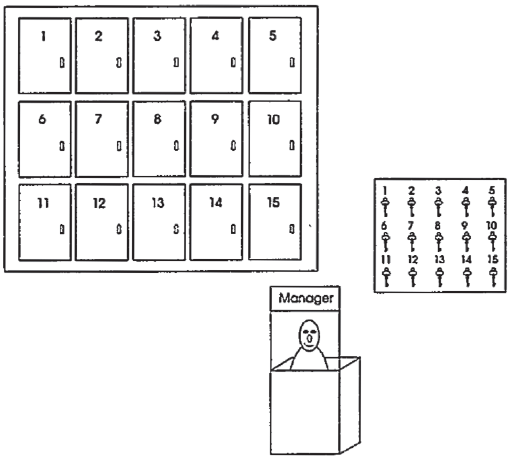


Fig. 2. A visualization of classes and objects [Gries 2008]. C is a class, represented as a file drawer. a0 is an identifier for an object of type C, represented as a manila folder.

above the computer and that is based on an analogy to which students can relate" [Gries 2008; Gries and Gries 2002, and see Figures 2 and 3].

The traditional way to use a visualization is for the teacher to complement their lectures and textbooks with pictures. Visualizations of a notional machine may be drawn on paper or blackboard or in presentation software. An alternative to relatively passive chalk-and-talk approaches is to involve the students; various teachers have experimented with having learners draw their own visualizations of program execution within a notional machine [e.g., Gries 2008; Gries and Gries 2002; Hertz and Jump 2013; Holliday and Luginbuhl 2004; Mselle 2011; Vagianou 2006]. Kinesthetic activities such as role-playing notional machine components in class are also sometimes used [e.g., Andrianoff and Levine 2002]. However, as some of the adopters of pen-and-paper visualization and related pedagogies have observed, student-drawn visualization of program dynamics has the weakness that drawing (and re-drawing) tends to take a lot of time and is often perceived as tedious by students [Gries et al. 2005; Vagianou 2006]. The time-consuming nature of drawing may also limit the number of visual examples that a teacher can expound in a lecture. To assist, computing educators have come up with a variety of software tools. These tools are the topic of our review in this article; the next section makes some further delimitations and positions our review within the broader field of software visualization.



memory pool	left-luggage room
memory manager	locker manager
pointer	keyring
memory cell	locker
address	key of a particular locker
list of available cells	board hooks with a key
`k1, k2: ^Type`	*k1* and *k2* are keyrings
`New(k1)`	ask for a locker, get its key and put it in keyring *k1*
`k1^`	open the locker whose key is in *k1* to access its contents
`k1 = NIL`	check whether *k1* is empty
`k1 = k2`	check whether *k1* and *k2* are both empty or both contain keys that open the same locker
`k1:= NIL`	leave empty keyring *k1*
`k1:= k2`	if *k2* is empty then leave *k1* empty, else make a copy of the key in *k2* and put it in *k1*
`Dispose(k1)`	free the locker whose key is in keyring *k1* hanging a copy of the key on its board hook

Fig. 3. A visual metaphor suggested by Jiménez-Peris et al. [1997] for object-oriented programming.

3. SCOPE OF THE REVIEW

Gračanin et al. [2005] define *software visualization* as follows.

> "The field of software visualization (SV) investigates approaches and techniques for static and dynamic graphical representations of algorithms, programs (code), and processed data. SV is concerned primarily with the analysis of programs and their development. The goal is to improve our understanding of inherently invisible and intangible software . . . The main challenge is to find effective mappings from different software aspects to graphical representations using visual metaphors."

As the field of software visualization is varied, many authors have tried to structure it by proposing classification systems and taxonomies of software visualization tools [see, e.g., Hundhausen et al. 2002; Lahtinen and Ahoniemi 2005; Maletic et al. 2002; Myers 1990; Naps et al. 2003; Price et al. 1993; Roman and Cox 1993; Stasko and Patterson 1992].[1] From a different perspective, Kelleher and Pausch [2005] laid out a classification of software environments for use in introductory programming education; the set of these systems overlaps with SV. In this section, we will use the classification systems of Maletic et al. and Kelleher and Pausch to give an overall feel for the field and to specify the scope of our review. Let us start, however, by considering some broad areas within software visualization.

Program visualization vs. algorithm visualization vs. visual programming vs. visual program simulation. Within software visualization, two broad subfields can be identified (Figure 4). *Algorithm visualization* (AV) systems visualize general algorithms (e.g., quicksort, binary tree operations), at a high level of abstraction, while *program visualization* (PV) systems visualize concrete programs, usually at a lower level.[2]

[1]Our use of the terms "taxonomy" and "classification" (or "classification system"), which are often used interchangeably, is based on that of Bloom [1956]. According to Bloom (p. 17), a *classification* serves a purpose if it is communicable and useful, while a *taxonomy* is intended to be validated through research. A taxonomy can also be used as a classification.

[2]In some of the older literature especially, "program visualization" refers to what we have just called software visualization, but at present, the terms are commonly used the way we use them here.

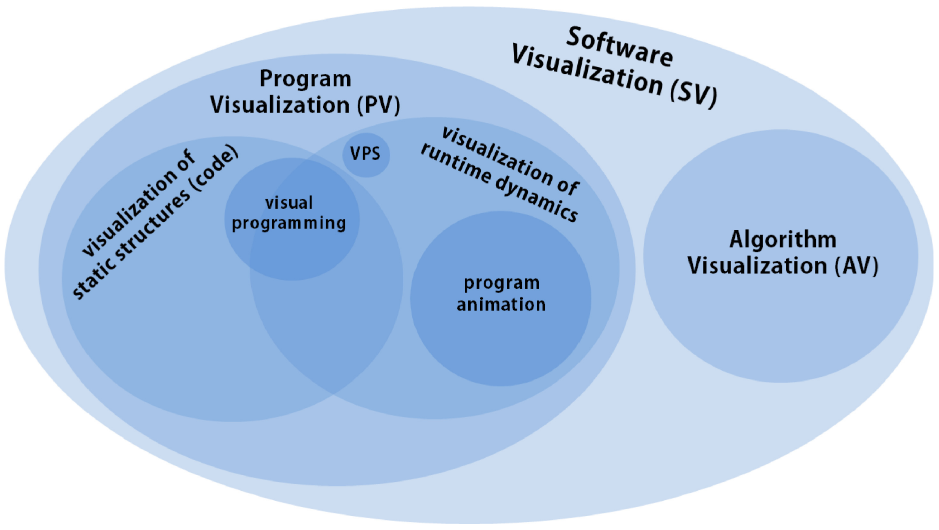

Software Visualization (SV)
Program Visualization (PV)
visualization of static structures (code)
VPS
visual programming
visualization of runtime dynamics
program animation
Algorithm Visualization (AV)

Fig. 4. Forms of software visualization [loosely adapted from Price et al. 1993]. The size of each area is not important. For the sake of simplicity, not all intersections are shown.

Within program visualization, some visualizations represent program code (e.g., dependencies or code evolution) while others illustrate the runtime dynamics of programs. *Program animation* refers to dynamic visualization of program dynamics—the common variety of program visualization in which the computer determines what happens during a program run, and visualizes this information to the user (e.g., as in a typical visual debugger). Also within program visualization, research on *visual programming* attempts to find new ways of *specifying* programs using graphics rather than visualizing software that is otherwise in non-graphical format. In *visual program simulation* (VPS) as defined by Sorva [2012], graphical elements serve a different purpose: the user "takes the role of the computer" and uses a visualization to learn and demonstrate what the computer does as it executes an existing program.[3]

Our review focuses on program visualization tools. Algorithm visualization tools operate at a level of abstraction that is too high to be interesting for learning about the fundamentals of program execution. We also do not cover visual programming. Although many visual programming environments do feature a facility for animating the execution of a visual program [see, e.g., Carlisle 2009; Scott et al. 2008], we have here focused on mainstream programming paradigms. We will, however, include a few selected systems with AV and visual programming functionality, which have additional features intended for teaching novices about the program dynamics of non-visual languages.

The classification of Maletic et al. Maletic et al. [2002] categorized software systems by their task (i.e., purpose), audience, target (i.e., type of content), representation, and medium. Even though Maletic et al. were themselves concerned with programming-in-the-large and not with education, their framework is well suited to explaining what part of the SV landscape we are interested in.

[3]Visual program simulation is a PV analogue of the *visual algorithm simulation* supported by the TRAKLA2 AV system [Korhonen et al. 2009a] in which learners simulate abstract algorithms by manipulating a visualization.

The *task* of the systems we review is to aid the learning and teaching of introductory programming, with an intended *audience* of novice programmers and CS1 teachers. This goal is different from the goal of many other SV systems, which seek to help (expert) programmers to learn about complex software systems. As for the *target* dimension, our review focuses on systems that visualize the execution-time dynamics of program runs using concrete values within a notional machine. This rules out, among other things, systems that only generate abstract (symbolic) control flow or data flow diagrams based on program code. Any form of *representation* will do as far as our review is concerned, as long as the *medium* is an electronic one and the visualization can be viewed onscreen. (We do include simple textual representations of variable values as a visualization of what happens within a program run.)

The classification of Kelleher and Pausch. Of the various kinds of programming languages and environments categorized by Kelleher and Pausch [2005], our focus is on what they called systems for *helping learners track program execution* and for *providing models of program execution* through metaphors or illustrations. This means that in terms of Kelleher and Pausch's scheme we rule the following to be out of scope.

— *Empowering systems*, which attempt to support the use of programming in pursuit of another goal (e.g., end-user programming, edutainment).
— *Learning support systems*, which try to ease the process of learning to program through means such as motivating contexts and social learning.
— Systems for *expressing programs*, which attempt to make it easier for beginners to express instructions to the computer.
— Systems for *structuring programs*, which attempt to facilitate the organization of instructions by changing the language or programming paradigm (e.g., Pascal, Smalltalk) or by making existing programming languages or paradigms more accessible (e.g., BlueJ).
— *Actors-in-microworlds* approaches, which make programming concrete by replacing a general-purpose programming language by a mini-language whose commands have a straightforward physical explanation in a virtual microworld.

Specialized vs. generic systems. We may also group systems that visualize program execution into specialized and generic systems. *Specialized systems* visualize the program dynamics related to a particular concept or construct (or very few). For instance, there are systems for visualizing parameter passing [e.g., Naps and Stenglein 1996], pointers [e.g., the visualizations embedded into the tutoring system by Kumar 2009], expression evaluation [Brusilovsky and Loboda 2006; Kumar 2005], using objects [e.g., the BlueJ object bench: Kölling 2008], recursion [e.g., Eskola and Tarhio 2002; Velázquez-Iturbide et al. 2008], and assignment [e.g., Ma 2007]. *Generic systems*, on the other hand, feature a notional machine that can deal with many language constructs and visualize various aspects of program execution.

Specialized systems have the advantage of being able to center on a topic. They may abstract out all other aspects of the notional machine for less clutter and a clear learning focus, and can also make use of visual tricks and metaphors that suit the particular topic especially well. Generic systems provide better coverage of content; a related benefit is that learners do not have to learn to use multiple different tools, many of which come with a learning curve of their own. Moreover, generic systems provide a big picture of a notional machine which a motley combination of specialized tools will not provide; this may help learners integrate their conceptual knowledge into a coherent whole. Generic systems also tend to admit students' own programs to be visualized, which is not so common in specialized systems.

In this review, we focus exclusively on generic systems.

Scope of the review, summarized. This article contains a review of generic program visualization systems in which the execution of programs that have been written in a general-purpose language in a traditional non-visual way is visualized onscreen in a manner suitable for novice programmers so that the system can be used to learn about program execution.

Within this scope, we have been inclusive; we list all the systems we are aware of rather than just providing examples.

Before turning to the actual systems, let us consider in some more detail the ways in which learners may use a visualization.

4. ENGAGING LEARNERS WITH SOFTWARE VISUALIZATIONS

"They will look at it and learn" thought many an enthusiastic programming teacher while putting together a visualization. But it might take more than that. Petre writes the following.

> "In considering representations for programming, the concern is formalisms, not art—precision, not breadth of interpretation. The implicit model behind at least some of the claims that graphical representations are superior to textual ones is that the programmer takes in a program in the same way that a viewer takes in a painting: by standing in front of it and soaking it in, letting the eye wander from place to place, receiving a 'gestalt' impression of the whole. But one purpose of programs is to present information clearly and unambiguously. Effective use requires purposeful perusal, not the unfettered, wandering eye of the casual art viewer." [Petre 1995]

The representational aspects of a visualization—its constituent parts and level of abstraction—are doubtless relevant to learning. These aspects can play a decisive part in defining what it is possible to learn from the visualization and what content the visualization is suitable for learning about. Yet although the careful design of a visualization is important, it is not the whole story. Even a visualization that has been painstakingly crafted to be as lucid as possible may fail to aid learning in practice. In the past decade, educators with an interest in SV have increasingly paid attention to how learners *engage* with visualizations.

In this section, we discuss the research on learner engagement in software visualization and the so-called engagement taxonomies that have been put forward in the CER literature (Subsection 4.1), before describing the adapted taxonomy that we ourselves use for structuring our review and describing how PV systems engage learners (Subsection 4.2). Subsection 4.3 then elaborates on why we thought it was worthwhile to suggest a new framework instead of using the existing ones.

4.1. Earlier Work on Learner Engagement

4.1.1. An Influential Meta-Study: Engage to Succeed. Naps comments on how attitudes to software visualization in computing education have changed during the past couple of decades.

> "As often happens with eye-catching new technologies, we have gone through what in retrospect were predictable phases: An initial, almost giddy, awe at the remarkable graphic effects, followed by disenchantment that the visualizations did not achieve their desired purpose in educational applications. Then we went through a period of empirical evaluation in which we began to sort out what worked and what did not." [Naps 2005]

The recent period of sobering reflection has been fueled by a meta-study of the effectiveness of algorithm visualizations by Hundhausen et al., who compared 24 earlier experiments to nutshell.

> "In sum, our meta-study suggests that AV technology is educationally effective, but not in the conventional way suggested by the old proverb 'a picture is worth 1,000 words'. Rather, according to our findings, the form of the learning exercise in which AV technology is used is actually more important than the quality of the visualizations produced by AV technology." [Hundhausen et al. 2002]

Hundhausen et al. observed that most of the successful AV experiments are predicted by the theory of personal constructivism (see, e.g., Larochelle et al. [1998]; Phillips [2000]), which emphasizes the need to actively engage with one's environment to construct one's subjective knowledge. Part of the benefit of active engagement may also come simply from the fact that learners end up spending more time with the visualization, increasing the time spent on studying.

Although the original work of Hundhausen et al. was done on the algorithm visualization side of the SV world, it has been hypothesized that the conclusions apply to program visualization as well. Compatible results have also been reported by researchers of educational visualization outside SV: videos and animations can be perceived as "easy" and their impact may be inferior to that of static images or even to not viewing a visualization at all, whereas interactive multimedia that cognitively engages the learner can lead to more elaborative processing (integration with prior knowledge) and consequently to better learning [see, e.g., Najjar 1998; Scheiter et al. 2006, and references therein].

4.1.2. Engagement Taxonomies: The OET and the EET. Drawing on the work of Hundhausen and his colleagues, a sizable working group of SV researchers put their heads together to differentiate between the ways in which visualization tools engage learners [Naps et al. 2003]. They presented a taxonomy whose levels describe increasingly engaging ways of interacting with a visualization: no viewing, viewing, responding, changing, constructing, and presenting. Naps et al. noted that the levels after viewing in their taxonomy, which we call the OET for *original engagement taxonomy*, do not form a strict hierarchy, but the authors did nevertheless hypothesize that using a visualization on a higher level of the taxonomy is likely to have a greater impact on learning than using it on one of the lower levels.

SV systems sometimes feature modes that engage students at different levels. Even a single activity may engage students at multiple levels of the taxonomy. Indeed, by definition, all activities at any but the no viewing level always involve viewing the visualization in addition to whatever else the student does. Naps et al. hypothesized that "more is better": a mix of different engagement levels leads to better learning.

Myller et al. [2009] presented a more fine-grained version of the engagement taxonomy. Their *extended engagement taxonomy* (EET) somewhat changes the definition of some of the original categories, and introduces four additional levels, resulting in: no viewing, viewing, controlled viewing, entering input, responding, changing, modifying, constructing, presenting, and reviewing. The research focus of Myller et al. was on program visualization and collaborative learning. They added to the hypotheses of Naps et al. by proposing that as the level of engagement between collaborating visualization users and the visualization increases, so does the communication and collaboration between the users.

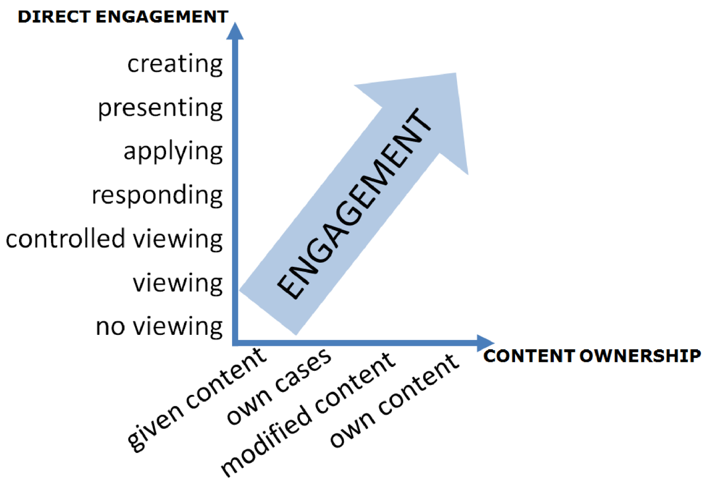


Fig. 5. The two dimensions of the 2DET engagement taxonomy.

There is limited empirical support for the engagement taxonomies. The authors of the OET have discussed how the reseach that inspired the taxonomy maps onto it [Naps et al. 2003]. Some studies since have been explicitly grounded in the OET or the EET (e.g., Korhonen et al. [2009b]; Laakso et al. [2009]; Lauer [2006]; Myller et al. [2007b, 2009]), and a meta-study was conducted by Urquiza-Fuentes and Velázquez-Iturbide [2009], who surveyed successful evaluations of software visualizations and related them to the OET. Overall, it can be said that although empirical work to date does tentatively support a few of the hypotheses built into the engagement taxonomies (in the context of AV especially), a solid general validation of the OET or the EET does not exist. (In Section 6, we will return to what can be said about learner engagement in the specific context of the PV tools that are the focus of our review.)

4.2. The 2DET

In order to structure our review and to help us produce a nuanced and yet succinct and systematic description of the learning activities supported by different program visualization systems, we have chosen to use an adaptation of the earlier engagement taxonomies.

The engagement taxonomy of Figure 5—for lack of a better name, let us call it *2DET*—has two dimensions. The first dimension, *direct engagement*, is concerned with the engagement that the learner has with the visualization itself. The second dimension, *content ownership*, is concerned with an indirect form of engagement that results from the learner's relationship with the target software, that is, the content of the visualization. Both dimensions contribute to the learner's overall engagement with the visualization, as shown in Figure 5.

Most of the levels along the two dimensions are similar to those in the OET and the EET, but we have modified some of the definitions and names. We have summarized the categories of the 2DET in Table I and will describe them in some more detail in the following sections.

4.2.1. The Direct Engagement Dimension. The direct engagement dimension consists of seven categories, or levels.

Table I. The Categories of the 2DET Engagement Taxonomy

	The direct engagement dimension	
#	**Level**	**Description**
1	No viewing	The learner does not view a visualization at all.
2	Viewing	The learner views a visualization with little or no control over how he does it.
3	Controlled viewing	The learner controls how he views a visualization, either before or during the viewing, for example, by changing animation speed or choosing which images or visual elements to examine.
4	Responding	The learner responds to questions about the target software, either while or after viewing a visualization of it.
5	Applying	The learner makes use of or modifies given visual components to perform some task, for example, direct manipulation of the visualization's components.
6	Presenting	The learner uses the visualization in order to present to others a detailed analysis or description of the visualization and/or the target software.
7	Creating	The learner creates a novel way to visualize the target software, for example, by drawing or programming or by combining given graphical primitives.

	The content ownership dimension	
#	**Level**	**Description**
1	Given content	The learner studies given software whose behavior is predefined.
2	Own cases	The learner studies given software but defines its input or other parameters either before or during execution.
3	Modified content	The learner studies given software that they can modify or have already modified.
4	Own content	The learner studies software that they created themselves.

On the first level, no viewing, no visualization is used. Examples: studying source code; reading a textual description of how an example program works.

When viewing, the learner has no control over how he views the visualization. Examples: watching a video or non-stop animation of a dynamic process without being able to control its pacing; watching someone else use a visual debugger.

On the controlled viewing level, the learner plays a part in deciding what he or she sees. He may change the pace of the visualization or choose which part of a visualization to explore. Examples: stepping through a program in a visual debugger; viewing a sequence of program state diagrams printed on paper; choosing which variables to show in a visual program trace (either before or during execution); navigating an avatar through a virtual 3D world.

When responding to a visualization, the learner uses the visualization to answer questions presented to him either while or after he views it. Examples: What will the value of that variable be after the next line is executed? What is the time complexity of this program?

A learner applies a visualization when he makes use of or modifies given visual components to perform some task related to the target software. The task may be to carry out some procedure or algorithm, to illustrate a specific case, or to create a piece of software, for instance. Examples are simulating the stages of an algorithm using a given visualization of a data structure, using given elements to record a visualization of how the evaluation of an expression proceeds, using given visual components to produce a piece of code, and annotating a program's source code so that a visualization of key stages is produced when it is run.

When engaging on the presenting level, the learner makes use of a visualization to present an analysis or description to others. What is presented can concern either the visualization or the target software. The presentation task has to be demanding enough to require significant reflection and a detailed consideration of the visualization on the part of the presenter. Examples are an in-class presentation of a visualization and the program it represents and a peer review of a visualization created by another student.

Finally, creating a visualization means designing a novel way to visualize some content. Examples are drawing a visualization of a given program or a self-written program, producing a visualization by combining given graphical primitives previously void of meaning in the domain of the content, and writing a program that visualizes software.

4.2.2. The Content Ownership Dimension. There are four categories on the content ownership dimension.

Given content means that the learner engages with a visualization of software that they have not produced themselves and whose behavior they do not significantly affect. Examples are a program that does not expect user input and has been provided by a teacher or chosen from a selection of predefined examples in an SV tool's library, and a ready-made program to which the learner only provides some relatively arbitrary input (e.g., a program that prompts the learner for their name and prints it out).

Own cases is defined as above, except that the learner can choose inputs or other parameters that significantly affect what the target software does. Examples are choosing a data set for an algorithm and entering inputs that affect loop termination during a program visualization that illustrates control flow.

Modified content means that the learner engages with a visualization of given software which they have already modified themselves or may modify while using the visualization.

Own content means that the learner engages with a visualization of software that they wrote themselves. An example is using a visualization tool to examine one's programming project. (N.B. This is distinct from creating on the direct engagement dimension, in which the learner creates a visualization.)

4.3. Why a New Taxonomy?

We feel that the 2DET allows a clearer and richer expression of modes of visualization use than the OET and the EET. This has been our primary motivation in introducing it, and in this article, we use the taxonomy mainly as a classification tool that makes it more convenient to describe the systems we review. That said, we do expect that both kinds of engagement highlighted by the two dimensions of the 2DET can help bring about the "purposeful perusal" of program visualizations that Petre called for (Section 4), making the 2DET potentially useful in future research efforts. We will now provide a brief rationale for preferring the 2DET.

4.3.1. Content Ownership as a Separate Dimension. We believe that content ownership—a learner's involvement with the content of a visualization—is an aspect of engagement that merits greater explicit consideration in the context of educational visualization.

Increased content ownership can lead to better learning simply through a greater time investment and higher cognitive effort. A learner who has not merely studied a given example but has instead designed their own use cases or created the content from scratch is likely to be more familiar with the content and more capable of mapping the visualization to the content that it represents.

We also see a strong motivational aspect in the content ownership dimension of the 2DET. Whereas the direct engagement dimension represents increasing activity in task requirements, content ownership is related to the reasons why a learner cares about what is being visualized and consequently to learners' willingness to cognitively engage with visualizations. For example, we would expect that a student is typically more likely to care about (and learn from) a visualization of, say, a buggy program they created themselves than about a visualization of a given example program that they are supposed to debug. Various learning theories can be read as supporting this hypothesis, including those constructivist ones which emphasize learners' participation in shaping the learning process. For instance, the constructionist theory of Papert [1993] sees learners as makers whose desire to build things (such as programs) can be harnessed for better learning.

For the most part, the earlier work on the engagement taxonomies ignores or finesses the issue of content ownership. The EET does introduce two categories—entering input and modifying—that are defined in terms of the learner's relationship with the target software. This, we feel, is helpful and has inspired our present work. However, the EET conflates two dimensions that we think are best analyzed separately. Our two dimensions represent two different aspects of engagement that are intuitive and that are always present simultaneously in any mode of engagement with a visualization. We also find that it is difficult to come up with convincing justifications for placing the categories involving content ownership within a single-dimensional framework; for instance, it is challenging to find strong analytical arguments for the locations of entering input and modifying in the EET.

4.3.2. Other Issues with the Taxonomies. Creating a visualization from scratch or from primitive components is a cognitively demanding task, which requires a kind of reflection on the properties of the visualization that is distinct in nature from what is required to manipulate a given visualization or to add annotations (of given kinds) to source code. However, all these activities would belong on the rather crowded constructing level of the OET. The 2DET separates these two forms of engagement as creating and applying [cf. Lauer 2006].

While the EET is more nuanced in this respect than the OET, some specific details of the category system seem dubious. For instance, are there reasons to believe that providing input while viewing a program run (which counts as entering input) is generally and substantially a less engaging activity than providing an input set before viewing (which counts as modifying)? We have tried to word the category definitions of the 2DET to address some issues of this kind.

The EET features separate categories for presenting work to others and reviewing the work of others. To us, both appear to be tasks of analysis and communication that are more or less equally engaging, so we have used a single category for both activities.[4]

The ordering of categories above controlled viewing along the direct engagement dimension of the 2DET can be seen as loosely analogous to the revised Bloom's taxonomy by Anderson et al. [2001], and indeed our choice of terms has been influenced by this well-established framework. One criticism of the engagement taxonomies is also analogous to a criticism of Bloom's taxonomy (made, e.g., by Hattie and Purdie [1998]): a particular kind of task does not necessarily lead to a certain kind of (cognitive) response. For instance, a student may provide only a surface-level answer to a profound question. Another problem concerns the responding category especially: there are many different kinds of questions, some of which call for considerably more cognitive effort than others. The 2DET as presented does not fix this issue.

5. THE SYSTEMS

This section describes a number of program visualization tools for introductory programming education, as delimited in Section 3 above. A summary of the tools appears

[4]Oechsle and Morth, whose work inspired the addition of reviewing at the top of the EET, in fact put reviewing in place of, rather than on top of, presenting in their adaptation of the OET [Oechsle and Morth 2007].

Table II. A Summary of Selected Program Visualization Systems (Part 1/2)

System name (or author)	**Page**	**At least since**	**Status**	**Overall purpose**	**Paradigm**	**Language**	**Eval.**
LOPLE / DynaMOD / DynaLab	19	1983	inactive	examples	imp	Pascal, [Ada], [C]	S
EROSI	20	1996	inactive	examples	imp	Pascal	Q, S
(Fernández et al.)	20	1998	inactive	examples	OO	Smalltalk	none
PlanAni	21	2002	active?	examples	imp	Pascal, Java, C, Python	E, Q
Metaphor-based OO visualizer	22	2007	active?	examples	OO	Java	E
(Miyadera et al.)	22	2006	inactive	examples	imp	C_{SS}	?
CSmart	22	2011	active	assignments	imp	C	S
regular visual debuggers	25	varies	varies	debug/tr	varies	varies	varies
Basic Programming	26	1979	inactive	develop	imp	BASIC	?
Amethyst	26	1988	inactive	debug/tr	imp	Pascal	none
DISCOVER	26	1992	inactive	develop	imp	pseudocode	E
VisMod	28	1996	inactive	develop, debug/tr, [AV]	imp	Modula-2	A
Bradman	28	1991	inactive	debug/tr	imp	C	E
(Korsh et al.)	29	1997	inactive	debug/tr, AV	imp	$C{+}{+}_{SS}$	none
VINCE	29	1998	inactive	debug/tr	imp	C	E, S
OGRE	30	2004	inactive?	debug/tr	imp, OO	C++	E, S
VIP	30	2005	active	debug/tr	imp	$C{+}{+}_{SS}$	E, S, Q
JAVAVIS	33	2002	inactive	debug/tr	imp, OO	Java	A
(Seppälä)	33	2003	inactive	debug/tr	imp, OO	Java	none
OOP-Anim	33	2003	inactive	debug/tr	imp, OO	Java	none
JavaMod	33	2004	inactive	debug/tr	imp, OO	Java	none
JIVE	34	2002	active	debug/tr	imp, OO	Java	A
Memview	34	2004	inactive?	debug/tr	imp, OO	Java	A
CoffeeDregs	34	2009	active	examples	imp, OO	Java	A

in Tables II and IV, which give an overview of each system, and Tables III and V, which go into more detail about the systems' functionality and empirical evaluations. The rest of this long section is structured primarily in terms of the support that the systems have for the direct engagement dimension of the 2DET. More specifically, this section consists of the following sections.

5.1 A legend for Tables II through V.
5.2 Systems that support at most the controlled viewing level of the direct engagement dimension and which do not support own content. That is, systems that are meant for automatically visualizing the execution of teacher-given example programs (with possible user modifications).
5.3 Systems that support the controlled viewing level of the direct engagement dimension, and possibly responding, but no higher, and that do support own content. That is, systems that provide automatic, controllable visualizations for learners' own programs (and maybe also embedded questions). More than half of the systems reviewed fall in this category.
5.4 Systems that support some form of applying a visualization. (Some of these systems also feature other forms of engagement that are ranked lower in the 2DET, such as responding or controlled viewing.)

Table III. A Summary of How Selected PV Systems Present Notional Machines and Engage Learners (Part 1/2)

System name (or author)	Notional machine elements	Representation	Step grain	Direct engagement	Content ownership
LOPLE / DynaMOD / DynaLab	Control, Vars, Calls	symbols	S	ctrl'd viewing	own cases, [own content]
EROSI	Control, Vars, Calls	abstract 2D, [audio]	S	ctrl'd viewing	given content
PlanAni	Control, Vars, ExprEv, Structs	visual metaphors, smooth animation, explanations	E	ctrl'd viewing	own cases
(Fernández et al.)	Refs, Objs, Classes, Calls	abstract 2D, visual metaphors	MP	ctrl'd viewing	modified content
Metaphor-based OO visualizer	Control, Vars, ExprEv, Refs, Objs, Classes, Calls, Structs	visual metaphors, smooth animation, explanations	E	ctrl'd viewing	own cases
(Miyadera et al.)	Control, Vars, ExprEv, Calls	abstract 2D	S?	ctrl'd viewing	given content
CSmart	Control, Vars, ExprEv	explanations, visual metaphors, audio	S	ctrl'd viewing	given content
regular visual debuggers	Control, Vars, Calls, Objs, Refs, Structs (e.g.)	standard widgets	S	ctrl'd viewing	own content
Basic Programming	Control, Vars, ExprEv	symbols	E	ctrl'd viewing	own content
Amethyst	Control, Vars, Calls, Structs, [Refs]	abstract 2D	S?	ctrl'd viewing	own content
DISCOVER	Control, Vars	abstract 2D	S	ctrl'd viewing	own content
VisMod	Control, Vars, Refs, Calls, Structs	standard widgets, abstract 2D	S	ctrl'd viewing	own content
Bradman	Control, Vars, ExprEv, Refs	symbols, explanations, [abstract 2D], [smooth animation]	S	ctrl'd viewing	own content
(Korsh et al.)	Control, Vars, Refs, Calls, ExprEv, Structs	abstract 2D	E	ctrl'd viewing	own content
VINCE	Control, Vars, Refs, Calls, Addrs, Structs	abstract 2D, explanations	S	ctrl'd viewing	own content
OGRE	Control, Vars, Refs, Objs, Classes, Calls, Structs	abstract 3D, smooth animation, explanations	S	ctrl'd viewing	own content
VIP	Control, Vars, ExprEv, Calls, Refs, Structs	standard widgets, explanations, [abstract 2D]	S	ctrl'd viewing	own content
JAVAVIS	Vars, Refs, Objs, Calls, Structs	abstract 2D, UML	S	ctrl'd viewing	own content
(Seppälä)	Control, Vars, Refs, Objs, Calls	abstract 2D	S	ctrl'd viewing	own content
OOP-Anim	Control, Vars, Refs, Objs, Classes	abstract 2D, smooth animation, explanations	S	ctrl'd viewing	own content
JavaMod	Control, Vars, ExprEv, Refs, Objs, Calls, Structs	standard widgets, UML	E	ctrl'd viewing	own content
JIVE	Control, Vars, Refs, Objs, Classes, Calls, Structs	standard widgets, abstract 2D	S	ctrl'd viewing	own content
Memview	Control, Vars, Refs, Objs, Classes, Calls, Structs	standard widgets	S	ctrl'd viewing	own content
CoffeeDregs	Control, Vars, Refs, Objs, Classes, Calls, Structs	abstract 2D	S	ctrl'd viewing	own content

Table IV. A Summary of Selected Program Visualization Systems (Part 2/2)

System name (or author)	Page	At least since	Status	Overall purpose	Paradigm	Language	Eval.
JavaTool	34	2008	active	develop, assignments	imp	Java_{SS}	none
EVizor	35	2009	active	develop, assignments	imp, OO	Java	E, S
Eliot / Jeliot I	35	1996	inactive	debug/tr, AV	imp, [func]	C, Java_{SS}, [Scheme]	S, Q
Jeliot 2000 / Jeliot 3	36	2003	active	debug/tr, [develop], [assignments]	imp, OO	Java, [C], [Python]	E, S, Q
GRASP / jGRASP	39	1996	active	develop, debug/tr, AV	imp, OO	Java, C, C++, Objective-C, Ada, VHDL	E (on AV), A
The Teaching Machine	39	2000	active	debug/tr, AV, [assignments]	imp, OO	C++, Java	S
ETV	40	2000	inactive	debug/tr	imp	C, Java, Perl, Lisp	A
HDPV	40	2008	inactive	debug/tr, AV	imp, OO	C, C++, Java	none
Jype	40	2009	inactive?	develop, assignments, AV	imp, OO	Python	A
Online Python Tutor	41	2010	active	debug/tr, assignments	imp, OO	Python	none
typical functional debuggers	41	varies	varies	debug/tr	func	varies	varies
ZStep95	41	1994	inactive	debug/tr	func	Lisp_{SS}	none
(Kasmarik and Thurbon)	42	2000	inactive	debug/tr?	imp, OO	Java	E
CMeRun	42	2004	inactive	debug/tr	imp	C++_{SS}	A
Backstop	43	2007	inactive	debug/tr	imp	Java_{SS}	S
(Gilligan)	43	1998	inactive	develop	imp, [OO]	Pascal, [Java]	none
ViRPlay3D2	44	2008	unknown	SW design, examples	OO	CRC cards	none
(Dönmez and İnceoğlu)	45	2008	unknown	examples, assignments	imp	$\text{C\#}_{\text{SS}}$	none
Online Tutoring System	45	2010	inactive	assignments	imp	VBA_{SS}	E, S
UUhistle	46	2009	active	assignments, debug/tr	imp, OO	Python, [Java]	E, S, Q
ViLLE	48	2005	active	examples, assignments	imp	$\text{various}_{\text{SS}}$, pseudocode	E, S
WinHIPE	49	1998	active	debug/tr, develop	func	Hope	E, S

5.5 As an addendum, a brief commentary on low-level (e.g., machine code) approaches to program visualization (whichever their engagement level). The systems mentioned here do not appear in the tables.

As a result of the prevalence of controlled viewing and, to a lesser extent, applying in existing tools, these subsections cover all the tools within scope that we are aware of.

5.1. Legend for Tables II through V

In several columns of those tables, square brackets mark features that are peripheral, untried, or at an early stage of implementation.

System name (or author). The name of the system. Closely related systems are grouped together as one item. Systems without a published name have the authors' names in parentheses instead.

Table V. A Summary of How Selected PV Systems Present Notional Machines and Engage Learners (Part 2/2)

System name (or author)	Notional machine elements	Representation	Step grain	Direct engagement	Content ownership
JavaTool	Control, Vars, Structs	abstract 2D	S	ctrl'd viewing	own content
EVizor	Control, Vars, Calls, Refs, Objs, Classes, Structs	abstract 2D, smooth animation, explanations	S	ctrl'd viewing, responding	own content / given content
Eliot / Jeliot I	Control, Vars, [ExprEv], Structs	abstract 2D, smooth animation	event-based	ctrl'd viewing	own content
Jeliot 2000 / Jeliot 3	Control, Vars, ExprEv, Calls, Refs, Objs, Classes, Structs	abstract 2D, smooth animation	E	ctrl'd viewing, [responding]	own content
GRASP / jGRASP	Control, Vars, Calls, Refs, Objs, Structs	standard widgets, abstract 2D, smooth animation	S	ctrl'd viewing	own content
The Teaching Machine	Control, Vars, ExprEv, Calls, Addrs, Refs, Objs, Structs	standard widgets, abstract 2D	E	ctrl'd viewing	own content
ETV	Control, Vars, Calls	standard widgets	S	ctrl'd viewing	own content
HDPV	Control, Vars, Calls, Refs, Objs, Structs	abstract 2D	E	ctrl'd viewing	own content
Jype	Control, Vars, Objs, Refs, Calls, Structs	standard widgets, abstract 2D	S	ctrl'd viewing	own content
Online Python Tutor	Control, Vars, Objs, Classes, Refs, Calls, Structs	abstract 2D	S	ctrl'd viewing	own content
typical functional debuggers	Control, Vars, ExpEv (e.g.)	standard widgets, abstract 2D	E	ctrl'd viewing	own content
ZStep95	Control, Vars, ExprEv	standard widgets, abstract 2D	E	ctrl'd viewing	own content
(Kasmarik and Thurbon)	Control, Vars, Refs, Calls, Objs, Structs	abstract 2D	S	ctrl'd viewing	own content?
CMeRun	Control, Vars, ExprEv	text	S	ctrl'd viewing	own content
Backstop	Control, Vars, ExprEv	text	S	ctrl'd viewing	own content
(Gilligan)	Control, Vars, ExprEv, Calls, Structs, [Objs], [Classes], [Refs]	visual metaphors, standard widgets	E	applying	own content
ViRPlay3D2	Vars, Objs, Classes, Refs, Calls	virtual 3D world	MP	ctrl'd viewing, applying	own content / given content
(Dönmez and İnceoğlu)	Control, Vars, ExprEv	standard widgets	E	applying	own code
Online Tutoring System	Control, Vars, ExprEv	standard widgets, explanations	E	applying	given content
UUhistle	Control, Vars, ExprEv, Calls, Refs, Objs, Classes, Structs	abstract 2D, smooth animation, explanations	E	ctrl'd viewing, responding, applying	own content / given content
ViLLE	Control, Vars, Calls, Structs	standard widgets, explanations	S	ctrl'd viewing, responding, applying	given content, [modified content]
WinHIPE	Control, Vars, ExprEv, Refs, Calls, Structs	abstract 2D	E	ctrl'd viewing, applying	own content / given content

Page. The page within this article where the description of the system begins. The entries in Tables II to V are sorted by this number.

At least since. The year when the system was first used, its first version was released, or the first article on the system was published. May not be accurate, but gives an idea of when the system came into being. The years listed in this column are the basis of the timeline in Figure 6.

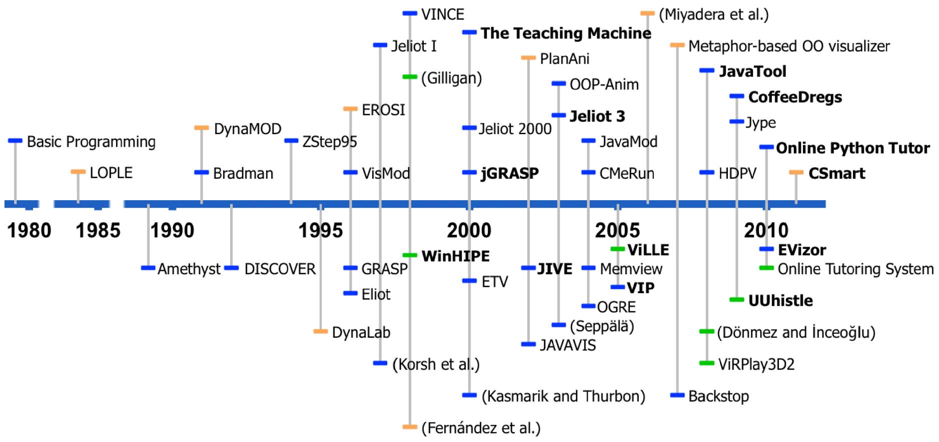


Fig. 6. A timeline of the program visualization systems reviewed. The years are approximations based on the first publications and system versions that we are aware of. The names in bold face indicate currently active systems, that is, projects whose recent activity we have found evidence of. The colors correspond to modes of engagement supported by the systems: orange means the controlled viewing of examples (Section 5.2) and blue of own content (Section 5.3); green is used for systems that support some form of applying a visualization (Section 5.4).

Status. Our understanding—or best guess—based on web searches and/or personal communication, of whether the system is still being used in teaching, maintained, and/or developed. May be inaccurate.

Overall purpose. The overall purpose of the system: what students can do with it. *Examples* is listed when the system is intended for studying given examples only. *Debug/tr* refers to any stepwise tracing of user code (possibly for the purpose of finding bugs); it subsumes *examples*. *Develop* subsumes debugging, and means that the system is intended to be used as a software development environment. *Assignments* means that the tool facilitates an assignment type—for example, multiple-choice questions or visual program simulation—that is not a part of a programmer's usual routine of read/test/debug/modify/write/design code. Although this review does not include pure algorithm visualization systems, we mention AV as a goal of AV/PV hybrid systems.

Paradigm. The programming paradigms that the software can primarily help learn about. *Imp* is used for imperative and procedural programming, which may include the occasional or implicit use of objects such as arrays. *OO* stands for richer object-oriented programming, and *func* for functional programming.

Language. The programming language(s) in which the programs being visualized (the target software) are written. For systems which visualize user-written programs, we have used the subscript $_{SS}$ to mean that only a significantly limited subset of a language is available for use (e.g., functions or object-orientation are missing). Lesser limitations are common and not marked.

Evaluation. The types of empirical evaluations of the system in the context of introductory programming, to the best of our knowledge. *Anecdotal* (A) means that only anecdotal evidence has been reported about student and/or teacher experiences with the system (but this still implies that the system has been used in actual practice). *Experimental* (E) refers to quantitative, controlled experiments or quasi-experiments. *Survey* (S) refers to producing descriptive statistics and/or quotes from systematically collected user feedback or other data. *Qualitative* (Q) refers to rigorous qualitative research (e.g., grounded theory, phenomenography).

Notional machine elements. What the visualization covers: a non-exhaustive list of key elements of the notional machine visualized by the system. *Control* refers to control flow: the system makes explicit which part of program code is active at which stage. *Vars* stands for variables. *ExprEv* means expression evaluation. *Calls* refers to the sequencing of function/procedure/method calls and returns (which may or may not be expressed in terms of call stacks). *Refs* stands for references and/or pointers, *Addrs* for memory addresses. *Objs* is short for objects. *Classes* means that the system visualizes classes not only as source code or a static class diagram, but as a part of the program runtime. *Structs* refers generally to any composite data—arrays, records, lists, trees, and the like—that has a bespoke representation within the system. All these categories are our own abstractions. They are realized in different ways in different systems. It has not been possible to try out all the systems; the descriptions given are an interpretation of the descriptions and images in the literature.

Representation. What the visualization primarily consists of, on the surface: the kinds of visual elements used.

Step grain. The size of the smallest step with which the user can step through the program. *Statement* (S) means that an entire statement, definition, or declaration (usually a single line) is executed at once, although stepping into and out of functions/methods may be a separate step. *Expression* (E) means that the user steps through stages of expression evaluation in more detail. *Message passing* (MP) refers to stepping through the program one object interaction at a time.

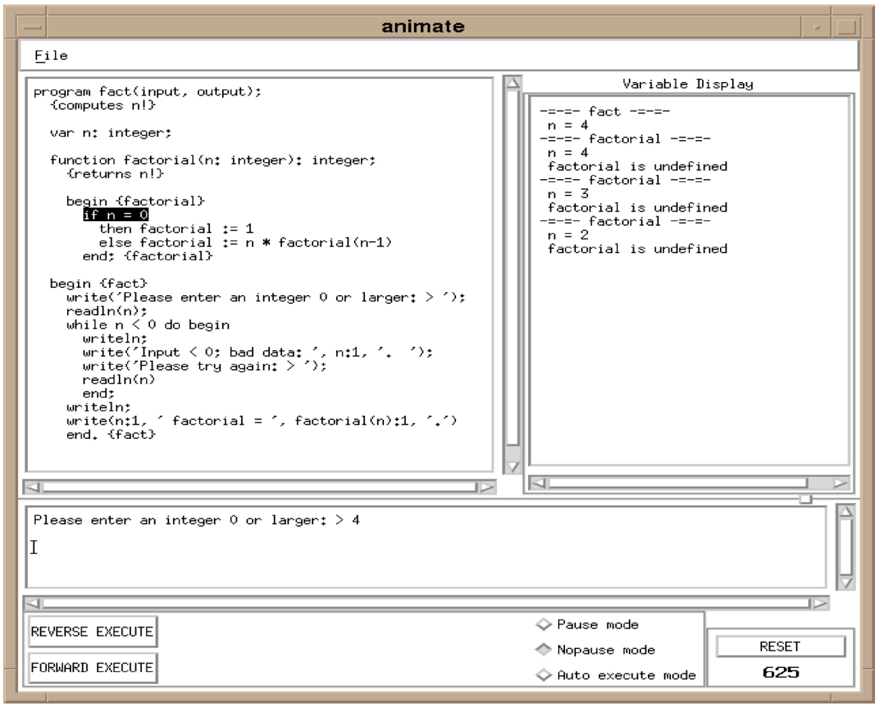
Direct engagement. The levels of direct engagement between learner and visualization which the system explicitly supports, in terms of the 2DET (Section 4.2). The presenting and creating levels of the 2DET do not feature in the table, as none of the systems explicitly support them, which of course does not imply that the visualizations shown by the systems cannot be presented to others. The basic engagement level of merely viewing a visualization is not listed separately unless it is the only one present (which is not the case in any of the tools reviewed). We consider each system as a generic PV system: modes of interaction that are particular to a specific subsystem or programming concept (e.g., an object inspection tool) are not listed.

Content ownership. The degree of learners' ownership of the software whose behavior is visualized by the system. Again, the levels are taken from the 2DET taxonomy. In this column, we have generally only listed the highest degree of ownership that the system allows, as systems that support learner-defined content also support teacher-defined content. However, given content is separately listed along with own content in cases where the system has a distinct mode of use specifically meant for ready-made example programs.

5.2. Controlled Viewing of Given Examples

The following systems are primarily meant for controlled viewing of built-in or teacher-defined examples (i.e., given content). Some of them also feature some support for own cases or modified content.

5.2.1. Early Libraries of Examples: LOPLE/DYNAMOD/DynaLab, EROSI, and Fernández et al.'s Tool. Although many software visualization systems were developed in the 1980s, few that we are aware of fall within the scope of this review, as education-oriented systems tended to deal with algorithm visualization [e.g., BALSA; see Brown 1988] whereas the program visualization systems were intended for expert use [e.g., VIPS; Isoda et al.



```
program fact(input, output);
  {computes n!}

  var n: integer;

  function factorial(n: integer): integer;
    {returns n!}

    begin {factorial}
      if n = 0
        then factorial := 1
        else factorial := n * factorial(n-1)
      end; {factorial}

  begin {fact}
    write('Please enter an integer 0 or larger: > ');
    readln(n);
    while n < 0 do begin
      writeln;
      write('Input < 0; bad data: ', n:1, '.  ');
      write('Please try again: > ');
      readln(n)
      end;
    writeln;
    write(n:1, ' factorial = ', factorial(n):1, '.')
    end. {fact}
```

```
-=-=- fact -=-=-
 n = 4
-=-=- factorial -=-=-
 n = 4
 factorial is undefined
-=-=- factorial -=-=-
 n = 3
 factorial is undefined
-=-=- factorial -=-=-
 n = 2
 factorial is undefined
```

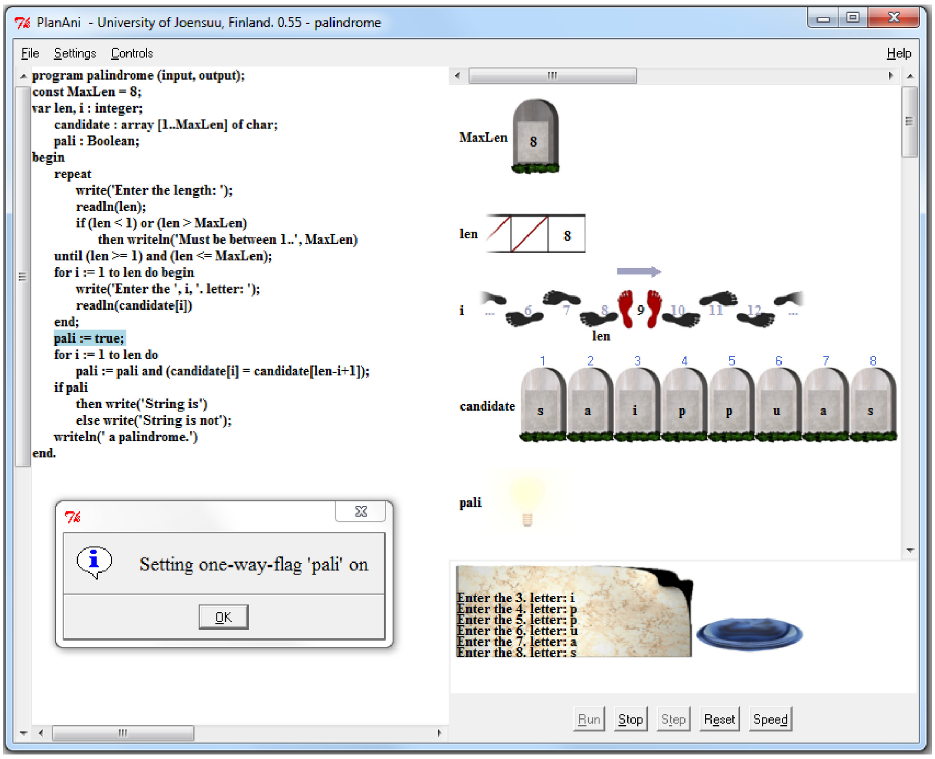
```
Please enter an integer 0 or larger: > 4
```

Fig. 7. DynaLab executing an example program [Birch et al. 1995].

1987]. An exception to this trend was *LOPLE*, a dynamic Library of Programming Language Examples [Ross 1983]. LOPLE was designed to allow novices to step through the execution of given example programs. The manner of execution in LOPLE was similar to that of a modern visual debugger. LOPLE evolved first into *DYNAMOD* [Ross 1991] and then into DynaLab [Birch et al. 1995; Boroni et al. 1996]. DynaLab (Figure 7) allowed execution to be stepped backwards, a feature that novices prized, according to student feedback. The earliest work was done with Pascal; the platform later supported various other programming languages as well. The authors provide anecdotal evidence of the usefulness of the tool in their teaching [Boroni et al. 1996; Ross 1991]. Their students liked the system, too [Ross 1991].

George [2000a, 2000b, 2000c, 2002] evaluated a system called *EROSI*, which was built primarily for illustrating procedure calls and recursion. EROSI featured a selection of program examples, whose execution it displayed. Subprogram calls were shown in separate windows with arrows illustrating the flow of control between them. George demonstrated through analyses of student assignments and interviews that the tool was capable of fostering a viable “copies” model of recursion which students could then apply to program construction tasks. He reports that students liked EROSI.

Fernández et al. [1998] created a tool for illustrating the inner behavior of object-oriented Smalltalk programs to students on a high level of abstraction. This unnamed system was built on top of a more generic learning platform called LearningWorks, which used a visual book metaphor to present information. The system visualized the dynamic object relationships within teacher-defined example systems of classes by drawing object diagrams in which interobject references and messages passed were shown as arrows of different kinds.


PlanAni - University of Joensuu, Finland. 0.55 - palindrome
File Settings Controls Help
MaxLen
8
len
8
i
... 6 7 8 9 10 11 12 ...
len
candidate
1 2 3 4 5 6 7 8
s a i p p u a s
pali
Run Stop Step Reset Speed

```
program palindrome (input, output);
const MaxLen = 8;
var len, i : integer;
    candidate : array [1..MaxLen] of char;
    pali : Boolean;
begin
    repeat
        write('Enter the length: ');
        readln(len);
        if (len < 1) or (len > MaxLen)
            then writeln('Must be between 1..', MaxLen)
    until (len >= 1) and (len <= MaxLen);
    for i := 1 to len do begin
        write('Enter the ', i, '. letter: ');
        readln(candidate[i])
    end;
    pali := true;
    for i := 1 to len do
        pali := pali and (candidate[i] = candidate[len-i+1]);
    if pali
        then write('String is')
        else write('String is not');
    writeln(' a palindrome.')
end.
```

```
Setting one-way-flag 'pali' on
OK
```

```
Enter the 3. letter: i
Enter the 4. letter: p
Enter the 5. letter: p
Enter the 6. letter: u
Enter the 7. letter: a
Enter the 8. letter: s
```

Fig. 8. PlanAni executing a Pascal program. Variables are visualized in role-specific ways. A "fixed value" is represented by carving a value in stone. A "most-recent holder" is represented by a box with previous values crossed out. A "stepper" is represented by footprints leading along a sequence of values. The lamp representing a "one-way flag" is being switched on, as explained by the popup dialog.

5.2.2. Roles and Metaphors: PlanAni and the Metaphor-Based OO Visualizer. *Roles of variables* [Sajaniemi n.d.] are stereotypical variable usage patterns. For instance, a variable with the role "stepper" is assigned values according to some predefined sequence (for example, 0, 1, 2, etc.) while a "fixed value" is a variable whose value is never changed once it is initially assigned to. Sajaniemi and his colleagues built a system called PlanAni for the visualization of short, imperative CS1 program examples in terms of the variables involved and in particular their roles [Sajaniemi and Kuittinen 2003]. Each variable is displayed using a role-specific visual metaphor (see Figure 8). For instance, "a fixed value is depicted by a stone giving the impression of a value that is not easy to change, and . . . A stepper is depicted by footprints and shows the current value and some of the values the variable has had or may have in the future, together with an arrow giving the current direction of stepping". PlanAni visualizes operations on these variables (e.g., assignment) as animations which, again, are role-specific.

PlanAni cannot visualize arbitrary programs, only examples that a teacher has configured in advance (in Pascal, C, Python, or Java). The teacher can include explanations to be shown at specific points during the execution sequence. Through roles, the system aims to develop students' repertoire of variable-related solution patterns for typical problems.

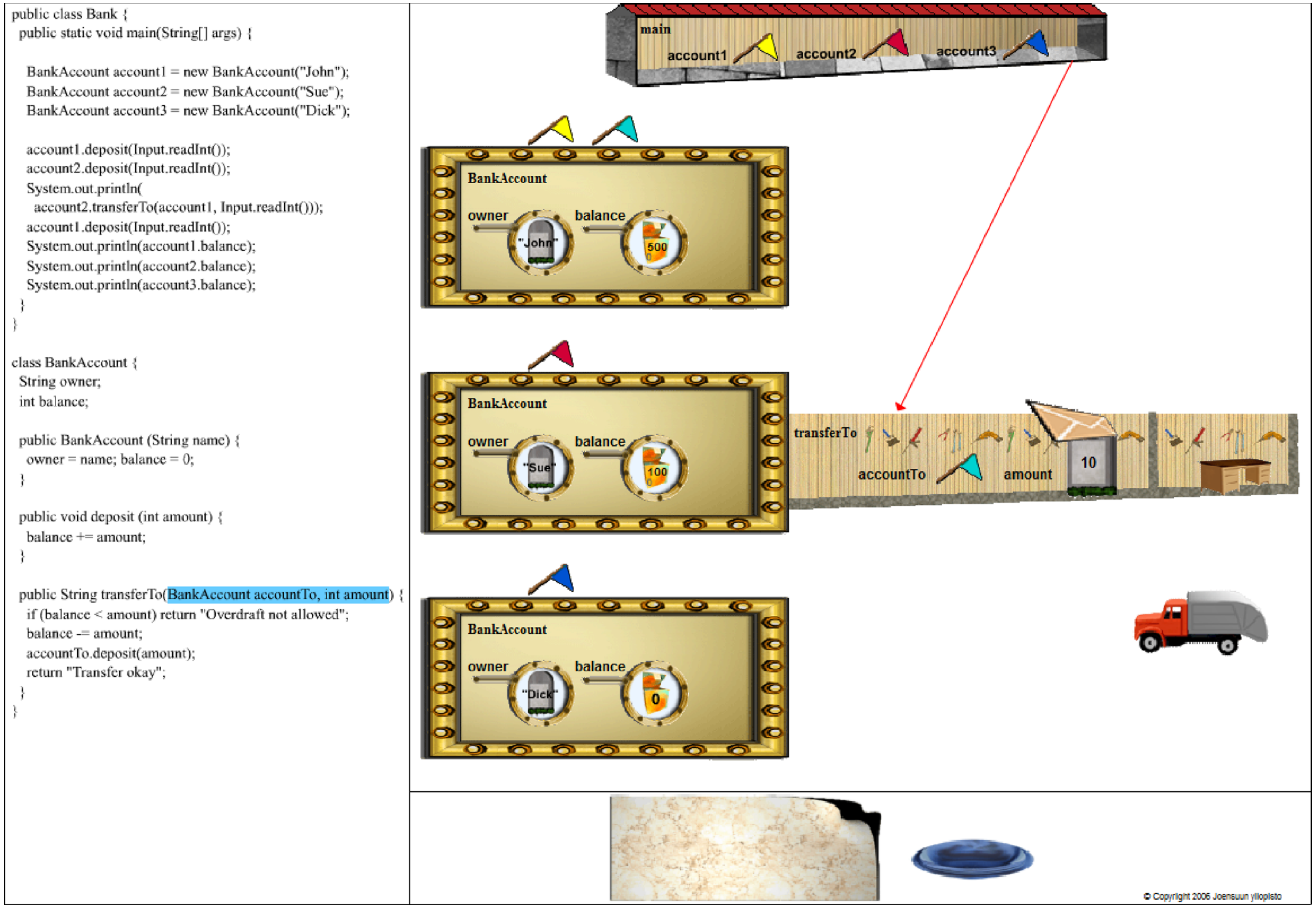
Sajaniemi and Kuittinen [2003] report that PlanAni had a positive impact on in-class discussions compared to a control group that used a regular visual debugger. The students liked the system and worked for longer with it, whereas the debugger group tended to surf the web more during labs. Sajaniemi and Kuittinen further argue, on the basis of an experiment [Sajaniemi and Kuittinen 2005], that using PlanAni helped foster the adoption of role knowledge and PlanAni users understood programs more deeply than non-users did, although the deeper understanding of the PlanAni users was not significantly reflected in the correctness of their answers. Expanding upon these results, Byckling and Sajaniemi [2005] report that students using PlanAni outperformed other students in code-writing tasks and exhibited significantly more forward-developing behavior while coding, which is suggestive of increased programming knowledge.

Nevalainen and Sajaniemi [2005] used eye-tracking technology to compare the targeting of visual attention of PlanAni users on the one hand and visual debugger users on the other. As might be expected, PlanAni users focused a great deal more on variables. Nevalainen and Sajaniemi also studied program summaries written by the two groups immediately after using the tool, and conclude that PlanAni increased the use of higher-level information at the expense of low-level, code-related information. Nevalainen and Sajaniemi further report that students found PlanAni to be clearer but relatively unpleasant to use (because too slow) compared to a visual debugger. In another publication, the same authors similarly compared how novice programmers used a regular PlanAni and a variant with no animations [Nevalainen and Sajaniemi 2006]. They found that irrespective of the version of the tool, the users mainly relied on textual cues (popup windows and program code). Nevalainen and Sajaniemi conclude that the location and size of visualizations is more important than animation, and that using the role images is more significant than animating them. In yet another experiment, however, Nevalainen and Sajaniemi [2008] did not find the presence of role images to be crucial to the formation of role knowledge compared to a version of PlanAni in which only textual explanations of roles were present. Stützle and Sajaniemi [2005] found that the role metaphors used in PlanAni worked better than a neutral set of control metaphors.

A "sequel" to PlanAni is the metaphor-based animator for object-oriented programs envisioned by Sajaniemi and his colleagues, who recommend that their system be used by students who have first grasped some fundamental programming concepts using PlanAni and are now learning about object-oriented concepts [Sajaniemi et al. 2007]. Their OO animator (Figure 9) also uses visual metaphors for variable roles but further adds support for classes (blueprints), objects (filled-in pages), references (flags), method calls and activations (envelopes and workshops), and garbage collection (by a garbage truck). The system is otherwise fairly similar to PlanAni. It does not work with arbitrary programs but only with predefined examples; the existing incarnation of the system is a web page on which a number of examples can be accessed as Flash animations.

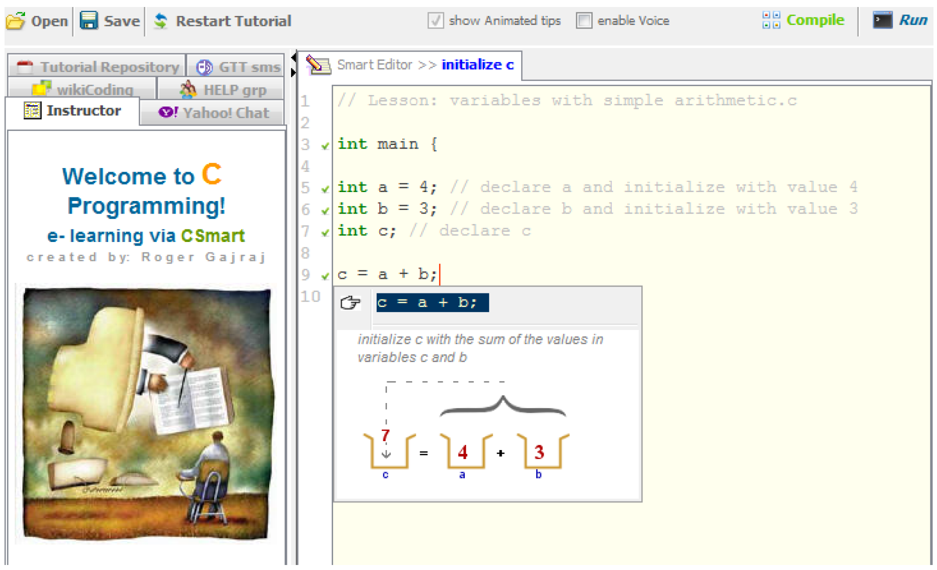
5.2.3. Examples in C and Related Languages: Miyadera et al.'s System and CSmart. A system for animating C programs was presented by Miyadera et al. [2007]. Their innominate system resembled a regular visual debugger, but allowed the user to step backwards in execution and was capable of graphically visualizing and animating the creation of variables, for instance. At least the primary goal of the system appears to have been the generation of animations of teacher-provided programs. The authors' main research focus was learners' use of animation controls and the assessment of the difficulty of different lines of code; the system collected such data and presented it for the teacher to study.

The *CSmart* [Gajraj et al. 2011] takes an approach to visualization that is quite different from all the other tools that we reviewed, and features an unusual form of controlled viewing. Instead of visualizing the execution of an existing program, CSmart



```
public class Bank {
  public static void main(String[] args) {

    BankAccount account1 = new BankAccount("John");
    BankAccount account2 = new BankAccount("Sue");
    BankAccount account3 = new BankAccount("Dick");

    account1.deposit(Input.readInt());
    account2.deposit(Input.readInt());
    System.out.println(
      account2.transferTo(account1, Input.readInt()));
    account1.deposit(Input.readInt());
    System.out.println(account1.balance);
    System.out.println(account2.balance);
    System.out.println(account3.balance);
  }
}

class BankAccount {
  String owner;
  int balance;

  public BankAccount (String name) {
    owner = name; balance = 0;
  }

  public void deposit (int amount) {
    balance += amount;
  }

  public String transferTo(BankAccount accountTo, int amount) {
    if (balance < amount) return "Overdraft not allowed";
    balance -= amount;
    accountTo.deposit(amount);
    return "Transfer okay";
  }
}
```

Fig. 9. The metaphor-based OO animator of Sajaniemi et al. running a Java program. Objects are shown as filled-in pages of a blueprint book. Method invocations (here, of `main` and `transferTo`) are shown as workshops which contain local variables. References are represented by flags whose color matches a flag on the target object. The garbage truck occasionally gathers unreferenced objects.



Welcome to C Programming!
e- learning via CSmart
created by: Roger Gajraj

```
1  // Lesson: variables with simple arithmetic.c
2
3  int main {
4
5  int a = 4; // declare a and initialize with value 4
6  int b = 3; // declare b and initialize with value 3
7  int c; // declare c
8
9  c = a + b;
10
```

c = a + b;

initialize c with the sum of the values in variables c and b

Fig. 10. The user has just finished typing in a line of code within the CSmart IDE. The system's instructions to the user are still visible below line 10. In this case, they consist of the code that the student must duplicate, a short teacher-authored comment, and a graphical metaphor of the statement to be written.

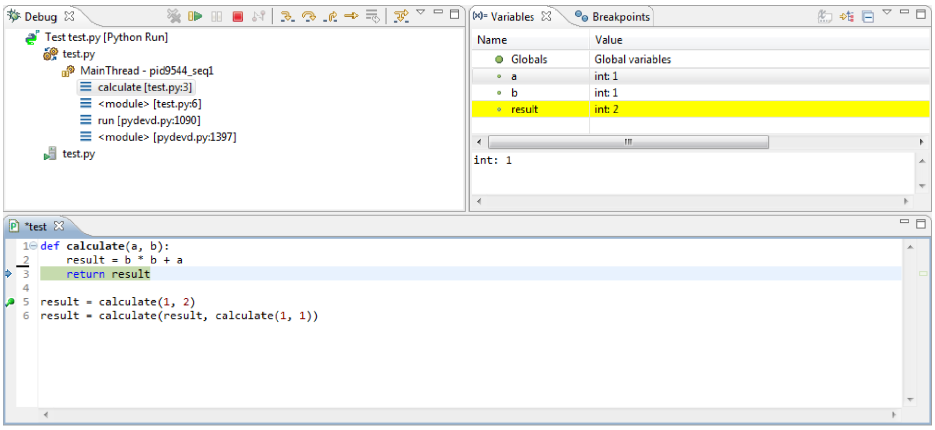
already visualizes each statement of a C program to the student *before* the student types it in.

CSmart provides example-based tutorials in which the system assists the student in duplicating teacher-defined example programs and thereby practicing programming fundamentals. The system knows exactly what program it requires the student to write in each programming assignment. It instructs the student at each step using text, graphics, and audio. Some of the guidance is teacher-annotated into the model solutions, some generated automatically. Various visual metaphors have been built into the system to illustrate the runtime semantics of the programming language; an example is shown in Figure 10. Gajraj et al. report that their students liked CSmart.

5.3. Controlled Viewing of User Content

In terms of the 2DET taxonomy, the tools listed in this subsection represent the mainstream of program visualization for introductory programming education: controlled viewing of own content. (In addition, one of the systems, EVizor, supports responding.) Most of these systems can be described as educationally motivated variants of the visual debuggers that professional programmers use, or as IDEs that contain such debuggers. Since there are a lot of systems of this kind, we present them grouped into a few (non-taxonomical) themes, as follows.

5.3.1 "Regular" visual debuggers for imperative programming.
5.3.2 Early work: program animation systems for the formerly popular CS1 languages BASIC and Pascal, their immediate relatives, and pseudocode.
5.3.3 Program animation systems for C/C++.
5.3.4 Program animation systems for Java.
5.3.5 The various versions of the much-studied Eliot/Jeliot program animation system.



```
Test test.py [Python Run]
    test.py
        MainThread - pid9544_seq1
            calculate [test.py:3]
            <module> [test.py:6]
            run [pydevd.py:1090]
            <module> [pydevd.py:1397]
    test.py
```

Name	Value
Globals	Global variables
a	int: 1
b	int: 1
result	int: 2

```
int: 1
```

```
1 def calculate(a, b):
2     result = b * b + a
3     return result
4
5 result = calculate(1, 2)
6 result = calculate(result, calculate(1, 1))
```

Fig. 11. The PyDev debugger for Python programs within the Eclipse IDE. A Python program is being executed, with line 3 up next. Threads and call stacks are listed in the top left-hand corner. Global and local variables are shown on the right. The yellow highlight signifies a change in the value of a variable.

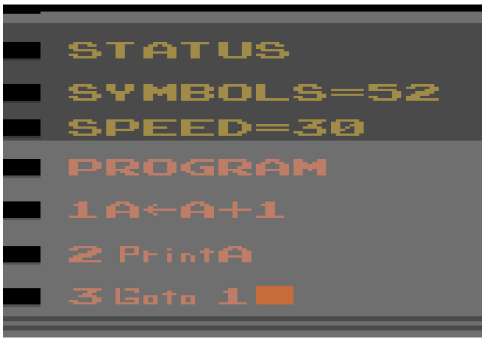
5.3.6 Other multi-language program animation systems (which support both C/C++ and Java, and sometimes other languages).
5.3.7 Program animation systems for Python.
5.3.8 Visualization tools for functional programming.
5.3.9 Tools that do not support stepping through programs in vivo but instead produce snapshots of programs that can be studied post mortem.

5.3.1. Regular Visual Debuggers.

> "Tools that reflect code-level aspects of program behavior, showing execution proceeding statement by statement and visualizing the stack frame and the contents of variables . . . are sometimes called *visual debuggers*, since they are directed more toward program development rather than understanding program behavior." [Pears et al. 2007, p. 209]

Programming experts—and novices, though not as often as many programming teachers would like—use debugging software such as that shown in Figure 11 to find defects in programs and to become familiar with the behavior of complex software. These tools are not particularly education-oriented or beginner-friendly, but can still be useful in teaching [see, e.g., Cross et al. 2002] and may be integrated into otherwise beginner-friendly environments such as the BlueJ IDE [Kölling 2008].

Typical visual debuggers have significant limitations from the point of view of learning programming fundamentals. They generally step through code only at the statement level, leaving most of the educationally interesting dynamics of expression evaluation implicit. The visualization and user interface controls of a "regular visual debugger" are geared toward programming-in-the-large, not toward explicating programming concepts and principles such as assignment, function calls, parameter passing, and references, all of which the user is assumed to understand already. Only information essential for an expert programmer is shown, with the goal of helping the programmer to find interesting (defective) stages of execution in as few steps as possible while ignoring as many details as possible. The target programs may be large in terms of both code and data. For such reasons, the visualization shown by a typical visual debugger is not particularly graphic, and consists primarily of text within standard GUI widgets.



```
STATUS
SYMBOLS=52
SPEED=30
PROGRAM
1 A←A+1
2 Print A
3 Goto 1
```

Fig. 12. A part of the Basic programming environment on the Atari 2600 [image from Atwood 2008].

Bennedsen and Schulte [2010] conducted an experimental study in which a group of CS1 students used the visual debugger built into the BlueJ IDE to step through object-oriented programs, while a control group used manual tracing strategies. They found no significant differences in the performance of the groups on a post-test of multiple-choice questions on program state. A rerun of the experiment using a different debugger yielded similar results. Bennedsen and Schulte surmise that "it could be that the debugger is not useful for understanding the object interaction but just for finding errors in the program execution."

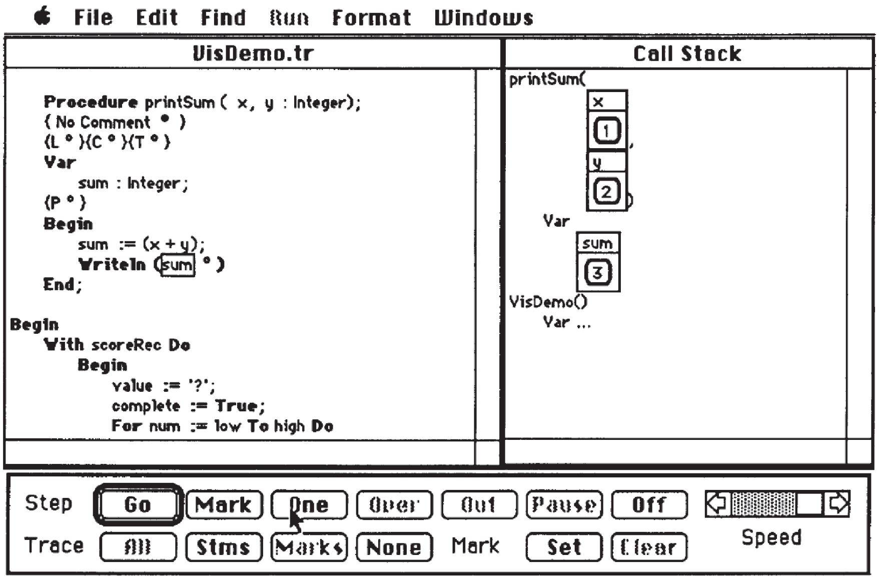
Despite their limitations, visual debuggers are worth a mention in this section because they are highly useful tools that novices do encounter in CS1 courses, because they do visualize certain aspects of program dynamics, and because they serve as a point of departure for reviewing the more education-oriented systems below.

5.3.2. Early Work: Systems for BASIC, Pascal (Plus Relatives), and Pseudocode: Basic Programming, Amethyst, DISCOVER, and VisMod. An early educational PV system that supported visual tracking of program execution was *Basic Programming*, "an instructional tool designed to teach you the fundamental steps of computer programming" [Robinett 1979]. Basic Programming was an integrated environment for the Atari 2600 computer, in which the user could input BASIC code and view its execution. In addition to the Status and Program regions shown in Figure 12, the system featured a Variables region that displayed the current values of variables during a program run and a Stack region that showed the stages of expression evaluation in more detail than a regular visual debugger does. The system also provided a 2D graphics region for displaying sprites.

Amethyst (Figure 13) was a PV system prototype that visualized data as two-dimensional graphics during a program run [Myers et al. 1988]. Amethyst differed from the earlier algorithm visualization systems from which it was derived in that it created visualizations automatically for any program. The user would nevertheless manually mark through the GUI which data items should be visualized.

DISCOVER [see Ramadhan et al. 2001, and references therein] was a prototype of an intelligent tutoring system that featured an explicit conceptual model of a notional machine in the form of a visualization (Figure 14). We discuss here only the software visualization features of DISCOVER.

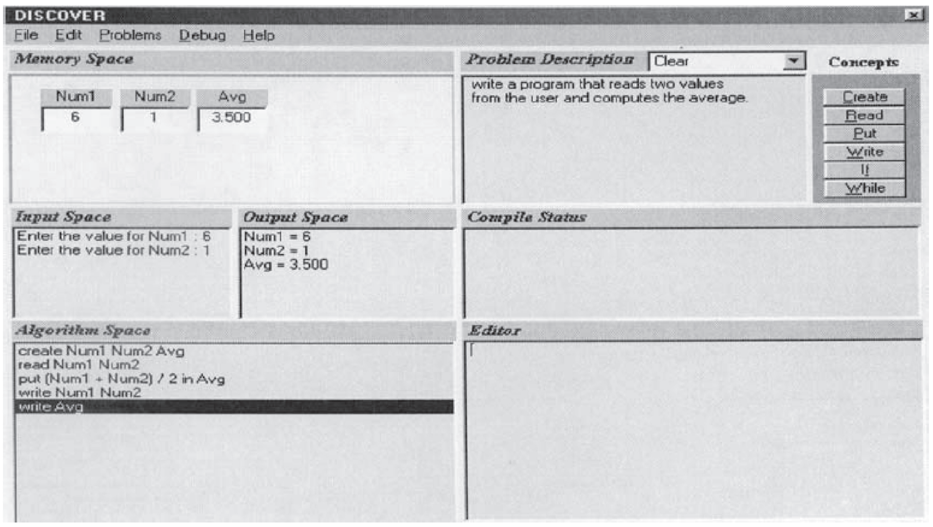
One way for students to use DISCOVER was similar to that typical of the other tools reviewed in this section: the student would step through an existing program's execution and observe changes in variables, etc., with the help of a graphical machine model. A more unusual feature of the system was the immediate visualization of partial solutions during editing. When this functionality was enabled and the user typed in a new



```
   Procedure printSum ( x, y : Integer);
   { No Comment  * }
   {L * }{C * }{T * }
   Var
       sum : Integer;
   {P * }
   Begin
       sum := (x + y);
       Writeln (sum * )
   End;

Begin
   With scoreRec Do
       Begin
           value := '?';
           complete := True;
           For num := low To high Do
```

```
printSum(
          x
          1 ,
          y
          2 )
     Var
          sum
          3
VisDemo()
     Var ...
```

Fig. 13. A Pascal program in Amethyst [Myers et al. 1988]. Amethyst used different shapes to emphasize the different types of values; for instance, the rounded rectangles in this figure denote integer values.



Num1	Num2	Avg
6	1	3.500

write a program that reads two values from the user and computes the average.

```
Enter the value for Num1 : 6
Enter the value for Num2 : 1
```

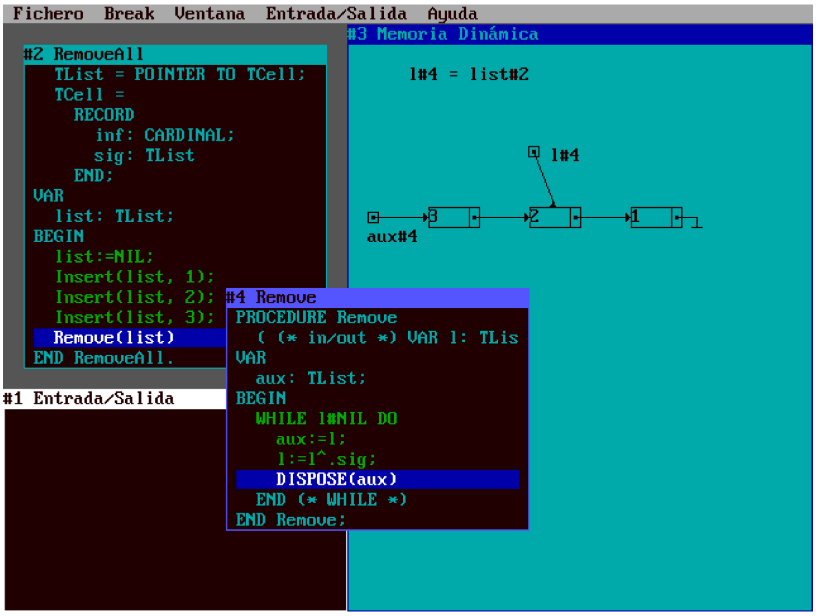
```
Num1 = 6
Num2 = 1
Avg = 3.500
```

```
create Num1 Num2 Avg
read Num1 Num2
put (Num1 + Num2) / 2 in Avg
write Num1 Num2
write Avg
```

Fig. 14. The user has just completed writing a program in DISCOVER [image from Ramadhan et al. 2001]. He has used the editor to input pseudocode into the Algorithm Space. In the immediate execution mode, the effects of each new statement on execution instantly shown through updates to the other panels.

statement into their program, DISCOVER would instantly show onscreen the effects of the new statement on the entire program run.

Some of the evaluations of DISCOVER examined the impact of the visualization. In an experiment, students who used a visualization-enabled version of the system made fewer errors and completed tasks quicker than other students who used a version that did not visualize execution or support line-by-line stepping through execution;



```
  TList = POINTER TO TCell;
  TCell =
    RECORD
      inf: CARDINAL;
      sig: TList
    END;
VAR
  list: TList;
BEGIN
  list:=NIL;
  Insert(list, 1);
  Insert(list, 2);
  Insert(list, 3);
  Remove(list)
END RemoveAll.
```

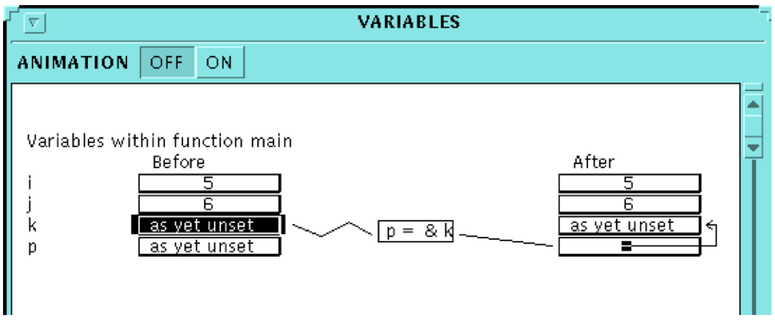
```
PROCEDURE Remove
  ( (* in/out *) VAR l: TLis
VAR
  aux: TList;
BEGIN
  WHILE l#NIL DO
    aux:=l;
    l:=l^.sig;
    DISPOSE(aux)
  END (* WHILE *)
END Remove;
```

Fig. 15. A Modula-2 program in VisMod [Jiménez-Peris et al. 1999].

however, the the result was not statistically significant [Ramadhan et al. 2001; see also Ramadhan 2000].

VisMod [Jiménez-Peris et al. 1999] was a beginner-friendly programming environment for the Modula-2 language. The system had various features designed with the novice programmer in mind, including a pedagogically oriented visual debugger (Figure 15). The system had a cascading-windows representation for the call stack, as well as line graphics of certain data structures (lists and trees). The authors report that students liked it.

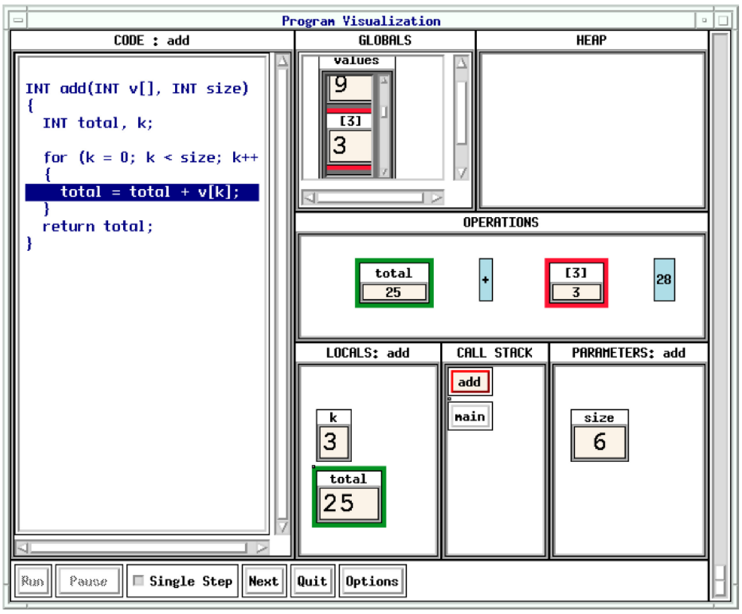
5.3.3. C/C++ Animators: Bradman, Korsh et al.'s Tool, OGRE, VINCE, and VIP. Smith and Webb [1991, 1995a] created *Bradman*, a visual debugger intended for novice C programmers. The explicit goal of the system was to improve students' mental models of program execution by helping them visualize the dynamic behavior of programs. As with any debugger, a user of Bradman could put in their own code and examine its behavior statement by statement. Compared to regular debuggers, a novelty in Bradman was the detailed English explanations of each statement as it was executed. Smith and Webb report that students liked these explanations and reacted particularly positively to a version of Bradman that included them in comparison to one that did not [Smith and Webb 1995b]. The other novelty in Bradman was its illustration of changes in program state. Bradman showed previous and current states of a program side by side for convenient comparison. This is pictured in Figure 16, which also illustrates how Bradman visualized references using graphical arrows. The explicit treatment of state changes was particularly useful since Bradman did not support stepping backwards. Smith and Webb [2000] report on an experimental evaluation of Bradman in which they found that CS1 students who used Bradman for examining teacher-given programs performed significantly better in a multiple-choice post-test on parameter passing than did students without access to Bradman. Other, similar tests performed during the intervention did not yield statistically significant differences between groups.



Variables within function main

	Before		After
i	5		5
j	6		6
k	as yet unset	p = & k	as yet unset
p	as yet unset		■

Fig. 16. One of Bradman's GUI windows after the C statement `p = & k` has just been executed [Smith and Webb 1995a]. The previous state is shown side by side with the current state. (The rest of the program does not appear in the Variables window.)

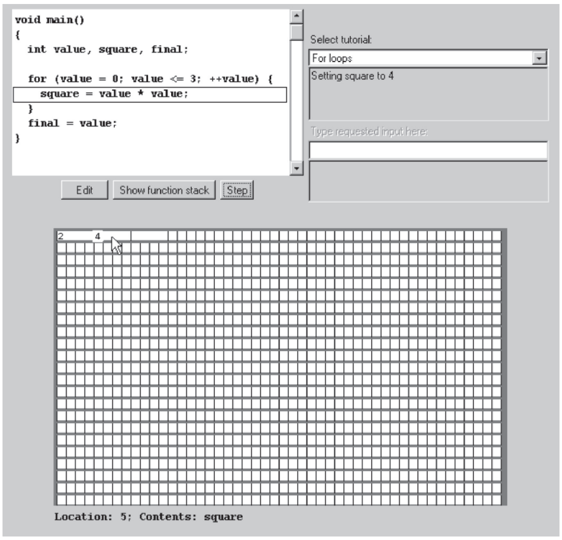


```
INT add(INT v[], INT size)
{
  INT total, k;

  for (k = 0; k < size; k++
  {
    total = total + v[k];
  }
  return total;
}
```

Fig. 17. The program visualization tool of Korsh et al. [image from Korsh and Sangwan 1998]. The user has used uppercase type declarations to mark which parts of the program should be visualized on the right. Expression evaluation is shown step by step in the Operations panel.

A prototype system for visualizing the behavior of C++ programs for CS1 and CS2 students was presented by LaFollette et al. [2000; Korsh and Sangwan 1998]. Their tool used abstract graphics (mostly boxes inside boxes; see Figure 17) to visualize the values of variables, the call stack, and the stages of expression evaluation. The system required the user to annotate the source code to indicate which parts he wished to visualize, which may have made it quite challenging for novices to use effectively. The user could also adjust the level of detail by selecting which operations were animated and which were not.

VINCE was a tool for exploring the statement-by-statement execution of C programs—self-written by students or chosen from a selection of given examples [Rowe and Thorburn 2000]. VINCE visualized computer memory on a relatively low level of abstraction, as a grid which illustrated where pointers point and references refer



```
void main()
{
  int value, square, final;

  for (value = 0; value <= 3; ++value) {
    square = value * value;
  }
  final = value;
}
```

Select tutorial: For loops

Setting square to 4

Location: 5; Contents: square

Fig. 18. VINCE executing a C program [Rowe and Thorburn 2000]. Each square in the grid corresponds to a single byte of memory. Note that the mouse cursor is over a four-byte integer, which is currently stored in the variable `square` as indicated by the text at the bottom.

(Figure 18). The system's authors compared the confidence and C knowledge of CS1 students who used VINCE for extra tutorials over a three-week period to those of a control group who did not do the extra tutorials. Their results suggest that VINCE had no significant impact on students' self-assessment of their programming ability, but the VINCE users did in fact learn more than the control group (as might be expected since they had extra learning activities). The students liked VINCE.

A related system that came out later, *OGRE*, visualized C++ programs using 3D graphics [Milne and Rowe 2004]. Each scope within a running program (e.g., a method activation) was represented by a flat plane on which small 3D figures appear to represent objects and variables. References and pointers were shown as arrows, and data flow as an animation in which a cylinder moved through a pipe between source and target (Figure 19). The user could step forward and backward, and use 3D-game-like controls for moving about, rotating the view, and zooming in and out. Milne and Rowe [2004] conducted an experimental study to determine the effectiveness of the OGRE approach. Their target group was not a CS1 course but second-year students who had just completed a course on object-oriented C++ programming. Milne and Rowe report that students who were given additional OGRE-based tutorials on certain difficult topics to complement other learning materials could answer questions on those same topics significantly better than other students who had not had that access. An interview study showed that students liked OGRE, as did the instructors who had used it.

VIP is another system for visualizing C++ programs that is akin to a visual debugger but intended for a CS1 audience [Virtanen et al. 2005]. It displays relationships between variables (e.g., references are shown as arrows) and highlights the details of expression evaluation. As shown in Figure 20, VIP has a facility for displaying teacher-given hints and instructions to the student at predefined points in ready-made example programs [Lahtinen and Ahoniemi 2007]. The system does not support object-orientation.

Isohanni (née Lahtinen) and her colleagues have reported on the use of VIP in a number of recent papers.

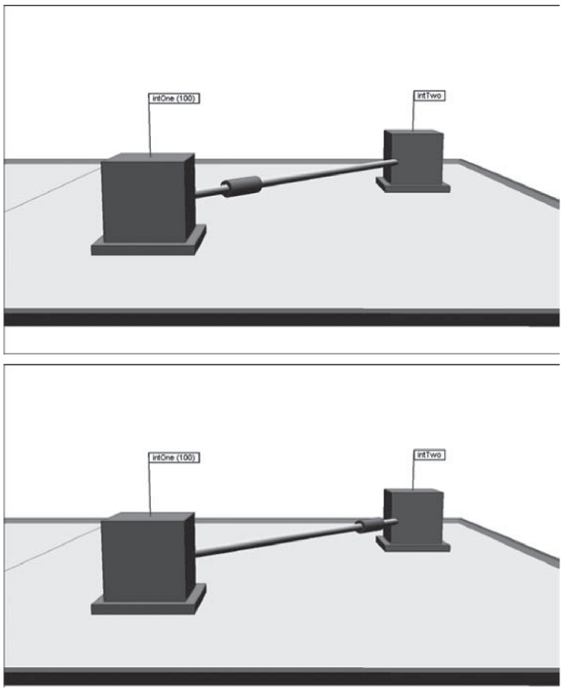
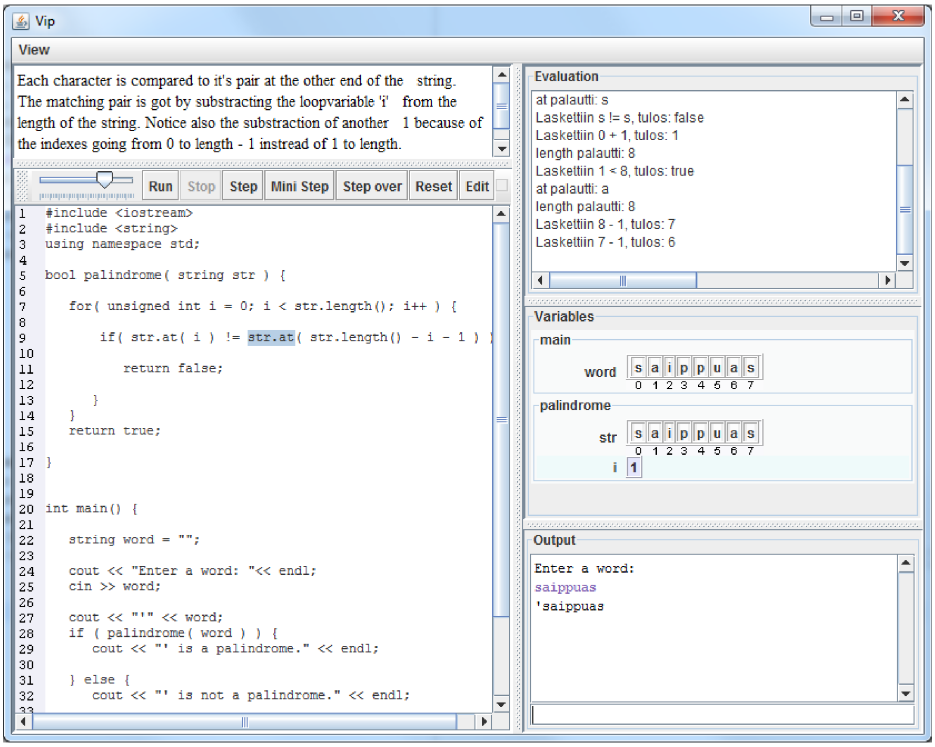


Fig. 19. Two snapshots of the animation that OGRE used to show a value being assigned from a variable to another [Milne and Rowe 2004].

Lahtinen et al. [2007a] describe a CS1 course in which students were given the opportunity to use VIP when doing voluntary “pre-exercises” in preparation for lab sessions. Of the students who did the pre-exercises, more than a quarter chose to use VIP, but traditional pen-and-paper strategies were more popular still. Lahtinen et al. also observed that the volunteers who used VIP were less likely to drop out of the course; this does not imply that VIP was responsible for this trend, however.

Another article reports a survey of programming students in a number of European universities on their opinions of program visualization [Lahtinen et al. 2007b]. This study was not specific to VIP, but roughly half of the students surveyed had taken a course in which VIP was used. The survey results suggest that the students who found programming challenging but manageable were the most positive about using visualizations, while the strongest and weakest students were less impressed. These findings are in line with the study of Ben-Bassat Levy et al. [2003] on Jeliot 2000, discussed above, in which a “middle effect” was observed.

Ahoniemi and Lahtinen [2007] conducted an experiment in which randomly selected students used VIP during CS1, while a control group did not. They tested the students on small code-writing tasks. No significant differences were found when the entire treatment group and control group were considered. However, Ahoniemi and Lahtinen also identified among their students “novices and strugglers” who either had no prior programming experience or for whom the course was difficult. Of the novices and strugglers, the ones who used VIP did significantly better than the ones in the control group. Ahoniemi and Lahtinen also surveyed the students to find out how much time they had used to review materials and found that the novices and strugglers in the VIP group used more time than the ones in the control group did; this is not very surprising, since the students who used VIP had some extra materials (the visualizations



Each character is compared to it's pair at the other end of the string. The matching pair is got by substracting the loopvariable 'i' from the length of the string. Notice also the substraction of another 1 because of the indexes going from 0 to length - 1 instread of 1 to length.

```
#include <iostream>
#include <string>
using namespace std;

bool palindrome( string str ) {

   for( unsigned int i = 0; i < str.length(); i++ ) {

       if( str.at( i ) != str.at( str.length() - i - 1 ) 

          return false;

       }
   }
   return true;

}


int main() {

   string word = "";

   cout << "Enter a word: "<< endl;
   cin >> word;

   cout << "'" << word;
   if ( palindrome( word ) ) {
      cout << "' is a palindrome." << endl;

   } else {
      cout << "' is not a palindrome." << endl;
```

```
at palautti: s
Laskettiin s != s, tulos: false
Laskettiin 0 + 1, tulos: 1
length palautti: 8
Laskettiin 1 < 8, tulos: true
at palautti: a
length palautti: 8
Laskettiin 8 - 1, tulos: 7
Laskettiin 7 - 1, tulos: 6
```

	0	1	2	3	4	5	6	7
word	s	a	i	p	p	u	a	s
str	s	a	i	p	p	u	a	s

i 1

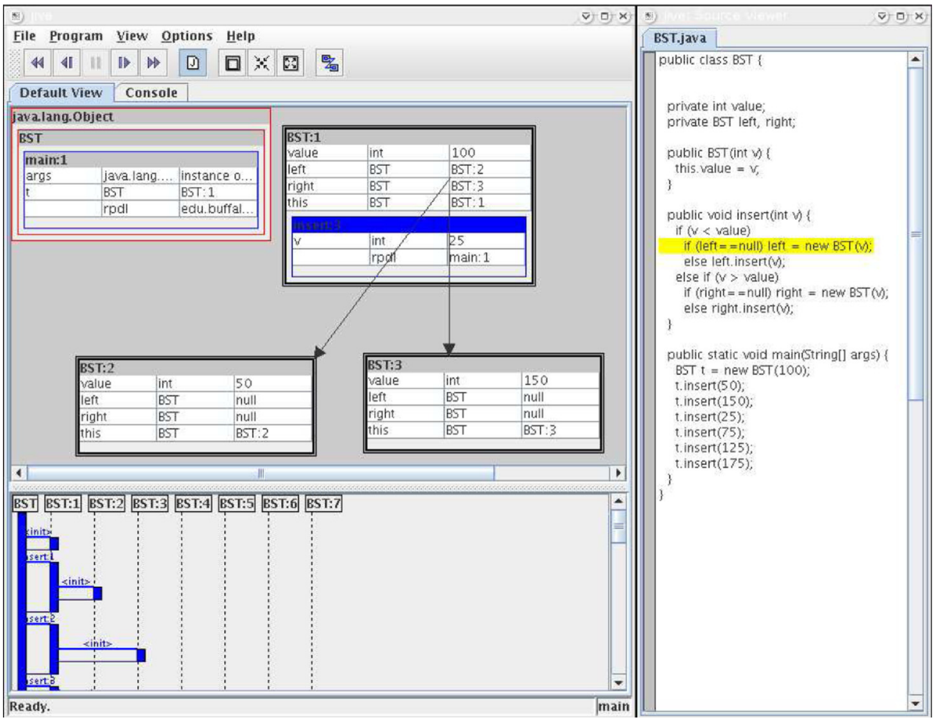
```
Enter a word:
saippuas
'saippuas
```

Fig. 20. A C++ program within VIP. The at method of a string is about to be called. The evaluation pane in the top right-hand corner displays previous steps; in this case, the most recent steps involve the evaluation of at's parameter expression. This is a teacher-given example program, into which the teacher has embedded natural language information about individual lines of code. One such explanatory message is shown in the top left-hand corner.

and their usage instructions) to study. The authors conclude that the benefit of visualizations may not be directly due to the visualization itself but to how the visualization makes studying more interesting and leads to increased time on task.

Isohanni and Knobelsdorf [2010] qualitatively explored how CS1 students used VIP on their own. They report that students use VIP for three purposes: exploring code, testing, and debugging. Isohanni and Knobelsdorf discuss examples of ways in which students use VIP for debugging in particular. Some students used VIP in the way intended by the teacher, that is, to step through program execution in order to find the cause of a bug. Other working patterns were also found, however. Some students stopped running the program in VIP as soon as a bug was found and from then on relied on static information visible in VIP's GUI (an extremely inefficient way of making use of VIP, the authors argue). Others still abandoned VIP entirely after they discovered a bug. Isohanni and Knobelsdorf's study shows that students do not necessarily use visualization tools in the way teachers intend them to, and underlines the need for explicit teaching about how to make use of a PV tool. In another article, the authors further describe the ways in which students use VIP; this work provides an illustration of how increasing familiarity with a visualization tool over time can lead to increasingly productive and creative ways of using it [Isohanni and Knobelsdorf 2011].

5.3.4. Java Animators: JAVAVIS, Seppälä's Tool, OOP-Anim, JavaMod, JIVE, Memview, CoffeeDregs, JavaTool, and EVizor. Three distinct system prototypes from around the same time



java.lang.Object

BST

main:1

args	java.lang....	instance o...
t	BST	BST:1
	rpdl	edu.buffal...

BST:1

value	int	100
left	BST	BST:2
right	BST	BST:3
this	BST	BST:1

insert:3

v	int	25
	rpdl	main:1

BST:2

value	int	50
left	BST	null
right	BST	null
this	BST	BST:2

BST:3

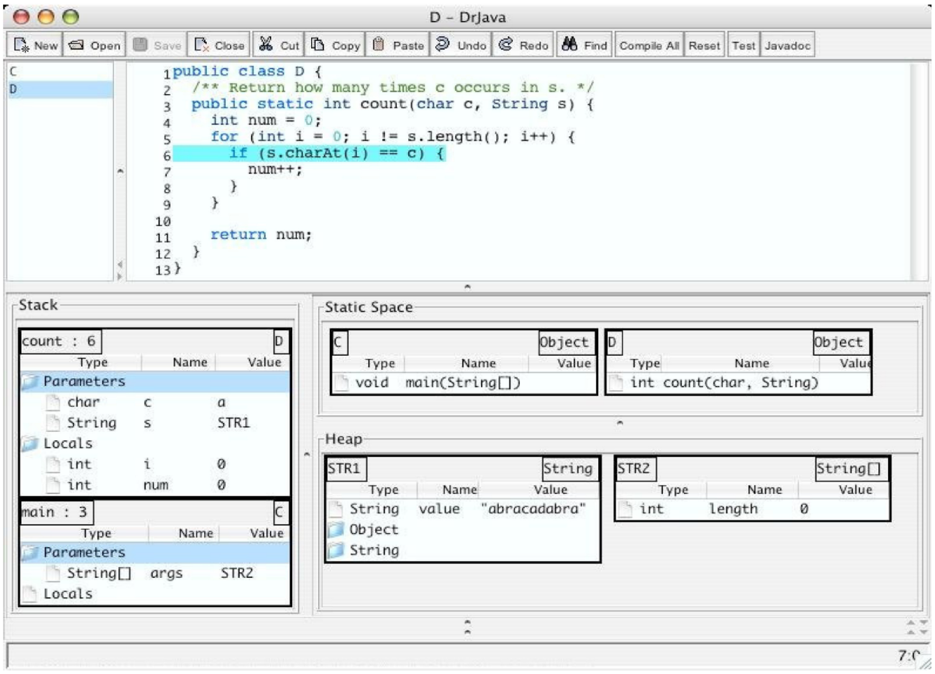
value	int	150
left	BST	null
right	BST	null
this	BST	BST:3

```
public class BST {


  private int value;
  private BST left, right;

  public BST(int v) {
    this.value = v;
  }

  public void insert(int v) {
    if (v < value)
      if (left==null) left = new BST(v);
      else left.insert(v);
    else if (v > value)
      if (right==null) right = new BST(v);
      else right.insert(v);
  }

  public static void main(String[] args) {
    BST t = new BST(100);
    t.insert(50);
    t.insert(150);
    t.insert(25);
    t.insert(75);
    t.insert(125);
    t.insert(175);
  }
}
```

Fig. 21. JIVE displays various aspects of a Java program's execution [Gestwicki and Jayaraman 2005]. Object relationships are illustrated on the left, with a sequence diagram of history information below it.

built on UML to illustrate the execution of Java programs. *JAVAVIS* [Oechsle and Schmitt 2002] was an educationally motivated debugger, which used UML object diagrams and sequence diagrams to expose program behavior in a learner-friendly way. Unlike many of the other systems reviewed, JAVAVIS featured limited support for multithreading. Seppälä [2004] presented a similar visual debugger for CS1 use. His prototype system visualized the execution of object-oriented Java programs as dynamic object state diagrams, a notation that "attempts to show most of the runtime state of the program in a single diagram." In particular, the system visualized both method invocations and references between objects in the same graph. Seppälä's visualization essentially combined elements of the object diagrams and collaboration diagrams of UML. *OOP-Anim* Esteves and Mendes [2003, 2004] was a program visualization tool for object-oriented programs, which produced a step-by-step visualization of its execution, showing classes in UML, and objects as lists of instance variables and methods. Variables were visualized as small boxes which could store references to objects (shown as connecting lines between variable and object).

Gallego-Carrillo et al. [2004] presented *JavaMod*, a visual debugger for Java programs with applications in education. The primary difference between JavaMod and a regular visual debugger lies in how JavaMod treated each structural element of the code separately rather than stepping through the program line by line. For instance, the initialization, termination check, and incrementation of a `for` loop were each highlighted as separate steps in execution (cf., e.g., the Basic Programming system, described previously, and Jeliot, The Teaching Machine, and UUhistle, discussion forthcoming).



```
public class D {
  /** Return how many times c occurs in s. */
  public static int count(char c, String s) {
    int num = 0;
    for (int i = 0; i != s.length(); i++) {
      if (s.charAt(i) == c) {
        num++;
      }
    }

    return num;
  }
}
```

count : 6 — D

Type	Name	Value
Parameters		
char	c	a
String	s	STR1
Locals		
int	i	0
int	num	0

main : 3 — C

Type	Name	Value
Parameters		
String[]	args	STR2
Locals		

C — Object

Type	Name	Value
void	main(String[])	

D — Object

Type	Name	Value
int	count(char, String)	

STR1 — String

Type	Name	Value
String	value	"abracadabra"
Object		
String		

STR2 — String[]

Type	Name	Value
int	length	0

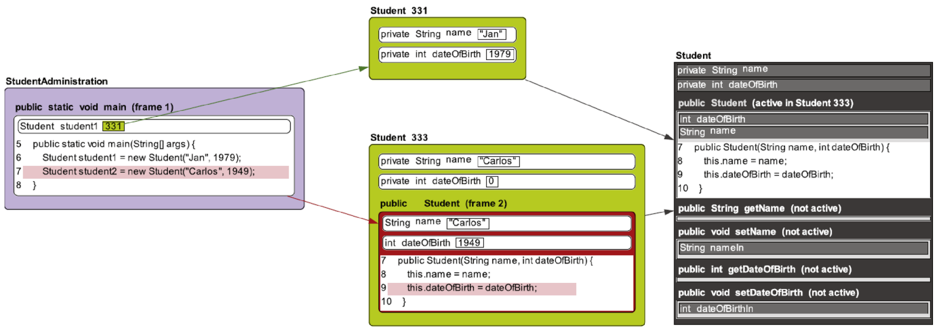
Fig. 22. A Java program running in Memview [Gries et al. 2005]. The classes C and D appear within Static Space, objects within the Heap, and frames on the Stack.

JIVE is a sophisticated debugging environment for Java programs [Gestwicki and Jayaraman 2005; Lessa et al., n.d.] currently implemented as a plugin for the Eclipse IDE. JIVE is not meant exclusively for pedagogical use and its expandable visualizations (Figure 21) can be used also for examining larger object-oriented programs, including multithreaded ones. JIVE supports reverse stepping and has various other features beyond the scope of our review. The authors have used the system in a variety of courses, introductory-level programming among them.

The creation of the *Memview* debugger [Gries et al. 2005] was motivated by the desire to support the use of Gries and Gries's teachable memory model (see Section 2.2 above) by introducing a system that automates the creation of memory diagrams. Memview, which is an add-on for the DrJava IDE [Allen et al. 2002], works much like a regular visual debugger, but has a more sophisticated and beginner-friendly way of displaying the contents of memory that works well for small CS1 programs (Figure 22). Gries et al. [2005] report anecdotal evidence of the tool's success in teaching.

CoffeeDregs [Huizing et al. 2012] is a visualization tool explicitly meant for presenting a conceptual model of Java program execution for learners. It emphasizes the relationships between classes and objects, which are represented as abstract diagrams. Only some tentative evaluations of the system prototype have been carried out so far [Luijten 2009].

One more tool for visualizing Java programs to novices is *JavaTool* [Brito et al. 2011; Mota et al. 2009]. The system is similar to a regular visual debugger, but features beginner-friendly controls for stepping through an animation of the program. Only a small subset of Java is supported (no objects; only a handful of standard functions can be called). JavaTool's most distinguishing feature is that it is designed as a plugin for the popular Moodle courseware. It is intended to be used for small programming


StudentAdministration
public static void main (frame 1)
Student student1 331
Student 331
private String name "Jan"
private int dateOfBirth 1979
Student 333
private String name "Carlos"
private int dateOfBirth 0
public Student (frame 2)
String name "Carlos"
int dateOfBirth 1949
Student
private String name
private int dateOfBirth
public Student (active in Student 333)
int dateOfBirth
String name
public String getName (not active)
public void setName (not active)
String nameIn
public int getDateOfBirth (not active)
public void setDateOfBirth (not active)
int dateOfBirthIn

```
public static void main(String[] args) {
    Student student1 = new Student("Jan", 1979);
    Student student2 = new Student("Carlos", 1949);
}
```

```
public Student(String name, int dateOfBirth) {
    this.name = name;
    this.dateOfBirth = dateOfBirth;
}
```

Fig. 23. A static method, two objects, and a class in EVizor [Moons and De Backer 2013]. The constructor of the student class has been invoked from the main method and is being executed.

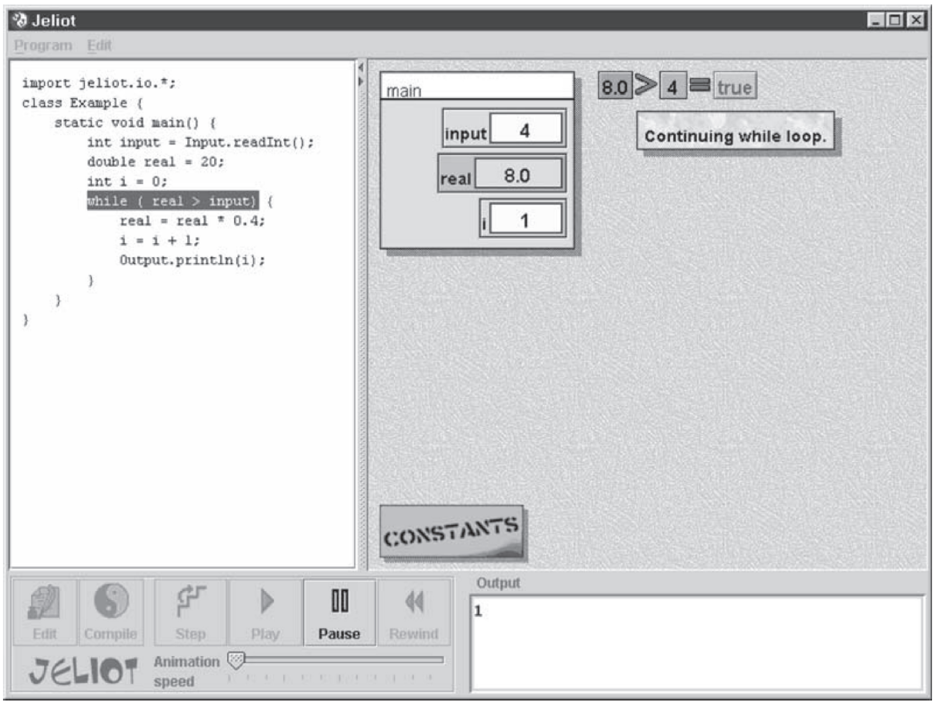
assignments in which the student writes and debugs code in their web browser and submits it for grading, receiving instant feedback from the system and optionally peer feedback from other students.

EVizor is a visualization plugin for the Netbeans IDE [Moons and De Backer 2013]. It visualizes Java program executions as abstract diagrams (Figure 23), allowing the user to step forward and backwards, to zoom and pan the visualization, and to adjust the positions of the visual elements. The user can also access online explanations of the elements on demand. In addition to controlled viewing, EVizor supports embedding interactive popup quizzes into a visualization—a form of responding. Unlike the large majority of the other tools we have reviewed, the visualization in EVizor is explicitly grounded in cognitive psychology, in particular cognitive load theory (e.g., textual explanations appear close to the relevant visual elements to avoid split attention) and research on visual perception (e.g., the colors used have been chosen so that they are easy for humans to recognize visually). Experiments by the system's authors indicate that students found it easier to answer questions about program behavior when they had access to EVizor than when they did not [Moons and De Backer 2013].

5.3.5. Long-Term Tool Development: From Eliot to Jeliot 3. One of the longest-lasting and most-studied program visualization tools for CS1 is *Jeliot*. Its longevity, as with almost any successful piece of software, is based on shedding its skin a few times, sometimes accompanied by a change of viscera. The stages of the Jeliot project have recently been reviewed by Ben-Ari et al. [2011].

Jeliot started out as *Eliot* [Sutinen et al. 1997], a software visualization tool that graphically animated data (variables) in user-defined C programs. In Eliot, the user selected which variables were animated, and also had a partial say in what the visualization of a program run looked like through the choice of colors and locations for the boxes that represented variables. Eliot's goals were primarily on the algorithm visualization side, and it reportedly worked best when used by programmers who had at least a bit of experience rather than by complete beginners.

Jeliot I, a Java implementation of Eliot [Haajanen et al. 1997], was a proof-of-concept system for bringing program animations to the World Wide Web. Lattu et al. [2000] evaluated Jeliot I in two introductory programming courses. Some of their interviewees found Jeliot I to be helpful and suitable for beginners. Although a statistical comparison was not possible, Lattu et al. observed that students in a course whose teaching was reworked to include Jeliot I as much as possible in lectures and assignments gained much more from the experience than did students in a course that introduced the system only briefly and gave it to students as a voluntary learning aid. Lattu et al. conclude that using Jeliot I forces teachers to rethink how they teach



```
import jeliot.io.*;
class Example {
    static void main() {
        int input = Input.readInt();
        double real = 20;
        int i = 0;
        while ( real > input) {
            real = real * 0.4;
            i = i + 1;
            Output.println(i);
        }
    }
}
```

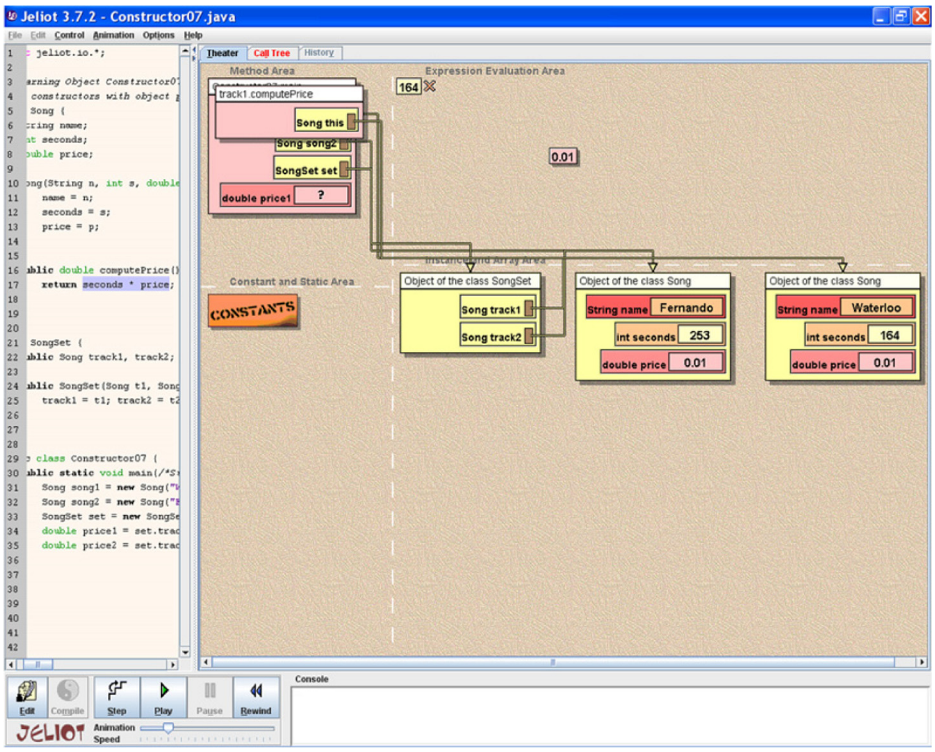
Fig. 24. A snapshot of a Java program being executed in Jeliot 2000 [Ben-Bassat Levy et al. 2003]. The code is shown on the left. On its right is shown the topmost frame of the call stack, with local variables. A conditional has just been evaluated in the top right-hand corner.

programming. Despite the positive experiences, Lattu et al. [2000, 2003] also found problems with bringing Jeliot I to a CS1 context: the GUI was too complex for some novices to use, and many aspects of program execution, which novices would have found helpful to see, were left unvisualized (e.g., objects and classes).

Jeliot 2000 [Ben-Bassat Levy et al. 2003] was a reinvention of Jeliot as a more beginner-friendly tool, with complete automation and a more straightforward GUI (Figure 24). In Jeliot 2000, the user did not control the appearance of the visualization or what aspects of execution were animated. Instead, Jeliot 2000 automatically visualized Java program execution in a detailed and consistent manner, all the way down to the level of expression evaluation. Control decisions were shown as explanatory text (see Figure 24). The user stepped through the animation using control buttons reminiscent of a household remote control. Jeliot 2000 did not support object-oriented concepts, although references to array objects were graphically displayed as arrows.

Ben-Bassat Levy et al. studied introductory programming students using Jeliot 2000 in a year-long course. They found that students using the system significantly improved the results of the students who used it, with an apparent “middle effect”.

> “Even in long-term use, animation does not improve the performance of all students: the better students do not really need it, and the weakest students are overwhelmed by the tool. But for many, many students, the concrete model offered by the animation can make the difference between success and failure. Animation does not seem to harm the grades neither of the stronger students who enjoy playing with it but do not use it, nor of weaker students for whom the animation is a burden. . . . The consistent improvement in the



```
: jeliot.io.*;

arning Object Constructor0
 constructors with object 
 Song {
tring name;
nt seconds;
ouble price;

ong(String n, int s, double
   name = n;
   seconds = s;
   price = p;


ublic double computePrice()
   return seconds * price;



 SongSet {
ublic Song track1, track2;

ublic SongSet(Song t1, Song
   track1 = t1; track2 = t2



 class Constructor07 {
ublic static void main(/*S
   Song song1 = new Song("
   Song song2 = new Song("
   SongSet set = new SongSe
   double price1 = set.trac
   double price2 = set.trac
```

Fig. 25. Jeliot 3 executing a Java program. The visualization extends that of Jeliot 2000 (Figure 24). The main novelty of Jeliot 3 is its support for object orientation. This picture [from Ben-Ari et al. 2011], also shows how active method calls are stacked on top of each other in the Method Area.

> average scores of the mediocre students confirms the folk-wisdom that they are the most to benefit from visualization." [Ben-Bassat Levy et al. 2003]

As the improvement in grades occurred only some way into the course, Ben-Bassat Levy et al. conclude that "animation must be a long-term part of a course, so that students can learn the tool itself". Ben-Bassat Levy et al. further found that the students who used Jeliot 2000 developed a different and better vocabulary for explaining programming concepts such as assignment than did a control group that did not use Jeliot. This, the authors remind us, is particularly significant from the socio-linguistic point of view, according to which verbalization is key to understanding.

Jeliot 2000 was improved upon by adding support for objects and classes to produce Jeliot 3 [Moreno and Myller 2003; Moreno et al. 2004]. Jeliot 3, shown in Figure 25, can be used as a standalone application or as a plugin for the pedagogical IDE BlueJ [Myller et al. 2007a]. Kirby et al. [2010] created a variation of Jeliot for visualizing C programs by combining the user interface of Jeliot 3 with a C++ interpreter taken from the VIP tool (see below) [Kirby et al. 2010]. Jeliot has also been extended to generate prediction questions that students can answer with their mobile phones in classroom situations Pears and Rogalli [2011a, 2011b].

A number of studies have investigated Jeliot 3 in practice.

Three small-scale studies have explored students' use of Jeliot 3 in first-year programming courses [Kannusmäki et al. 2004; Moreno and Joy 2007; Sivula 2005]. Kannusmäki et al. studied the use of Jeliot 3 in a distance education CS1 course, in which using the tool was voluntary. They report that their weakest students in particular found Jeliot helpful for learning about control flow and OOP concepts, but some other students chose not to use the tool at all. Moreno and Joy's students found Jeliot 3 easy to use, and most of those who tried it continued to use it voluntarily for debugging. Despite this, Moreno and Joy found that their students did not always understand Jeliot's animations and failed to apply what they did understand. Sivula also reports positive effects of Jeliot on motivation and program understanding, but points to how the students he observed did not use Jeliot's controls effectively for studying programs and ignored much of what might have been useful in the visualization.

Ebel and Ben-Ari [2006] used Jeliot 3 in a high-school course, and report that the tool brought about a dramatic decrease in unwanted pupil behavior. Ebel and Ben-Ari's results came from studying a class whose pupils suffered from “a variety of emotional difficulties and learning disabilities,” but had normal cognitive capabilities. Although it is unclear how well this result can be generalized to other contexts, the study does indicate that program visualization can help students focus on what is to be learned.

Bednarik et al. [2006] used eye-tracking technology to compare the behavior of novices and experts who used Jeliot 3 to read and comprehend short Java programs. They found that experts tested their hypotheses against Jeliot's animation, using Jeliot as an additional source of information. Novices, on the other hand, *relied* on the visualization, interacting with the GUI and replaying the animation more. They did not read the code before animating it.

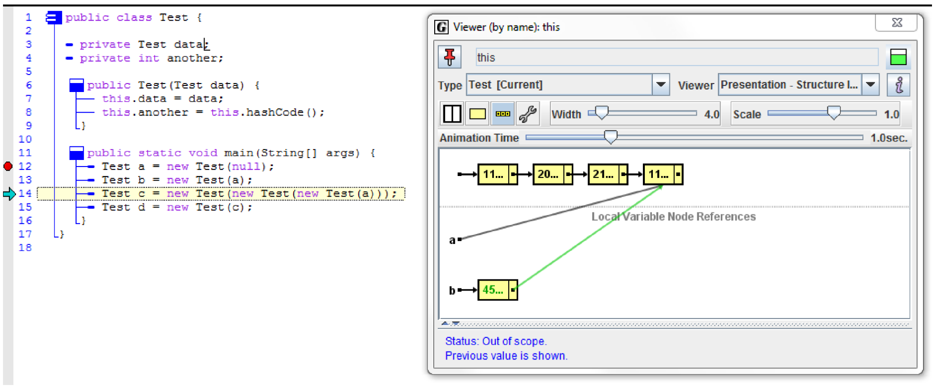
Sajaniemi et al. [2008] studied the development of student-drawn visualizations of program state during a CS1. Parts of their findings concern the effect of visualization tools on such drawings. Some of the students used Jeliot 3 for part of the course, then switched to the metaphorical OO animations (see above), while the others did the reverse. Irrespective of the group, the great majority of students did not use visual elements that were clearly taken from the PV tools. When they did, students' visualizations appeared to be mostly influenced by whichever tool they had used most recently. [Sajaniemi et al. 2008] discuss the differences in some detail, pointing out, for instance, that Jeliot users tended to stress expression evaluation more, but made more errors when depicting objects and their methods.

Myller et al. [2009] investigated the use of Jeliot 3 in a collaborative learning setting. CS1 students worked in small groups in a number of laboratory sessions, during which their behavior was observed by the researchers. They found that students were especially interactive when they were required to engage with Jeliot 3 by entering input to the program. Viewing the visualization made students particularly uninteractive, even when compared to moments where they were not viewing a visualization of program execution at all. Myller et al. conclude that having students merely view a visualization is not a good idea as it reduces collaboration.

Ma et al. [2009, 2011] observed that Jeliot 3 helped many of their students to learn about conditionals, loops, scope, and parameter passing, but that they found its visualization of references to be too complex. The authors contend that a simpler visualization system tailored for the specific topic of object assignment seemed to be more suitable for the task.

Maravić Čisar et al. [2010, 2011] used experimental studies in two consecutive years to evaluate the impact of Jeliot 3 on a programming course. During the course, some students used Jeliot for programming and debugging in the place of a regular IDE (which includes a visual debugger). In code comprehension post-tests, the students who used Jeliot performed significantly better than the control groups. Student feedback was also positive.

Wang et al. [2012] report a study in which a group of students were first shown textual explanations of how a program execution and then (separately) an animation



```
 1  public class Test {
 2
 3     private Test data;
 4     private int another;
 5
 6     public Test(Test data) {
 7        this.data = data;
 8        this.another = this.hashCode();
 9     }
10
11     public static void main(String[] args) {
12        Test a = new Test(null);
13        Test b = new Test(a);
14        Test c = new Test(new Test(new Test(a)));
15        Test d = new Test(c);
16     }
17  }
18
```

Viewer (by name): this

this

Type: Test [Current]   Viewer: Presentation - Structure I...

Width 4.0   Scale 1.0

Animation Time 1.0sec.

Status: Out of scope.
Previous value is shown.

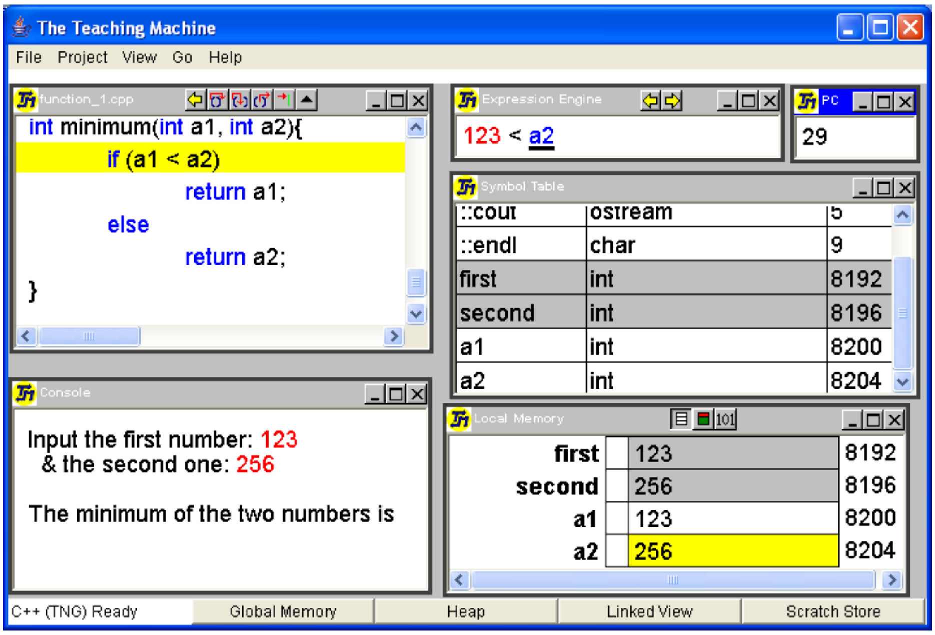
Fig. 26. An object viewer window within the jGRASP debugger. Control has just returned from the third constructor call on line 14, and a reference is about to be assigned to variable c.

of the program in Jeliot, while another group first viewed the animation and then saw the explanations. Wang et al. found that the explanations-first group outperformed the animations-first group in a post-test that required them to explain programming concepts.

5.3.6. Other Multi-Language Animators: jGRASP, The Teaching Machine, ETV, and HDPV. *GRASP* [Cross et al. 1996], later *jGRASP* [Cross et al. n.d., Cross et al. 2011], is another long-lived software visualization system. The system is an IDE that features a wide array of enhancements for the benefit of the novice programmer, such as the static highlighting of control structures in code, and conceptual visualization of data structures. Of most interest for present purposes is the aspect that falls within the domain of program animation: jGRASP's visual debugger features "object viewers", which can visualize not only certain high-level data structures but also runtime entities of CS1 interest, such as arrays, objects in general, and instance variables [Cross et al. 2011]. The original GRASP was meant for Ada programming; jGRASP has full support for several languages, including Java and C++. Experimental evaluations of jGRASP have so far revolved around its AV support in courses on data structures.

Like the later incarnations of Jeliot, *The Teaching Machine* [Bruce-Lockhart and Norvell 2000] is a system for visualizing the execution of user code at a detailed level. The Teaching Machine visualizes C++ and Java programs in a way that is similar to, but richer than, a regular visual debugger (Figure 27). In particular, the novice can choose to view the stages of expression evaluation in detail to aid comprehension. Stepping back within the execution is also supported. Moreover, as shown in Figure 28, The Teaching Machine is capable of automatically creating dynamic diagrams of the relationships between data. Later versions have added support for teacher-defined popup questions [Bruce-Lockhart et al. 2009]; this development appears to have been primarily motivated by a wish to quiz students about algorithms on a higher level of abstraction. The Teaching Machine can be used as a plugin within the Eclipse IDE.

The Teaching Machine has been used by its authors in various programming courses, who provide anecdotal evidence of how the system helped them teach in class [Bruce-Lockhart and Norvell 2007; Bruce-Lockhart et al. 2007]. Their students liked the system, especially when it was tightly integrated with course notes. However, while the students of more advanced courses used The Teaching Machine on their own to study programs with apparent success, CS1 students tended to leave the tool alone and only viewed it while the instructor was using it. Bruce-Lockhart and Norvell [2007] report on a modified CS1 course that did not work very well, in which



```
int minimum(int a1, int a2){
        if (a1 < a2)
                return a1;
        else
                return a2;
}
```

```
123 < a2
```

```
29
```

::cout	ostream	5
::endl	char	9
first	int	8192
second	int	8196
a1	int	8200
a2	int	8204

```
Input the first number: 123
 & the second one: 256

The minimum of the two numbers is
```

first	123	8192
second	256	8196
a1	123	8200
a2	256	8204

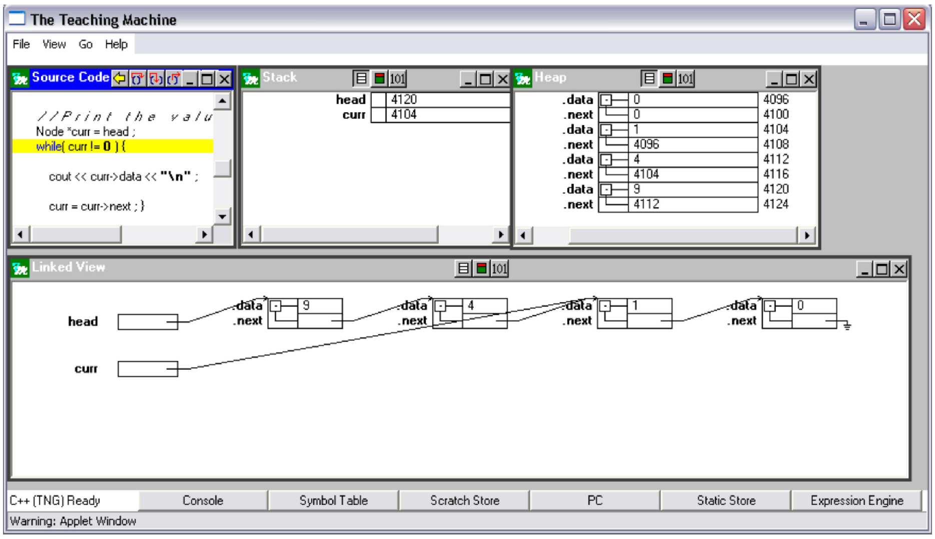
Fig. 27. The Teaching Machine executing a C++ program [Bruce-Lockhart et al. 2007]. The view resembles that of a regular visual debugger, the main difference being the Expression Engine at the top, which displays expression evaluation in detail. The part of the expression that will be evaluated next is underlined.

"we found at the end of the course that most of the first year students had never run [The Teaching Machine] on their own and consequently didn't really understand how either the [system], or the model it was based on, worked." However, after a suitable emergency intervention—extra labs in which students had to work hands-on with the system—at the end of the course, "student response was overwhelmingly positive and the course was at least rescued."

Terada [2005] described a tool called *ETV*, which was capable of producing visual traces of programs written in one of a variety of languages. ETV supported stepping both forwards and backwards, and visualized the call stack using cascading windows of source code (akin to VisMod, above). Terada [2005] used ETV in a first-year programming course, and reports a mix of student opinions of the tool.

Another multi-language program animator for C/C++ and Java, *HDPV*, was presented by Sundararaman and Back [2008]. Like systems such as Jeliot and jGRASP (above), HDPV displayed data in memory as an abstract diagram. In HDPV, users could only step forward in the execution trace, but had access to various advanced controls (e.g., zoom, pan, collapse, move elements) that enabled them to choose what to view. Like jGRASP, HDPV also featured algorithms for the automatic layout of certain common data structures.

5.3.7. Python in the Modern Browser: Jype and the Online Python Tutor. *Jype* is a web-based integrated development environment for Python programming [Helminen 2009; Helminen and Malmi 2010]. In addition to serving as an IDE, it can be used for exploring teacher-given example programs and as a platform for the automatic assessment of given programming assignments. Jype is intended specifically for CS1 use and has a number of beginner-friendly features, among them support for program and algorithm visualization. The system uses the Matrix framework [Korhonen et al. 2004] for



```
//Print the valu
Node *curr = head ;
while( curr != 0 ) {

   cout << curr->data << "\n" ;

   curr = curr->next ; }
```

```
head  4120
curr  4104
```

```
.data  0     4096
.next  0     4100
.data  1     4104
.next  4096  4108
.data  4     4112
.next  4104  4116
.data  9     4120
.next  4112  4124
```

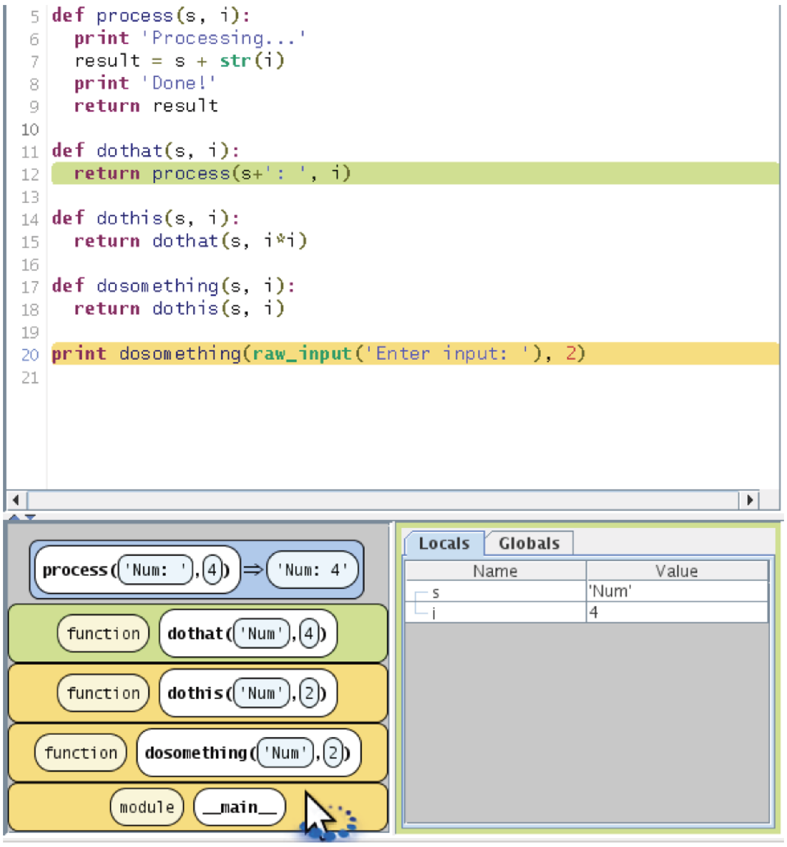
Fig. 28. Another layout of windows within The Teaching Machine [Bruce-Lockhart et al. 2007]. The a stack frame and the heap are explicitly shown, along with a linked view that graphically displays relationships between data.

automatically visualizing data structures such as arrays and trees when they appear in programs. Jype also visualizes the call stack in a richer way than a typical visual debugger, as illustrated in Figure 29.

Another new, web-based system for visualizing small Python programs is Online Python Tutor [Guo 2013], shown in Figure 30. Like Jype, it allows the user to step forward and backwards in a Python program at the statement level, and supports the distribution of teacher-given examples and small programming assignments with automatic feedback. The visualizations created by Online Python Tutor can be embedded onto web pages, and have been integrated into an online textbook, for instance. The system has many users; it has been used by thousands of students taking massive open online courses (MOOCs) [Guo 2013].

5.3.8. Systems for Functional Programming: 'Typical Functional Debuggers' and ZStep95. There are numerous debuggers for functional programs that allow the user to step through the stages of expression evaluation; some of these are explicitly education-oriented. An early system for visualizing the dynamics of functional programs (in the Miranda language) was presented by Auguston and Reinfelds [1994]. Mann et al. [1994] reported a study using LISP Evaluation Modeler, which traced the evaluation of Lisp expressions. They found positive transfer effects from the use of the system for debugging, and their students were eager to use it. ELM-ART [see, e.g., Weber and Brusilovsky 2001] is an intelligent Lisp programming tutor for beginners, which features a component that visualizes expression evaluation. The DrScheme/DrRacket IDE also uses some visual elements to present program dynamics, among its other beginner-friendly features [see, e.g., Findler et al. 2002]. (Systems of this kind appear as "typical functional debuggers" in Tables IV and V.)

ZStep95 was an untypical Lisp debugging environment suitable for educational use [Lieberman and Fry 1997]. It was an evolution of ZStep84 [Lieberman 1984]. ZStep95 allowed stepping backward and forward in a program and visualized expression evaluations in a floating window positioned on top of the evaluated code, thus



```
def process(s, i):
  print 'Processing...'
  result = s + str(i)
  print 'Done!'
  return result

def dothat(s, i):
  return process(s+': ', i)

def dothis(s, i):
  return dothat(s, i*i)

def dosomething(s, i):
  return dothis(s, i)

print dosomething(raw_input('Enter input: '), 2)
```

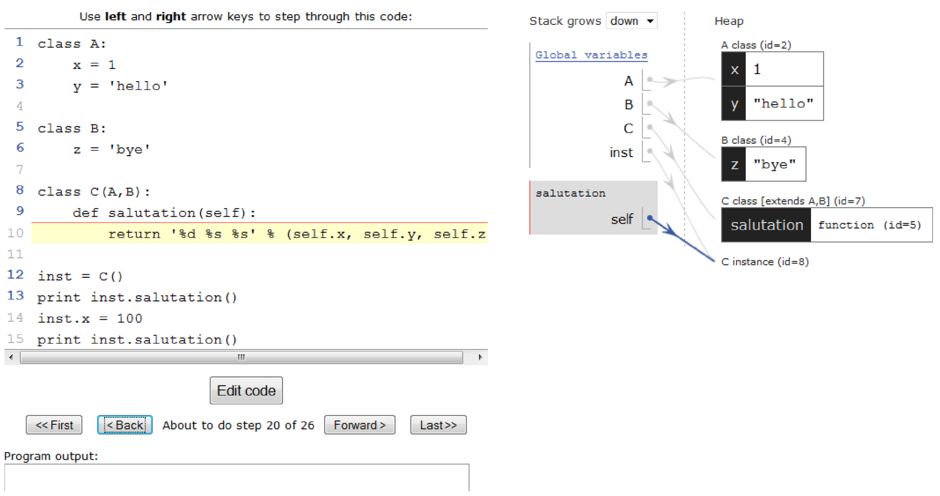
Name	Value
s	'Num'
i	4

Fig. 29. A part of Jype's user interface. The call stack is visualized at the bottom left. The blue box signifies returning from a function.

reducing the need to move gaze from source code to an animation area. A program run could also be controlled by jumping directly to a point where a selected expression is evaluated. Previous values of expression evaluations could be selected for viewing, as could the expressions that produced in a certain graphical output. We are not aware of an evaluation of ZStep in an educational context.

5.3.9. Producing Snapshots: Kasmarik and Thurbon's Tool, CMeRun, and Backstop. Kasmarik and Thurbon [2003] used an unnamed system that could produce visualizations of specific program examples. Their tool took Java code as input and produced a sequence of graphical diagrams to illustrate the given program's states, in particular the values that variables get at different stages of execution. The tool's authors evaluated their visualizations (which had been created by the teacher using the tool) experimentally in a CS1 course. They had students answer questions about small example programs, which were additionally accompanied by visualizations in the case of the treatment group. The treatment group's results were significantly better than the control group's, without a significant increase in the time they spent on the task.

Etheredge [2004] described a tool called CMeRun, designed to aid novice programmers debug their programs. CMeRun instrumented C++ code in such a way that when the code was run, an execution trace was printed out. The user could examine the trace



```
class A:
    x = 1
    y = 'hello'

class B:
    z = 'bye'

class C(A,B):
    def salutation(self):
        return '%d %s %s' % (self.x, self.y, self.z

inst = C()
print inst.salutation()
inst.x = 100
print inst.salutation()
```

Fig. 30. A program running in Online Python Tutor.

to see which statements were executed and what the values of variables were at each stage. For instance, an execution of the line

```
for (i = 0; i <= SIZE; i++)
```

might appear in the trace of a CMeRun-augmented program as

```
for (i = 0; i<3> <= SIZE<4>; i<3>++)
```

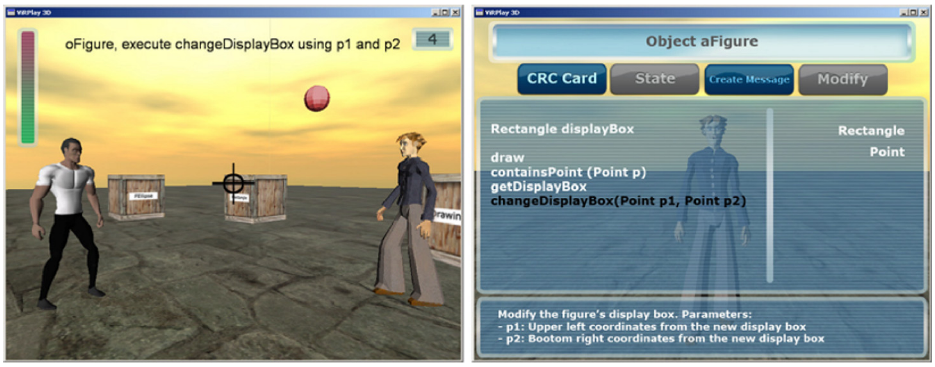
Backstop [Murphy et al. 2008] produces descriptions of Java program runs in a similar way. The system also has additional features intended to ease debugging, such as beginner-friendly explanations of runtime exceptions.

Etheredge [2004] and Murphy et al. [2008] have reported positive responses to CMeRun and Backstop from teacher and student evaluators.

5.4. Applying a Visualization

The seven systems listed in this section can all be used at the applying level of the 2DET's direct engagement dimension (the highest level explicitly supported by the systems we reviewed). In some of them, this is the primary way to use the system, while in others it is one of several modes of use. Overall, there is quite a bit of variety among these seven systems, each of which implement applying in a distinctive way. Nevertheless, with the exception of Gilligan's system and WinHIPE, the systems all provide interactions that involve the applying of given content in a way that can be loosely characterized as visual program simulation (as defined above in Section 3).

5.4.1. Programming-by-Demonstration to Learn About the Machine: Gilligan's System. *Programming by demonstration* is a visual programming paradigm that uses expressive GUIs to empower end-users to program without having to write program code: "the user should be able to instruct the computer to "Watch what I do," and the computer should create the program that corresponds to the user's actions" [Cypher 1993]. By and large, programming-by-demonstration systems, even those whose target audience is programming beginners, fall into the empowering systems category in Pausch's [2005] taxonomy (see Section 3) and out of the scope of this review. However, we are aware of one programming-by-demonstration system whose goals were different enough to justify an exception.



oFigure, execute changeDisplayBox using p1 and p2

Rectangle displayBox

draw
containsPoint (Point p)
getDisplayBox
changeDisplayBox(Point p1, Point p2)

Modify the figure's display box. Parameters:
- p1: Upper left coordinates from the new display box
- p2: Bootom right coordinates from the new display box

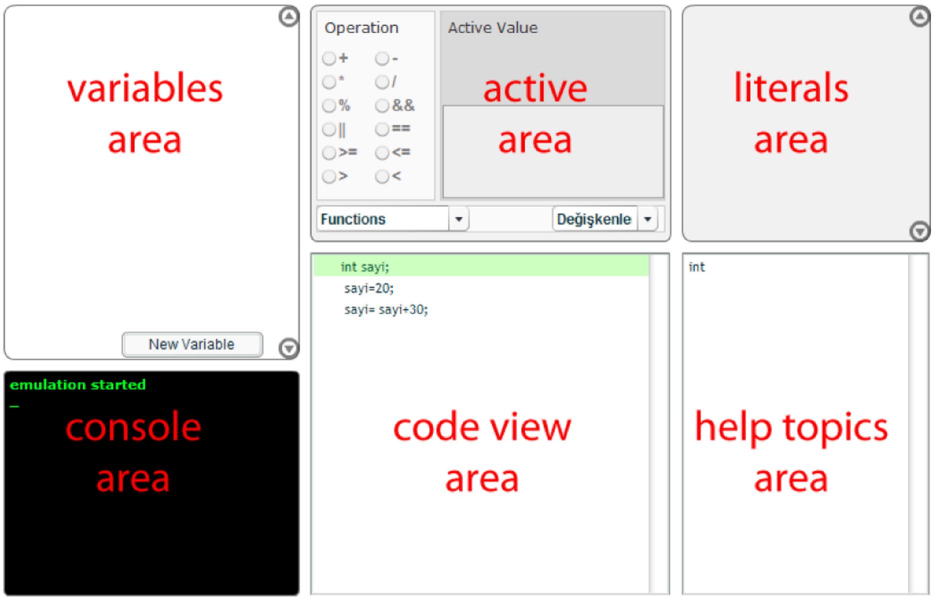
Fig. 31. Students interact within the virtual world presented by ViRPlay3D2 to design object-oriented programs [Jiménez-Díaz et al. 2008]. Two separate screenshots are shown here. On the left, a user is viewing message passing in action between two other objects (avatars) from a first-person perspective. Passing a message is graphically represented as throwing a ball to the recipient. On the right, a user is examining another object's Class-Responsibility-Collaboration card [Beck and Cunningham 1989], which describes how he can interact with the object.

Gilligan [1998] created a prototype of a programming-by-demonstration system for novice programmers. The system aimed not only to provide an accessible way of expressing programs but also to explicitly teach a model of computation in the process. It used analogies to everyday objects to present a user interface through which the novice programmer expresses what they wish their program to do: the computer's math processor and logic unit are represented by a calculator, a stack of initially blank paper represents memory, a clipboard with worksheets represents the call stack, and so forth.

The user of Gilligan's system—the programmer—takes on the role of a clerk who intends to accomplish a task using this equipment according to certain rules. In doing so, the user produces a sequence of actions that defines a program. Using the calculator produces an arithmetical or logical expression, for instance, and adding a new worksheet to the clipboard starts a subroutine call. The system writes and displays the resulting program as Pascal code that matches the user's interactions with the GUI. By engaging in these clerical activities, the user would learn about their correspondence to the execution model of Pascal programs and would—hypothetically, at least—be better equipped to transition to regular programming later.

The system was never evaluated in practice, as far as we know. A later prototype extended Gilligan's work to object-oriented Java programming [Deng 2003].

5.4.2. Role-Playing Objects: ViRPlay3D2. Jiménez-Díaz et al. [2005] presented early work on a 3D environment which was intended to allow students to role-play and learn to understand the (inter)actions of the objects of a given OOP program as it runs. This project has since evolved toward having groups of students role-play object actions in order to *specify* the actions of an OOP program that is to be created—an activity analogous to visual programming on a higher level of abstraction. Jiménez-Díaz et al. [2008, 2011] used a system called ViRPlay3D2 (Figure 31) to provide a virtual three-dimensional world in which students control avatars that each represent an object in the execution of an object-oriented program. Each student tries to follow the instructions specific to his or her kind of object. One of the students at a time is active and can delegate tasks to other objects by passing messages to them. The goal is to collaboratively produce a working object-oriented software design for a problem and to verify that it works. Students can choose to change the definitions of classes to improve their design. Execution simulations can be saved and reviewed. The system also features a



```
int sayi;
sayi=20;
sayi= sayi+30;
```

Fig. 32. Dönmez and İnceoğlu's [2008] visual program simulation system.

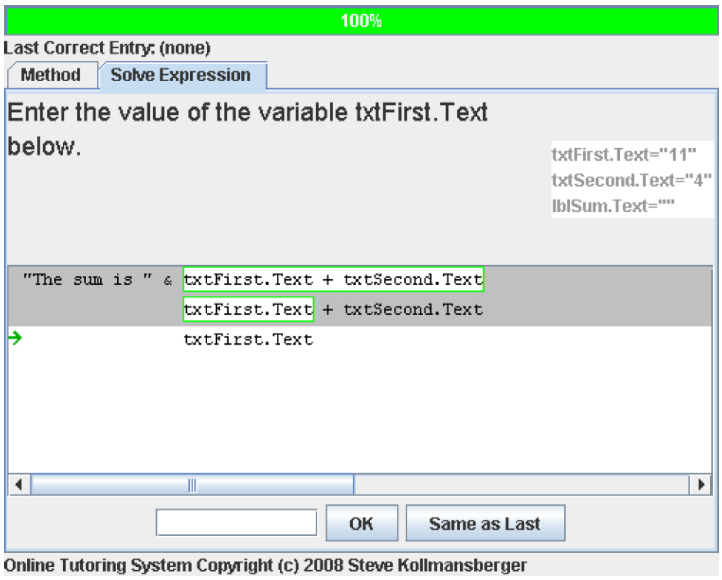
"scripted mode" in which the characters act automatically instead of being controlled by students. In addition to designing programs, ViRPlay3D2 can also be used to examine case studies of program design; in this usage, it provides a highly abstract form of visual program simulation.

The results of an experimental study did not show a significant difference between ViRPlay3D2 users and a control group that role-played object-oriented scenarios without the help of a software system [Jiménez-Díaz et al. 2011]. Both students and instructors liked the software.

5.4.3. Learner-Controlled Execution: Dönmez and İnceoğlu's Tool. Dönmez and İnceoğlu [2008] present a visual program simulation system prototype: their tool has the student take an active role in program execution in order to improve their understanding of the notional machine that underlies C# programming. Using the tool's GUI, students simulate the execution of code they have written in a limited subset of C#. Students use the GUI to evaluate arithmetical and logical expressions, and to create and assign to variables. The view, shown in Figure 32, resembles that of a regular visual debugger, with one major difference: there is no way to make the computer run or step through the program. Instead, GUI controls are present that allow and require the user to indicate what happens when the program is run. For instance, to create a new variable, the user is expected to press the New Variable button, type in the variable's name in the dialog that pops up, select its type, and click OK to close the dialog.

Dönmez and İnceoğlu's system could only handle certain simple kinds of programs. Many fundamental topics with complex runtime dynamics, such as functions (except for a handful of built-in single-parameter functions, as above), references, and objects, are not supported. To our knowledge, no evaluation of the system has been reported.

5.4.4. Text-Driven Simulation: Online Tutoring System. Kollmansberger [2010] developed and used another program visualization tool that can be characterized as a visual program simulation system. His Online Tutoring System (Figure 33) presented students with short ready-made programs written in Visual Basic for Applications. Similar to


100%
Last Correct Entry: (none)
Method
Solve Expression
OK
Same as Last
Online Tutoring System Copyright (c) 2008 Steve Kollmansberger

Enter the value of the variable txtFirst.Text below.

```
txtFirst.Text="11"
txtSecond.Text="4"
lblSum.Text=""
```

```
"The sum is " & txtFirst.Text + txtSecond.Text
                txtFirst.Text + txtSecond.Text
                txtFirst.Text
```

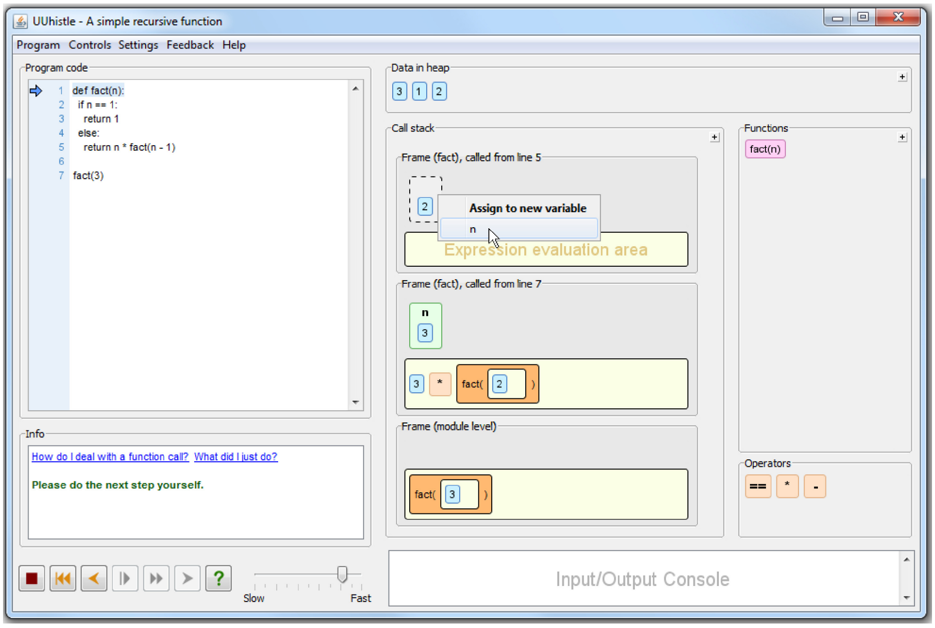
Fig. 33. An assignment on string catenation within the Online Tutoring System [Kollmansberger 2010]. The student has first chosen the expression `txtFirst.Text + txtSecond.Text` for evaluation, then the subexpression `txtFirst.Text`. Now the system informs them that the next step is to access the variable: consequently the user has to look at the values in memory (on the right) and type in `"11"` (in quotes, to highlight that it is a string). Subsequent steps include typing in `"4"` and `"114"` at the appropriate moments.

Dönmez and İnceoğlu's tool, the Online Tutoring System required the student to predict which statement the computer will execute next (by clicking on the correct line), and to indicate the order in which the parts of the statement are dealt with (by clicking on the appropriate part of the code). Furthermore, the student must type in the resulting values at every stage of expression evaluation and assignment. The system provided textual prompts at each step of the way to indicate what type the next step should be (e.g., subexpression selection or accessing a variable). When the student got an answer wrong, the system told them the correct answer, which the student then had to reproduce within the system in order to proceed. Incorrect answers were reflected in the student's grade for the assignment, but the same assignment could be tried repeatedly.

Kollmansberger reports a high course completion rate for two sections of a CS1 class that used the Online Tutoring System for 12 additional exercises at various points of the course. According to opinion surveys, the students liked it for the most part, although some found the user interface unwieldy and the exercises too repetitive.

5.4.5. Different Modes of Engagement: UUhistle. UUhistle is a program visualization system for CS1 that supports different modes of user interaction [Sorva 2012; Sorva and Sirkiä 2010]. UUhistle visualizes a notional machine for the Python programming language; the notional machine and the visualization largely resemble those of Jeliot 3, although UUhistle makes the call stack more explicit and focal.

UUhistle can be used for animating existing programs so that the user controls the pace of the execution. In an "interactive coding mode," UUhistle interprets and visualizes each statement as soon as the user types it in (cf. the DISCOVER system above). Moreover, UUhistle can be used as a platform for different visualization-based assigments. Teachers may embed popup questions into example programs for students to answer; they may also configure example programs as visual program simulation

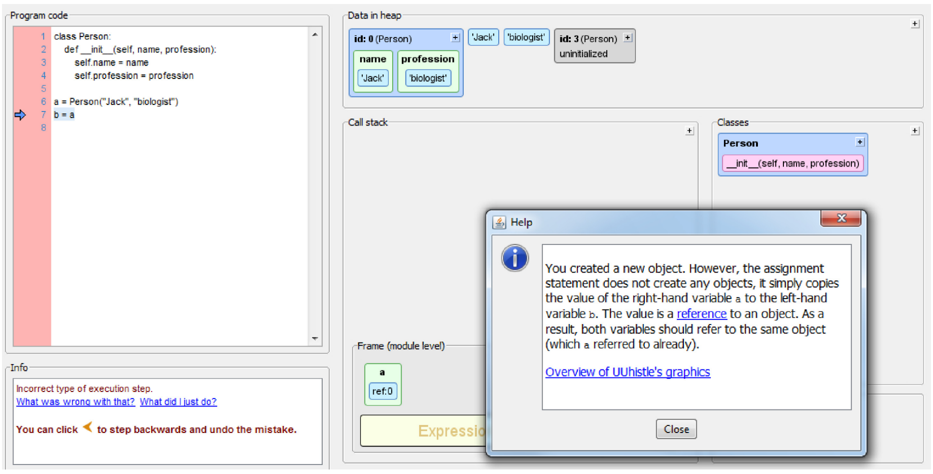

UUhistle - A simple recursive function
Program Controls Settings Feedback Help
Program code
Data in heap
Call stack
Functions
Frame (fact), called from line 5
Assign to new variable
n
Expression evaluation area
Frame (fact), called from line 7
Frame (module level)
fact(n)
Operators
Info
Slow
Fast
Input/Output Console

```
def fact(n):
  if n == 1:
    return 1
  else:
    return n * fact(n - 1)

fact(3)
```

How do I deal with a function call? What did I just do?

Please do the next step yourself.

Fig. 34. A visual program simulation exercise in UUhistle. The user is manually controlling the execution of a small recursive program, and has just dragged a parameter value into the topmost stack frame in order to create a new variable there. He is just about to name the variable using the context menu that has popped up. The Info box in the lower left-hand corner provides links to additional materials.



```
class Person:
    def __init__(self, name, profession):
        self.name = name
        self.profession = profession

a = Person("Jack", "biologist")
b = a
```

You created a new object. However, the assignment statement does not create any objects, it simply copies the value of the right-hand variable a to the left-hand variable b. The value is a reference to an object. As a result, both variables should refer to the same object (which a referred to already).

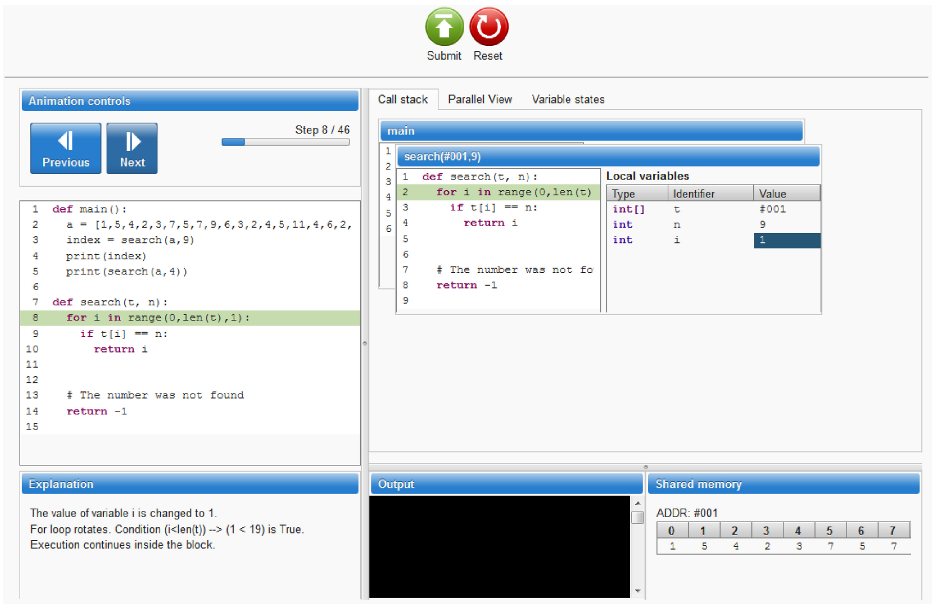
Overview of UUhistle's graphics

Incorrect type of execution step.
What was wrong with that? What did I just do?

You can click < to step backwards and undo the mistake.

Fig. 35. A UUhistle user has created a new object when they should have merely formed another reference to an existing object. UUhistle's feedback seeks to address a suspected misconception concerning object assignment.

exercises in which the learner directly manipulates the graphics using the mouse to execute a given program (cf. Dönmez and İnceoğlu's system and Online Tutoring System above). Examples of visual program simulation exercises in UUhistle are shown in Figures 34 and 35. An exploratory study has suggested that students' simulation behavior in UUhistle can be a useful source of information about the



```
def main():
  a = [1,5,4,2,3,7,5,7,9,6,3,2,4,5,11,4,6,2,
  index = search(a,9)
  print(index)
  print(search(a,4))

def search(t, n):
  for i in range(0,len(t),1):
    if t[i] == n:
      return i


  # The number was not found
  return -1

```

```
def search(t, n):
  for i in range(0,len(t)
    if t[i] == n:
      return i


  # The number was not fo
  return -1

```

Local variables

Type	Identifier	Value
int[]	t	#001
int	n	9
int	i	1

The value of variable i is changed to 1.
For loop rotates. Condition (i<len(t)) --> (1 < 19) is True.
Execution continues inside the block.

ADDR: #001

0	1	2	3	4	5	6	7
1	5	4	2	3	7	5	7

Fig. 36. A Python program within ViLLE.

students' misconceptions [Sirkiä and Sorva 2012], and UUhistle provides automatic feedback that seeks to address programming misconceptions that students' mistakes suggest they may have [Sorva 2012; Sorva and Sirkiä 2011, and see Figure 35]. The system can automatically grade students' solutions and submit them to courseware.

Sorva and his colleagues conducted mixed-methods research on an early UUhistle prototype, focusing in particular on visual program simulation exercises [Sorva et al., accepted; Sorva 2012]. They found that there is significant variation in how learners perceive the visualization and VPS exercises; for instance, some learners do not always perceive the visualization to represent computer behavior at all, and even when they do, may fail to associate what they see in UUhistle with the broader context of programming. On the other hand, others made these important connections and learned substantially from the system. The variance in students' emotional responses matches this variety of ways of understanding VPS. Students were further found to use a mix of tactics for solving VPS assignments, ranging from surface approaches and guesswork to reasoning about the content of the visualization. In a controlled experiment, a short VPS session in UUhistle resulted in improvement in students' ability to trace programs featuring function calls and parameters—but not other programs—compared to a control group. A complementary qualitative analysis suggested that the effectiveness of VPS exercises is likely to depend on a successful alignment of specific learning goals with the specific GUI interactions required of the learner.

5.4.6. Variety in Assignments: ViLLE. ViLLE is an online learning platform which started out as a program visualization system for CS1 that displays predefined program examples to students statement by statement [Rajala et al. 2007; Rajala et al. nd]. Our review focuses on ViLLE's program visualization functionality.

ViLLE comes with a selection of example programs; teachers can also define and share their own. The user can choose to view a program within ViLLE as either Java, C++, Python, PHP, JavaScript, or pseudocode, and change between languages at will. ViLLE supports a limited "intersection" of these languages.

ViLLE's program visualizer has several beginner-friendly features beyond what a regular visual debugger offers, some of which are shown in Figure 36. The user can step forward or backwards. Each code line is accompanied by an automatically generated explanation. Arrays are graphically visualized. Teacher-defined popup questions can be embedded into programs to appear at predefined points of the execution sequence. ViLLE can grade the answers to multiple-choice questions in the popup dialogs and communicate with a course management system to keep track of students' scores [Kaila et al. 2008].

Apart from multiple-choice questions, ViLLE also supports a few other assignment types (Rajala et al., http://ville.cs.utu.fi). Students may be required to sort lines of code that have been shuffled or to fill in some Java code to complete a program. Recent versions of ViLLE feature a "Clouds & Boxes" assignment type, which is essentially a simple form of visual program simulation that uses standard widgets as controls: the user has to carry out certain aspects of program execution manually (e.g., frame allocation, variable creation, line changes) while the computer takes care of certain other aspects (e.g., expression evaluation).

ViLLE's authors have investigated the impact of the tool with a series of experimental studies that involve levels of direct engagement between no viewing and responding.

Laakso et al. [2008] found that previous experience with using ViLLE had a significant effect on how well high school students learned from given examples, and conclude that to get the most out of a visualization tool, students need to be trained to use it effectively. Rajala et al. [2008] compared the performance of a treatment group using ViLLE and a control group on code-reading tasks. They found no significant differences in post-test scores between the groups. Kaila et al. [2009a] found that having students respond to ViLLE's popup questions during program execution had a better impact on learning than merely having students view example programs. Another publication by the authors reports that found that introductory programming students learned more when using ViLLE in pairs than when working alone, especially with regard to more challenging topics such as parameter passing [Rajala et al. 2009]. A longer-term evaluation studied three consecutive introductory programming course offerings in high school, whose teaching was identical except for ViLLE being used throughout the course in the third offering; using ViLLE significantly raised course grades [Kaila et al. 2010]. According to an opinion survey, students tend to like the system [Kaila et al. 2009b]; see also Alsaggaf et al. [2012].

5.4.7. Configuring Illustrations of Functional Programs: WinHIPE. WinHIPE is an education-oriented IDE for functional programming that includes a program visualization functionality [Pareja-Flores et al. 2007]. The system is not specifically meant for beginners—indeed, the authors have used it in more advanced courses [Pareja-Flores et al. 2007; Urquiza-Fuentes and Velázquez-Iturbide 2007, 2012]—but is a plausible choice for a functional CS1 as well.

WinHIPE visualizes an execution model of pure functional programs that is based on rewriting terms (no assignment statements). The user configures each animation by selecting which aspects are to be visualized and in what order term-rewriting proceeds in the visualization. Using this facility, teachers can produce and customize dynamic visualizations of selected examples, which can be exported for students to view. When a visualization is being viewed, the evaluation of expressions is shown step by step, and the user can step back and forth in the evaluation sequence. WinHIPE's authors have also experimented with an activity in which students use WinHIPE to produce visualizations of given source code [Urquiza-Fuentes and Velázquez-Iturbide 2007, 2012]; this is a form of applying a visualization to given content.

5.5. Low-Level Approaches

The systems reviewed so far have represented notional machines that operate on high-level languages at various levels of abstraction that are not very close to the underlying hardware. One way to learn about program dynamics—albeit rare in CS1—is to first build a low-level foundations by visualizing an actual or simplified system that explains programming on the assembly language or bytecode level. Many existing program visualization tools might serve a purpose in such an endeavor.

Biermann et al. [1994] created a system, This is How A Computer Works, for visualizing Pascal programs to freshmen at different levels of abstraction below the code: a compiler level, a machine architecture level, and a circuit level. The simulator MPC1 presented a simplified computer in terms of processor operation codes, RAM cells, and registers [Sherry 1995]. The SIMPLESEM Development Environment [Hauswirth et al. 1998] and MieruCompiler [Gondow et al. 2010] also map high-level language semantics to assembly code. EasyCPU [Yehezkel et al. 2007] visualizes a model of computer components—registers, I/O ports, etc.—and highlights how each component is active in the execution of an assembly-language program. A different sort of low-level approach was implemented in ITEM/IP, which visualized the execution of a Pascal-like language in terms of a Turing machine [Brusilovsky 1992].

Some low-level systems have featured learner-controlled simulation of a very low-level machine represented within a highly abstract three-dimensional virtual world. In the CpuCITY system, the student used an avatar to perform hardware-level operations such as “take this packet to RAM” [Bares et al. 1998]. JV^{2}M applies a similar immersive approach to Java programming by having a student-controlled avatar perform in-game tasks that match the bytecode instructions corresponding to a given Java program [see, e.g., Gómez-Martín et al. 2005, 2006]. Figure 37 shows a screenshot of JV^{2}M.

Various other low-abstraction PV systems exist; the previously discussed systems are a fairly arbitrary selection. Many of the systems may be useful for the goal of teaching about low-level phenomena (which is indeed the stated goal of most of the systems). We will not review low-level systems in more detail, as our primary interest is in CS1 and systems that help students learn the runtime semantics of a high-level language. We consider systems that deal with assembly language too detailed to be practical for this purpose as the “black boxes” inside the “glass box” of the notional machine shown are too small and too numerous to achieve the necessary simplicity [cf. du Boulay et al. 1981].[5]

6. DISCUSSION

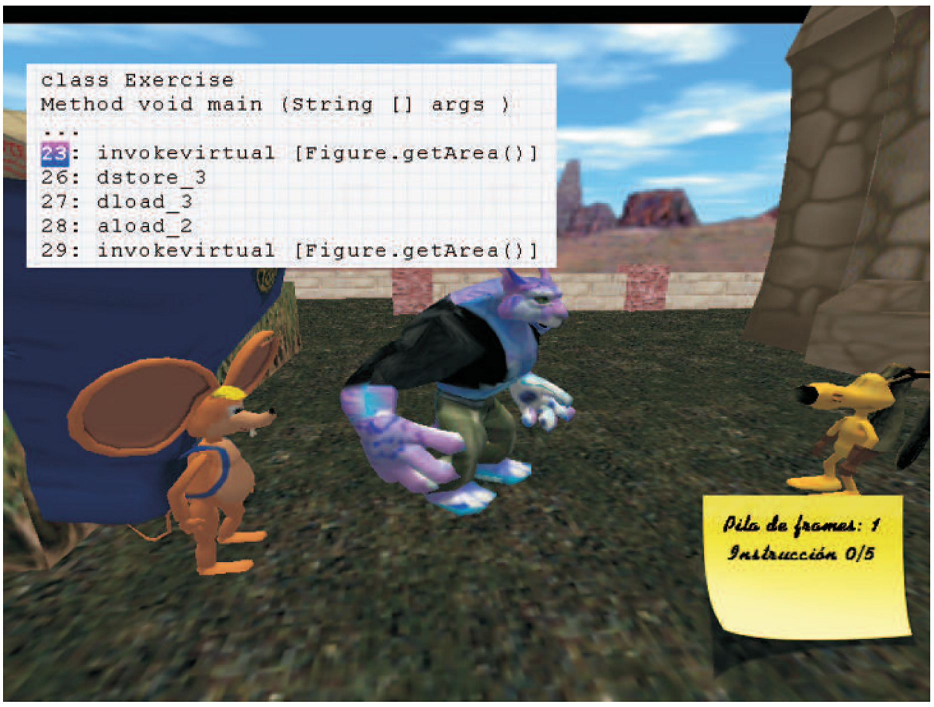
We comment briefly on a few themes that arise from our review.

6.1. System Characteristics Overviewed

The previous section illustrates the variety in the systems designed for teaching beginners about program dynamics, and also points to some trends and typical features within the set of such systems.

Most of the systems support a single programming language, often one that was a popular CS1 language at the time the system was created. Only some of the enduring systems support more than one language; the longest-lived ones (e.g., jGRASP, Jeliot, The Teaching Machine) all have evolved towards supporting multiple languages. The most common forms of representation in the systems are debugger-like

[5]We do think that exposing CS1 students to compiled code can be instructive, but as an additional measure, not as the main way to learn about a notional machine for a high-level paradigm.



```
class Exercise
Method void main (String [] args )
...
23: invokevirtual [Figure.getArea()]
26: dstore_3
27: dload_3
28: aload_2
29: invokevirtual [Figure.getArea()]
```

Fig. 37. The user is executing a compiled Java program in JV^{2}M [image from Gómez-Martín et al. 2006]. The bytecode instructions correspond to actions in the virtual world. The user's avatar is the big feline in the middle. The dog on the right is Javy, a helpful intelligent agent that provides advice.

standard widgets and abstract two-dimensional graphics; 3D and virtual worlds continue to be rare in systems of the kind we reviewed.

Controlled viewing is by far the most common form of direct engagement with visualization in the systems; in most, it is the only mode above viewing that is supported. Many of the systems work as extended visual debuggers that allow students to control the viewing of their own programs. The level of abstraction is commonly either that of a regular visual debugger, or just below so that the call stack, references/pointers, and expression evaluation are emphasized; much lower-level and higher-level visualizations also exist. Practically all the systems rely on a single, fixed level of abstraction in the visualization although they may allow different degrees of granularity in stepping through a program. Other common additions to regular debuggers include textual clarifications of execution steps and smooth animations that highlight the transitions between states. Modes of interaction with higher levels of direct engagement—responding and applying—are more common in recent systems than in older ones. None of the systems significantly features explicit support for presenting or creating.

6.2. The Problem of Dissemination

Most of the systems we reviewed appear to have been short-lived research prototypes that have been soon discarded once the system had been constructed or an evaluative study carried out. Very few of the systems have remained in active use and development for more than a few years. Only a small handful of systems have been used outside their site of creation; none can be said to belong in the mainstream of CS1 education. Open-source communities working on these systems are almost nonexistent.

Teachers' reasons for not adopting software visualization tools include Hundhausen et al. [2002], Naps et al. [2003], Ben-Bassat Levy and Ben-Ari [2007], and Kaila [2008]:

- the lack of teacher time for learning about systems or configuring them;
- poor fit with specific aspects of the course (e.g., programming language, teachers' personal pedagogical styles);
- attitudes toward software visualization;
- doubts about educational effectiveness (sometimes even the sentiment that SV tools further complicate topics that are already complicated);
- concern about SV taking away class time needed for other activities;
- lack of integration with other materials;
- the (perceived or real) poor usability of the tools;
- various detailed issues concerning specific visualizations.

A number of recent initiatives have sought to aid the dissemination of pedagogical practices within computing education in general and educational software visualization in particular. Forums have been launched for sharing [e.g., Fincher et al., n.d.; Korhonen, n.d.; Algoviz, n.d.], teachers have been given financial aid to lower collaboration thresholds [e.g., Korhonen, n.d.], and technical solutions have been developed to increase intercompatibility and extensibility so that tools are easier to integrate into different contexts (e.g., Karavirta [2007]; Moreno [2005]). Such initiatives can help promote the dissemination of PV tools; however, for CS1 teachers to make informed decisions about whether and how to use the tools, more empirical research is needed.

6.3. Empirical Evaluations and the Role of Engagement

Many of the visualization systems for CS1 have been evaluated informally, with positive teacher experiences and encouraging course feedback often being reported. However, student opinions are a notoriously treacherous measure of the effectiveness of a pedagogy [see, e.g., Clark 1982], and it is less clear whether—and how, for whom, under what circumstances, and at what cost—using the systems has resulted in significant learning gains. Various systems have also been evaluated more rigorously, either qualitatively or through controlled experiments. Usually, the evaluations have been carried out by the system authors themselves at their own institutions, often in their own teaching. Many of the findings from these studies have been positive, suggesting that the program visualizations have served a purpose, at least in the context for which they were crafted.

What about the role of engagement, identified as potentially so important by the software visualization community (Section 4)? To what extent do the hypothesized engagement effects apply to notional-machine visualization in CS1?

In Section 5, we referred to a number of studies that have evaluated the various program visualization systems. Table VI uses the 2DET to summarize those of the evaluations that sought to determine the relative effectiveness of different engagement levels. To produce Table VI, we made the following delimitations.

- The table lists only quantitative between-subject experiments that checked for the statistical significance of results.
- Only studies that used different modes of engagement as conditions are listed. (Not using a visualization at all does count as no viewing.) This excludes studies comparing two different tools (e.g., a regular visual debugger vs. an educationally oriented visualization) used in the same way.

Table VI. Experimental Evaluations of Program Visualization Tools Comparing Levels of Engagement

System	Source	Scope of experiment	Treatment	Control	Measure	Statistical significance?
two visual debuggers	Bennedsen and Schulte 2010	lab	controlled viewing / given content	no viewing / given content	reading tasks	no
Bradman	Smith and Webb 2000	lab	controlled viewing / given content	no viewing / given content	reading tasks	yes (in one task only)
VIP	Ahoniemi and Lahtinen 2007	lab	controlled viewing / given and modified content	no viewing / given and modified content	writing tasks	yes (among novices and strugglers)
ViLLE	Rajala et al. 2008	lab	controlled viewing (and responding?) / given content	no viewing / given content	reading and writing tasks	no
ViLLE	Kaila et al. 2009a	lab	responding / given content	no viewing and controlled viewing / given content	reading and writing tasks	yes
ViLLE	Kaila et al. 2010	course	responding / given content	no viewing / given content	reading and writing tasks	yes
UUhistle	Sorva 2012	lab	applying / given content	no viewing / given content	reading tasks	yes (in one task only)

— Only studies in which the participants learned about typical CS1 content are included.
— Studies that gave the experimental group additional assignments to do on top of those of the control group are excluded. (It is not a very interesting result that additional tasks lead to more learning.)
— Only studies in which researchers or teachers assessed learning outcomes are included. This rules out opinion polls, for instance.
— We only included studies in which an intervention was followed by a distinct assessment phase (e.g., an exam) that was the same for each group. (For instance, we are not interested here in whether students using a program visualization could answer questions about the program being visualized better than others who only saw the code. On the other hand, we would be interested in whether the students had learned to decipher a different program better as a result of their experiences with the visualization.)

Most of the experimental studies listed in Table VI involve comparisons between low levels of engagement, and usually pit controlled viewing or responding against not using a visualization at all. The studies have revolved largely around example-based learning and given content. Most of the existing systems have not been experimentally evaluated at all. The results of the evaluations to date largely support the use of visualization, but it is difficult to draw further conclusions. All in all, we must conclude the CER literature on generic PV systems does not yet tell us very much about what the relative effectiveness of levels of engagement beyond controlled viewing is for CS1 students.

6.4. Opportunities for Research

We list here a few avenues for future work within program visualization for CS1. Some of these paths have been explored more thoroughly in other contexts (e.g., other kinds of educational software, other forms of visualization), others less; in any case, we believe that in order to credibly transfer results to the context we are interested in, studies that involve genuine CS1 courses, learning objectives, and students are needed.

As noted above, learner engagement with visualizations is an area in which much remains to be discovered. The 2DET taxonomy that we presented in this article is one possible basis for generating and testing hypotheses on this subject. We recommend that researchers take explicitly into account also the second dimension of the 2DET, learners' engagement with the content being visualized, which has often been implicit or ignored in the CER literature to date. Future studies should also consider the transferability of what students learn from a visualization. Does the knowledge transfer to answering questions related to the visualization, to program-tracing tasks, or program-writing skills? Does the engagement level play a role in how transferable the knowledge learned is?

Direct engagement constitutes only one of the things that impacts the effectiveness of educational software visualization, and there may be trade-offs in adopting certain modes of engagement. One example of this was discussed by Sorva [2012], who noted that the authenticity of learning tasks is stressed as important by various forms of constructivist learning theory, and continued with the observation that "when professional programmers use program visualization tools to visualize program dynamics, it is typically on the controlled viewing level of direct engagement, most commonly in a visual debugger. ... Visual program simulation involves a trade-off: it sacrifices the authenticity of controlled viewing in order to provide a higher degree of direct engagement [the applying of given content] and encourage learners to make use of the visualization" [Sorva 2012, p. 224]. As another example, the highest levels of direct engagement—creating, especially—bring certain challenges in the context of PV for CS1, such as the time investment required of the students and the difficulty of creating visualizations when one's knowledge of fundamental concepts is fragile [ibid., p. 221]. Such trade-offs merit further study.

There is, within the educational SV community, increasing recognition that any potentially useful capabilities of a visualization system matter less than what the learners make of the system for themselves. Studies of CS1 students' tool usage patterns and their conceptions of PV tools have shown that many students do not perceive the potential benefits of PV and do not use PV tools as their teachers intended [e.g., Isohanni and Knobelsdorf 2010; Laakso et al. 2009; Sorva 2012]. This is something that system creators can affect, as can teachers. Those who create visualization systems must also remember that not nearly all the ways of promoting engagement involve software features; higher engagement may also be brought about by improved visualization-based pedagogies. The integration of program visualization with the rest of the teaching and learning environment has surfaced as an important theme in computing education research [see, e.g., Ben-Bassat Levy and Ben-Ari 2007; Sorva 2012]. Evaluative studies on interventions in which PV is strongly and longitudinally integrated into a CS1 course are scarce, however. Work could also be done in order to produce learning materials in which program visualization interfaces naturally with texts and other content so that using visualizations becomes a natural part of the learner's "reading process" [cf. Shaffer et al. 2011]. Peer learners are a key component of many teaching and learning environments; however, the social aspects of learning from program visualizations have seen little study.

We would like to see more studies in which the specific content that is being learned about (e.g., parameters, references, OOP fundamentals, or loops) is taken into consideration when evaluating generic PV systems. A generic PV system, or a particular mode of user interaction within a system, is not going to be equally effective for all objects of learning. Knowing how well a system or mode of interaction works for a specific goal would help both in choosing the right tools for teaching and in improving on the existing systems. A related path for tool development is to design systems to explicitly address the specific misconceptions that CS1 students are known to have about fundamental concepts; this is currently very rare in the generic PV systems that we have reviewed [see Sorva and Sirkiä 2011 for an example]. Our work in this article could also be extended by reviewing specialized PV systems that teach about particular topics.

The effects of cognitive load [in the sense of working memory load; Paas et al. 2003; Plass et al. 2010] and multimedia instruction [Mayer 2005, 2009] on learning have been studied in many contexts. Although the associated theories have a lot to say about the design of visual learning environments, they are not exerting much explicit influence on current work on educational PV [with exceptions; Moons and De Backer 2013]. For instance, careful consideration of the "split-attention effect" (according to which learning is hindered, for example, by having to pay attention to a visualization and a spatially separate textual explanation at the same time) could contribute to the design of better PV systems. Another limitation of much work to date is the almost nonexistent use of audio, despite the fact that using the auditory and visual channels simultaneously is one of the few known ways of alleviating the problem that human working memory is extremely limited in capacity. Moreover, cognitive load research provides one of the theoretical frameworks that could be used as a basis for creating PV systems that adapt to their users and in particular to the users' prior, and growing, knowledge.

A contribution to the field would also be the construction of a PV system that incorporates many of the good ideas that exist as fragments in the various prototype systems described in the literature. Building a PV system for CS1 that sees genuinely widespread adoption may require not only integration with other course materials but also a departure from the prevalent "I made a prototype for my thesis" mode of system development, perhaps in the form of commercial product development or a lively open-source project.

7. CONCLUDING REMARKS

The primary contribution of this article is to overview the existing literature on program visualization systems whose purpose is to help beginners learn about the execution-time dynamics of computer programs. Our review provides a description of the systems and summarizes the evaluative studies that have been reported in the literature. The review shows that program visualization systems for beginners are often short-lived research prototypes that support user-controlled viewing of program animations; a recent trend is to support more engaging modes of user interaction. Evaluations of the systems that we reviewed have tended to suggest a positive impact on learning introductory programming. Within the context of our survey, we have furthermore revisited the topic of learner engagement, observed that research to date is insufficient for drawing nuanced conclusions about the topic, and suggested a refined framework that could be used to structure future research. Future work there is a lot of: while many systems have been built, and many studies carried out, much remains unstudied about the complex interactions between program visualization tools, learners, learning environments, forms of engagement, and particular learning objectives.

REFERENCES

Ahoniemi, T. and Lahtinen, E. 2007. Visualizations in preparing for programming exercise sessions. *Elec. Notes Theoret. Comp. Sci. 178*, 137–144.

Algoviz. n.d. Algoviz.org: The algorithm visualization portal. http://www.algoviz.org/.

Allen, E., Cartwright, R., and Stoler, B. 2002. DrJava: A lightweight pedagogic environment for Java. *SIGCSE Bull. 34*, 1, 137–141.

Alsaggaf, W., Hamilton, M., Harland, J., D'Souza, D., and Laakso, M.-J. 2012. The use of laptop computers in programming lectures. In *Proceedings of the 23rd Australasian Conference on Information Systems (ACIS'12)*. 1–11.

Anderson, L. W., Krathwohl, D. R., Airasian, P. W., Cruikshank, K. A., Mayer, R. E., Pintrich, P. R., Raths, J., and Wittrock, M. C. 2001. *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives.* Longman.

Andrianoff, S. K. and Levine, D. B. 2002. Role playing in an object-oriented world. *SIGCSE Bull. 34,* 1, 121–125.

Atwood, J. 2008. Everything I needed to know about programming I learned from BASIC (blog post). http://www.codinghorror.com/blog/2008/04/everything-i-needed-to-know-about-programming-i-learned-from-basic.html.

Auguston, M. and Reinfelds, J. 1994. A visual Miranda machine. In *Proceedings of the Software Education Conference (SEC'94)*. 198–203.

Bares, W. H., Zettlemoyer, L. S., and Lester, J. C. 1998. Habitable 3D learning environments for situated learning. In *Proceedings of the 4th International Conference on Intelligent Tutoring Systems (ITS'98)*. 76–85.

Beck, K. and Cunningham, W. 1989. A laboratory for teaching object oriented thinking. *SIGPLAN Not. 24*, 10, 1–6.

Bednarik, R., Myller, N., Sutinen, E., and Tukiainen, M. 2006. Analyzing individual differences in program comprehension. *Tech. Instruc. Cogn. Learn. 3*, 3, 205–232.

Ben-Ari, M. 2001. Constructivism in computer science education. *J. Comp. Math. Sci. Teach. 20*, 1, 45–73.

Ben-Ari, M., Bednarik, R., Ben-Bassat Levy, R., Ebel, G., Moreno, A., Myller, N., and Sutinen, E. 2011. A decade of research and development on program animation: The Jeliot experience. *J. Vis. Lang. Comput. 22*, 375–384.

Ben-Bassat Levy, R. and Ben-Ari, M. 2007. We work so hard and they don't use it: Acceptance of software tools by teachers. *SIGCSE Bull. 39,* 3, 246–250.

Ben-Bassat Levy, R., Ben-Ari, M., and Uronen, P. A. 2003. The Jeliot 2000 program animation system. *Comp. Educ. 40,* 1, 1–15.

Bennedsen, J. and Schulte, C. 2010. BlueJ visual debugger for learning the execution of object-oriented programs? *ACM Trans. Comput. Educ. 10,* 2, 1–22.

Biermann, A. W., Fahmy, A. F., Guinn, C., Pennock, D., Ramm, D., and Wu, P. 1994. Teaching a hierarchical model of computation with animation software in the first course. *SIGCSE Bull. 26,* 1, 295–299.

Birch, M. R., Boroni, C. M., Goosey, F. W., Patton, S. D., Poole, D. K., Pratt, C. M., and Ross, R. J. 1995. DYNALAB: A dynamic computer science laboratory infrastructure featuring program animation. *SIGCSE Bull. 27,* 1, 29–33.

Bloom, B. S. 1956. *Taxonomy of Educational Objectives, Handbook 1: Cognitive Domain*. Addison Wesley.

Booth, S. 1992. Learning to program: A phenomenographic perspective. Doctoral dissertation, University of Gothenburg.

Boroni, C. M., Eneboe, T. J., Goosey, F. W., Ross, J. A., and Ross, R. J. 1996. Dancing with DynaLab: Endearing the science of computing to students. *SIGCSE Bull. 28,* 1, 135–139.

Brito, S., Silva, A. S., Tavares, O., Favero, E. L., and Francês, C. R. L. 2011. Computer supported collaborative learning for helping novice students acquire self-regulated problem-solving skills in computer programming. In *Proceedings of the 7th International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS'11)*. 65–73.

Brown, M. H. 1988. Exploring algorithms using Balsa-II. *Comp. 21,* 5, 14–36.

Bruce-Lockhart, M. P., Crescenzi, P., and Norvell, T. S. 2009. Integrating test generation functionality into the teaching machine environment. In *Electronic Notes in Theoretical Computer Science*, vol. 224, 115–124.

Bruce-Lockhart, M. P. and Norvell, T. S. 2000. Lifting the hood of the computer: Program animation with the teaching machine. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE'00)*. 831–835.

Bruce-Lockhart, M. P. and Norvell, T. S. 2007. Developing mental models of computer programming interactively via the Web. In *Proceedings of the 37th Annual Frontiers in Education Conference (FIE'07)*.

Bruce-Lockhart, M. P., Norvell, T. S., and Cotronis, Y. 2007. Program and algorithm visualization in engineering and physics. In *Electronic Notes in Theoretical Computer Science*, vol. 178, 111–119.

Brusilovsky, P. and Loboda, T. D. 2006. WADEIn II: A case for adaptive explanatory visualization. *SIGCSE Bull. 38,* 3, 48–52.

Brusilovsky, P. L. 1992. Intelligent tutor, environment and manual for introductory programming. *Educ. Train. Tech. Int. 29,* 1, 26–34.

Byckling, P. and Sajaniemi, J. 2005. Using roles of variables in teaching: Effects on program construction. In *Proceedings of the 17th Workshop of the Psychology of Programming Interest Group (PPIG'05)*. 278–292.

Carlisle, M. C. 2009. Raptor: A visual programming environment for teaching object-oriented programming. *J. Comput. Sci. Coll. 24,* 4, 275–281.

Clancy, M. 2004. Misconceptions and attitudes that interfere with learning to program. In *Computer Science Education Research*, S. Fincher and M. Petre Eds., Routledge, 85–100.

Clark, R. E. 1982. Antagonism between achievement and enjoyment in ATI studies. *Educat. Psychol. 17,* 2, 92–101.

Corritore, C. L. and Wiedenbeck, S. 1991. What do novices learn during program comprehension? *Int. J. Hum.-Comput. Inter. 3,* 2, 199–222.

Cross, II, J. H., Barowski, L. A., Hendrix, T. D., and Teate, J. C. 1996. Control structure diagrams for Ada 95. In *Proceedings of TRI-Ada: Disciplined Software Development (ADA'96)*. 143–147.

Cross, II, J. H., Hendrix, T. D., and Barowski, L. A. 2002. Using the debugger as an integral part of teaching CS1. In *Proceedings of the 32nd Annual Frontiers in Education Conference (FIE'02)*.

Cross, II, J. H., Hendrix, T. D., and Barowski, L. A. 2011. Combining dynamic program viewing and testing in early computing courses. In *Proceedings of the 35th Annual IEEE International Computer Software and Applications Conference (COMPSAC'11)*. 184–192.

Cross, II, J. H., Barowski, L. A., Hendrix, D., Umphress, D., and Jain, J. n.d. jGRASP - An integrated development environment with visualizations for improving software comprehensibility (website). http://www.jgrasp.org/.

Cypher, A. 1993. *Watch What I Do: Programming by Demonstration*. MIT Press.

Deng, J. 2003. Programming by demonstration environment for 1st year students. Master's thesis, School of Mathematics, Statistics and Computer Science, Victoria University of Wellington.

Dönmez, O. and İnceoğlu, M. M. 2008. A Web-based tool for novice programmers: Interaction in use. In *Proceedings of the International Conference on Computational Science and its Applications (ICCSA'08)*.

du Boulay, B. 1986. Some difficulties of learning to program. *J. Educ. Comput. Res. 2,* 1, 57–73.

du Boulay, B., O'Shea, T., and Monk, J. 1981. The black box inside the glass box: Presenting computing concepts to novices. *Int. J. Man-Mach. Stud. 14*, 237–249.

Ebel, G. and Ben-Ari, M. 2006. Affective effects of program visualization. In *Proceedings of the 2nd International Workshop on Computing Education Research (ICER'06)*. 1–5.

Eckerdal, A. and Thuné, M. 2005. Novice Java programmers' conceptions of "object" and "class", and variation theory. *SIGCSE Bull. 37,* 3, 89–93.

Eskola, J. and Tarhio, J. 2002. On visualization of recursion with Excel. In *Proceedings of the 2nd Program Visualization Workshop (PVW'02)*. 45–51.

Esteves, M. and Mendes, A. J. 2003. OOP-Anim, a system to support learning of basic object oriented programming concepts. In *Proceedings of the 4th International Conference on Computer Systems and Technologies: e-Learning (CompSysTech'03)*. 573–579.

Esteves, M. and Mendes, A. J. 2004. A simulation tool to help learning of object oriented programming basics. In *Proceedings of the 34th Annual Frontiers in Education Conference (FIE'04)*.

Etheredge, J. 2004. CMeRun: Program logic debugging courseware for CS1/CS2 Students. *SIGCSE Bull. 36,* 1, 22–25.

Fernández, A., Rossi, G., Morelli, P., Garcia Mari, L., Miranda, S., and Suarez, V. 1998. A learning environment to improve object-oriented thinking. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98)*.

Fincher, S., E. A. n.d. Share project: Sharing & representing teaching practice (website). http://www.sharingpractice.ac.uk/homepage.html.

Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., and Felleisen, M. 2002. DrScheme: A programming environment for scheme. *J. Func. Program. 12,* 2, 159–182.

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., and Zander, C. 2008. Debugging: Finding, fixing and flailing, a multi-institutional study of novice debuggers. *Comp. Sci. Educ. 18,* 2, 93–116.

Gajraj, R. R., Williams, M., Bernard, M., and Singh, L. 2011. Transforming source code examples into programming tutorials. In *Proceedings of the 6th International Multi-Conference on Computing in the Global Information Technology (ICCGI'11)*. 160–164.

Gallego-Carrillo, M., Gortázar-Bellas, F., and Velázquez-Iturbide, J. Á. 2004. JavaMod: An integrated Java model for Java software visualization. In *Proceedings of the 3rd Program Visualization Workshop (PVW'04)*. 102–109.

George, C. E. 2000a. EROSI - Visualizing recursion and discovering new errors. *SIGCSE Bull. 32,* 1, 305–309.

George, C. E. 2000b. Evaluating a pedagogic innovation: Execution models & program construction ability. In *Proceedings of the 1st Annual Conference of the LTSN Centre for Information and Computer Sciences (LTSN'00)*. 98–103.

George, C. E. 2000c. Experiences with novices: The importance of graphical representations in supporting mental models. In *Proceedings of the 12th Workshop of the Psychology of Programming Interest Group (PPIG'00)*. 33–44.

George, C. E. 2002. Using visualization to aid program construction tasks. *SIGCSE Bull. 34,* 1, 191–195.

Gestwicki, P. and Jayaraman, B. 2005. Methodology and architecture of JIVE. In *Proceedings of the ACM Symposium on Software Visualization (SoftVis'05)*. 95–104.

Gilligan, D. 1998. An exploration of programming by demonstration in the domain of novice programming. Master's thesis, School of Mathematics, Statistics and Computer Science, Victoria University of Wellington.

Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M. C., and Zilles, C. 2008. Identifying important and difficult concepts in introductory computing courses using a Delphi process. *SIGCSE Bull. 40,* 1, 256–260.

Gómez-Martín, P. P., Gómez-Martín, M. A., Díaz-Agudo, B., and González-Calero, P. A. 2005. Opportunities for CBR in learning by doing. In *Proceedings of the 6th International Conference on Case-Based Reasoning (ICCBR'05)*. 267–281.

Gómez-Martín, M. A., Gómez-Martín, P. P., and González-Calero, P. A. 2006. Dynamic binding is the name of the game. In *Proceedings of the Conference on Entertainment Computing (ICEC'06)*. 229–232.

Gondow, K., Fukuyasu, N., and Arahori, Y. 2010. MieruCompiler: Integrated visualization tool with “horizontal slicing” for educational compilers. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE'10)*. 7–11.

Gračanin, D., Matković, K., and Eltoweissy, M. 2005. Software visualization. *Innov. Syst. Softw. Eng. 1,* 2, 221–230.

Gries, D. 2008. A principled approach to teaching OO first. *SIGCSE Bull. 40,* 1, 31–35.

Gries, P. and Gries, D. 2002. Frames and folders: A teachable memory model for Java. *J. Comput. Sci. Coll. 17,* 6, 182–196.

Gries, P., Mnih, V., Taylor, J., Wilson, G., and Zamparo, L. 2005. Memview: A pedagogically-motivated visual debugger. In *Proceedings of the 35th Annual Frontiers in Education Conference (FIE'05)*. 11–16.

Guo, P. J. 2013. Online Python tutor: Embeddable web-based program visualization for CS education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE'13)*.

Haajanen, J., Pesonius, M., Sutinen, E., Tarhio, J., Teräsvirta, T., and Vanninen, P. 1997. Animation of user algorithms on the Web. In *Proceedings of Symposium on Visual Languages (VL'97)*. 360–367.

Hattie, J. and Purdie, N. 1998. The SOLO model: Addressing fundamental measurement issues. In *Teaching and Learning in Higher Education*, B. Dart and G. Boulton-Lewis Eds., Australian Council for Educational Research, 145–176.

Hauswirth, M., Jazayeri, M., and Winzer, A. 1998. A Java-based environment for teaching programming language concepts. In *Proceedings of the 28th Annual Frontiers in Education Conference (FIE'98)*. 296–300.

Helminen, J. 2009. Jype - An education-oriented integrated program visualization, visual debugging, and programming exercise tool for Python. Master's thesis, Department of Computer Science and Engineering, Helsinki University of Technology.

Helminen, J. and Malmi, L. 2010. Jype - A program visualization and programming exercise tool for Python. In *Proceedings of the 5th International Symposium on Software visualization (SOFTVIS'10)*. 153–162.

Hertz, M. and Jump, M. 2013. Trace-based teaching in early programming courses. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE'13)*. 561–566.

Holliday, M. A. and Luginbuhl, D. 2004. CS1 assessment using memory diagrams. *SIGCSE Bull. 36,* 1, 200–204.

Huizing, C., Kuiper, R., Luijten, C., and Vandalon, V. 2012. Visualization of object-oriented (Java) programs. In *Proceedings of the 4th International Conference on Computer Supported Education (CSEDU'12)*. 65–72.

Hundhausen, C. D., Douglas, S. A., and Stasko, J. T. 2002. A meta-study of algorithm visualization effectiveness. *J. Vis. Lang. Comput. 13,* 3, 259–290.

Isoda, S., Shimomura, T., and Ono, Y. 1987. VIPS: A visual debugger. *IEEE Softw. 4,* 3, 8–19.

Isohanni, E. and Knobelsdorf, M. 2010. Behind the curtain: Students' use of VIP after class. In *Proceedings of the 6th International Workshop on Computing Education Research (ICER'10)*. 87–96.

Isohanni, E. and Knobelsdorf, M. 2011. Students' long-term engagement with the visualization tool VIP. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research (KoliCalling'11)*. 33–38.

Jiménez-Díaz, G., Gómez-Albarrán, M., Gómez-Martín, M. A., and González-Calero, P. A. 2005. Software behavior understanding supported by dynamic visualization and role-play. *SIGCSE Bull. 37,* 3, 54–58.

Jiménez-Díaz, G., Gómez-Albarrán, M., and González-Calero, P. A. 2008. Role-Play virtual environments: Recreational learning of software design. In *Proceedings of the 3rd European Conference on Technology Enhanced Learning: Times of Convergence: Technologies Across Learning Contexts (EC-TEL'08)*. 27–32.

Jiménez-Díaz, G., González-Calero, P. A., and Gómez-Albarrán, M. 2011. Role-play virtual worlds for teaching object-oriented design: The ViRPlay development experience. *Softw. Prac. Exp. 42,* 2, 235–253.

Jiménez-Peris, R., Pareja-Flores, C., Patiño-Martínez, M., and Velázquez-Iturbide, J. Á. 1997. The locker metaphor to teach dynamic memory. *SIGCSE Bull. 29,* 1, 169–173.

Jiménez-Peris, R., Patiño-Martínez, M., and Pacios-Martínez, J. 1999. VisMod: A beginner-friendly programming environment. In *Proceedings of the ACM Symposium on Applied Computing (SAC'99)*. 115–120.

Jones, A. 1992. Conceptual models of programming environments: How learners use the glass box. *Instruct. Sci. 21,* 6, 473–500.

Kaila, E. 2008. A survey of Finnish university teachers on the teaching of programming and tool adoption, in Finnish. http://www.cs.hut.fi/Research/COMPSER/Verkostohanke/raportti.pdf.

Kaila, E., Rajala, T., Laakso, M.-J., and Salakoski, T. 2008. Automatic assessment of program visualization exercises. In *Proceedings of the 8th Koli Calling International Conference on Computing Education Research (KoliCalling'08)*. 105–108.

Kaila, E., Laakso, M.-J., Rajala, T., and Salakoski, T. 2009a. Evaluation of learner engagement in program visualization. In *Proceedings of the12th International Conference on Computers and Advanced Technology in Education (IASTED'09)*.

Kaila, E., Rajala, T., Laakso, M.-J., and Salakoski, T. 2009b. Effects, experiences and feedback from studies of a program visualization tool. *Inform. Educ. 8,* 1, 17–34.

Kaila, E., Rajala, T., Laakso, M.-J., and Salakoski, T. 2010. Effects of course-long use of a program visualization tool. In *Proceedings of the 12th Australasian Conference on Computing Education (ACE'10)*. 97–106.

Kannusmäki, O., Moreno, A., Myller, N., and Sutinen, E. 2004. What a novice wants: Students using program visualization in distance programming course. In *Proceedings of the 3rd Program Visualization Workshop (PVW'04)*. 126–133.

Karavirta, V. 2007. Integrating algorithm visualization systems. In *Electronic Notes in Theoretical Computer Science*, vol. 178, 79–87.

Kasmarik, K. and Thurbon, J. 2003. Experimental evaluation of a program visualization tool for use in computer science education. In *Proceedings of the Asia-Pacific Symposium on Information Visualisation (APVis'03)*. 111–116.

Kelleher, C. and Pausch, R. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv. 37,* 2, 83–137.

Kirby, S., Toland, B., and Deegan, C. 2010. Program visualization tool for teaching programming in C. In *Proceedings of the International Conference on Education, Training and Informatics (ICETI'10)*.

Kölling, M. 2008. Using BlueJ to introduce programming. In *Reflections on the Teaching of Programming: Methods and Implementations*, J. Bennedsen, M. E. Caspersen, and M. Kolling Eds., Springer, 98–115.

Kollmansberger, S. 2010. Helping students build a mental model of computation. In *Proceedings of the 15th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'10)*. 128–131.

Korhonen, A. n.d. Ohjelmoinnin perusopetuksen verkosto. http://www.cs.hut.fi/Research/COMPSER/Verkostohanke/index.shtml A website for networking amongst Finnish teachers of introductory programming. Finnish Virtual University.

Korhonen, A., Helminen, J., Karavirta, V., and Seppälä, O. 2009a. TRAKLA2. In *Proceedings of the 9th Koli Calling International Conference on Computing Education Research (KoliCalling'09)*. 43–46.

Korhonen, A., Laakso, M.-J., and Myller, N. 2009b. How does algorithm visualization affect collaboration? Video analysis of engagement and discussions. In *Proceedings of the 5th International Conference on Web Information Systems and Technologies (WEBIST'09)*. 479–488.

Korhonen, A., Malmi, L., Silvasti, P., Karavirta, V., Lönnberg, J., Nikander, J., Stålnacke, K., and Ihantola, P. 2004. Matrix - A framework for interactive software visualization. Research rep. TKO-B 154/04, Department of Computer Science and Engineering, Helsinki University of Technology.

Korsh, J. F. and Sangwan, R. 1998. Animating programs and students in the laboratory. In *Proceedings of the 28th Annual Frontiers in Education Conference (FIE'98)*. 1139–1144.

Kumar, A. N. 2005. Results from the evaluation of the effectiveness of an online tutor on expression evaluation. *SIGCSE Bull. 37,* 1, 216–220.

Kumar, A. N. 2009. Data space animation for learning the semantics of C++ pointers. *SIGCSE Bull. 41,* 1, 499–503.

Laakso, M.-J., Myller, N., and Korhonen, A. 2009. Comparing learning performance of students using algorithm visualizations collaboratively on different engagement levels. *J. Educ. Tech. Soc. 12,* 2, 267–282.

Laakso, M.-J., Rajala, T., Kaila, E., and Salakoski, T. 2008. The impact of prior experience in using a visualization tool on learning to program. In *Proceedings of Cognition and Exploratory Learning in Digital Age (CELDA'08)*. 129–136.

LaFollette, P., Korsh, J., and Sangwan, R. 2000. A visual interface for effortless animation of C/C++ programs. *J. Vis. Lang. Comput. 11,* 1, 27–48.

Lahtinen, E. and Ahoniemi, T. 2005. Visualizations to support programming on different levels of cognitive development. In *Proceedings of the 5th Koli Calling Conference on Computer Science Education (KoliCalling'05)*. 87—94.

Lahtinen, E. and Ahoniemi, T. 2007. Annotations for defining interactive instructions to interpreter based program visualization tools. In *Electronic Notes in Theoretical Computer Science*, vol. 178, 121–128.

Lahtinen, E., Ala-Mutka, K., and Järvinen, H.-M. 2005. A study of the difficulties of novice programmers. *SIGCSE Bull. 37,* 3, 14–18.

Lahtinen, E., Ahoniemi, T., and Salo, A. 2007a. Effectiveness of integrating program visualizations to a programming course. In *Proceedings of the 7th Baltic Sea Conference on Computing Education Research (KoliCalling'07)*. 195–198.

Lahtinen, E., Järvinen, H.-M., and Melakoski-Vistbacka, S. 2007b. Targeting program visualizations. *SIGCSE Bull. 39,* 3, 256–260.

Larochelle, M., Bednarz, N., and Garrison, J., Eds. 1998. *Constructivism and Education*. Cambridge University Press.

Lattu, M., Meisalo, V., and Tarhio, J. 2003. A visualization tool as a demonstration aid. *Comp. Educ. 41,* 2, 133–148.

Lattu, M., Tarhio, J., and Meisalo, V. 2000. How a visualization tool can be used - Evaluating a tool in a research & development project. In *Proceedings of the 18th Workshop of the Psychology of Programming Interest Group (PPIG'00)*. 19–32.

Lauer, T. 2006. Learner interaction with algorithm visualizations: Viewing vs. changing vs. constructing. *SIGCSE Bull. 38,* 3, 202–206.

Lessa, D., Czyz, J. K., Gestwicki, P. V., and Jayaraman, B. n.d. JIVE: Java interactive visualization environment (website). http://www.cse.buffalo.edu/jive/.

Lieberman, H. 1984. Steps toward better debugging tools for LISP. In *Proceedings of the ACM Symposium on LISP and Functional Programming (LFP'84)*. 247–255.

Lieberman, H. and Fry, C. 1997. ZStep 95: A reversible, animated source code stepper. In *Software Visualization: Programming as a Multimedia Experience*, J. Stasko, J. Domingue, B. Price, and M. Brown Eds., MIT Press, 277–292.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., and Thomas, L. 2004. A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bull. 36,* 4, 119–150.

Luijten, C. 2009. Interactive visualization of the execution of object-oriented programs. Master's thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology.

Ma, L. 2007. Investigating and improving novice programmers' mental models of programming concepts. Doctoral dissertation, Department of Computer & Information Sciences, University of Strathclyde.

Ma, L., Ferguson, J. D., Roper, M., Ross, I., and Wood, M. 2009. Improving the mental models held by novice programmers using cognitive conflict and Jeliot visualizations. *SIGCSE Bull. 41,* 3, 166–170.

Ma, L., Ferguson, J., Roper, M., and Wood, M. 2011. Investigating and improving the models of programming concepts held by novice programmers.. *Comp. Sci. Educ. 21,* 1, 57–80.

Maletic, J. I., Marcus, A., and Collard, M. L. 2002. A task oriented view of software visualization. In *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'02)*. 32–40.

Mann, L. M., Linn, M. C., and Clancy, M. 1994. Can tracing tools contribute to programming proficiency? The LISP evaluation modeler. *Inter. Learn. Envir. 4,* 1, 96–113.

Maravić Čisar, S., Pinter, R., Radosav, D., and Čisar, P. 2010. Software visualization: The educational tool to enhance student learning. In *Proceedings of the 33rd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO'10)*. 990–994.

Maravić Čisar, S., Radosav, D., Pinter, R., and Čisar, P. 2011. Effectiveness of program visualization in learning Java: A case study with Jeliot 3. *Int. J. Comp. Comm. Control 6,* 4, 669–682.

Mayer, R. E. 1975. Different problem-solving competencies established in learning computer programming with and without meaningful models. *J. Educ. Psych. 67,* 6, 725–734.

Mayer, R. E. 1976. Some conditions of meaningful learning for computer programming: advance organizers and subject control of frame order. *J. Educ. Psych. 68*, 143–150.

Mayer, R. E. 1981. The psychology of how novices learn computer programming. *ACM Comp. Surv. 13,* 1, 121–141.

Mayer, R. E., Ed. 2005. *The Cambridge Handbook of Multimedia Learning*. Cambridge University Press.

Mayer, R. E. 2009. *Multimedia Learning* 2nd Ed. Cambridge University Press.

Miller, L. A. 1981. Natural language programming: Styles, strategies, and contrasts. *IBM Syst. J. 20,* 2, 184–215.

Milne, I. and Rowe, G. 2002. Difficulties in learning and teaching programming - Views of students and tutors. *Educ. Inf. Technol. 7,* 1, 55–66.

Milne, I. and Rowe, G. 2004. OGRE: Three-dimensional program visualization for novice programmers. *Educ. Inf. Technol. 9,* 3, 219–237.

Miyadera, Y., Kurasawa, K., Nakamura, S., Yonezawa, N., and Yokoyama, S. 2007. A real-time monitoring system for programming education using a generator of program animation systems. *J. Comp. 2,* 3, 12–20.

Moons, J. and De Backer, C. 2013. The design and pilot evaluation of an interactive learning environment for introductory programming influenced by cognitive load theory and constructivism. *Comp. Educ. 60,* 1, 368–384.

Moreno, A. 2005. The design and implementation of intermediate codes for software visualization. Master's thesis, Department of Computer Science, University of Joensuu.

Moreno, A. and Joy, M. S. 2007. Jeliot 3 in a demanding educational setting. In *Electronic Notes in Theoretical Computer Science*, vol. 178, 51–59.

Moreno, A. and Myller, N. 2003. Producing an educationally effective and usable tool for learning, the case of the Jeliot family. In *Proceedings of the International Conference on Networked e-learning for European Universities (EUROPACE'03)*.

Moreno, A., Myller, N., Sutinen, E., and Ben-Ari, M. 2004. Visualizing programs with Jeliot 3. In *Proceedings of the International Working Conference on Advanced Visual Interfaces (AVI'04)*. 373–376.

Mota, M. P., Brito, S., Moreira, M. P., and Favero, E. L. 2009. Ambiente Integrado à Plataforma Moodle para Apoio ao Desenvolvimento das Habilidades Iniciais de Programação. An environment integrated into the moodle platform for the development of first habits of programming, in Portuguese. In *Anais do XX Simpósio Brasileiro de Informatica na Educacaon.*

Mselle, L. J. 2011. Enhancing comprehension by using random access memory (RAM) diagrams in teaching programming: Class experiment. In *Proceedings of the 23rd Annual Workshop of the Psychology of Programming Interest Group (PPIG'11)*.

Murphy, C., Kim, E., Kaiser, G., and Cannon, A. 2008. Backstop: A tool for debugging runtime errors. *SIGCSE Bull. 40,* 1, 173–177.

Myers, B. A. 1990. Taxonomies of visual programming and program visualization. *J. Vis. Lang. Comput. 1*, 97–123.

Myers, B. A., Chandhok, R., and Sareen, A. 1988. Automatic data visualization for novice Pascal programmers. In *Proceedings of the IEEE Workshop on Visual Languages (WVL'88)*. 192–198.

Myller, N., Bednarik, R., and Moreno, A. 2007a. Integrating dynamic program visualization into BlueJ: The Jeliot 3 extension. In *Proceedings of the 7th IEEE International Conference on Advanced Learning Technologies (ICALT'07)*. 505–506.

Myller, N., Laakso, M., and Korhonen, A. 2007b. Analyzing engagement taxonomy in collaborative algorithm visualization. *SIGCSE Bull. 39,* 3, 251–255.

Myller, N., Bednarik, R., Sutinen, E., and Ben-Ari, M. 2009. Extending the engagement taxonomy: Software visualization and collaborative learning. *ACM Trans. Comput. Educ. 9,* 1, 1–27.

Najjar, L. J. 1998. Principles of educational multimedia user interface design. *Hum. Fact. 40,* 2, 311–323.

Naps, T. L. 2005. JHAVE: Supporting algorithm visualization. *Comp. Graph. Appl. 25,* 5, 49–55.

Naps, T. L. and Stenglein, J. 1996. Tools for visual exploration of scope and parameter passing in a programming languages course. *SIGCSE Bull. 28,* 1, 305–309.

Naps, T. L., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S., and Velázquez-Iturbide, J. Á. 2003. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bull. 35,* 2, 131–152.

Nevalainen, S. and Sajaniemi, J. 2005. Short-term effects of graphical versus textual visualization of variables on program perception. In *Proceedings of the 17th Workshop of the Psychology of Programming Interest Group (PPIG'05)*. 77–91.

Nevalainen, S. and Sajaniemi, J. 2006. An experiment on short-term effects of animated versus static visualization of operations on program perception. In *Proceedings of the 2nd International Workshop on Computing Education Research (ICER'06)*. 7–16.

Nevalainen, S. and Sajaniemi, J. 2008. An experiment on the short-term effects of engagement and representation in program animation. *J. Educ. Comput. Res. 39,* 4, 395–430.

Oechsle, R. and Morth, T. 2007. Peer review of animations developed by students. In *Electronic Notes in Theoretical Computer Science*, vol. 178, 181–186.

Oechsle, R. and Schmitt, T. 2002. JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java Debug Interface (JDI). In *Revised Lectures on Software Visualization, International Seminar*, S. Diehl Ed., Springer, 176–190.

Paas, F., Renkl, A., and Sweller, J., Eds. 2003. Educational psychologist. *Cogn. Load Theory 38*, 1.

Pane, J. F., Ratanamahatana, C. A., and Myers, B. A. 2001. Studying the language and structure in non-programmers' solutions to programming problems. *Int. J. Hum.-Comp. Stud. 54,* 2, 237–264.

Papert, S. 1993. *Mindstorms: Children, Computers, and Powerful Ideas*. Da Capo Press.

Pareja-Flores, C., Urquiza-Fuentes, J., and Velázquez-Iturbide, J. Á. 2007. WinHIPE: An IDE for functional programming based on rewriting and visualization. *SIGPLAN Not. 42,* 3, 14–23.

Pea, R. D. 1986. Language-independent conceptual "bugs" in novice programming. *J. Educ. Comp. Res. 2,* 1, 25–36.

Pears, A. and Rogalli, M. 2011a. mJeliot: A tool for enhanced interactivity in programming instruction. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research (KoliCalling'11)*. 10–15.

Pears, A. and Rogalli, M. 2011b. mJeliot: ICT support for interactive teaching of programming. In *Proceedings of the 41st Annual Frontiers in Education Conference (FIE'11)*.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., and Paterson, J. 2007. A survey of literature on the teaching of introductory programming. *SIGCSE Bull. 39,* 4, 204–223.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., and Simmons, R. 1986. Conditions of learning in novice programmers. *J. Educ. Comp. Res. 2,* 1, 37–55.

Petre, M. 1995. Why looking isn't always seeing: Readership skills and graphical programming. *Comm. ACM 38,* 6, 33–44.

Phillips, D. C., Ed. 2000. *Constructivism in Education: Opinions and Second Opinions on Controversial Issues*. The National Society for the Study of Education.

Plass, J. L., Moreno, R., and Brünken, R., Eds. 2010. *Cognitive Load Theory*. Cambridge University Press.

Price, B. A., Baecker, R. M., and Small, I. S. 1993. A principled taxonomy of software visualization. *J. Vis. Lang. Comput. 4,* 3, 211–266.

Ragonis, N. and Ben-Ari, M. 2005. A long-term investigation of the comprehension of OOP concepts by novices. *Comp. Sci. Educ. 15,* 3, 203–221.

Rajala, T., Laakso, M.-J., Kaila, E., and Salakoski, T. 2007. VILLE - A language-independent program visualization tool. In *Proceedings of the 7th Baltic Sea Conference on Computing Education Research (KoliCalling'07)*. 151–159.

Rajala, T., Laakso, M.-J., Kaila, E., and Salakoski, T. 2008. Effectiveness of program visualization: A case study with the ViLLE tool. *J. Inf. Tech. Educ. Innov. Prac. 7*, 15–32.

Rajala, T., Kaila, E., Laakso, M.-J., and Salakoski, T. 2009. Effects of collaboration in program visualization. In *Proceedings of the Technology Enhanced Learning Conference (TELearn'09)*.

Rajala, T., Kaila, E., and Laakso, M.-J. ViLLE: Collaborative education tool (website). http://ville.cs.utu.fi.

Ramadhan, H. A. 2000. Programming by discovery. *J. Comp. Assist. Learn. 16*, 83–93.

Ramadhan, H. A., Deek, F., and Shilab, K. 2001. Incorporating software visualization in the design of intelligent diagnosis systems for user programming. *Art. Intell. Rev. 16*, 61–84.

Robinett, W. 1979. Basic Programming.

Roman, G.-C. and Cox, K. C. 1993. A taxonomy of program visualization systems. *Comp. 26,* 12, 97–123.

Ross, R. J. 1983. LOPLE: A dynamic library of programming language examples. *SIGCUE Outl. 17,* 4, 27–31.

Ross, R. J. 1991. Experience with the DYNAMOD program animator. *SIGCSE Bull. 23,* 1, 35–42.

Rowe, G. and Thorburn, G. 2000. VINCE - An online tutorial tool for teaching introductory programming. *Brit. J. Educ. Tech. 31,* 4, 359–369.

Sajaniemi, J. n.d. The roles of variables home page. http://cs.joensuu.fi/∼saja/var_roles/.

Sajaniemi, J. and Kuittinen, M. 2003. Program animation based on the roles of variables. In *Proceedings of the ACM Symposium on Software Visualization (SoftVis'03)*. 7–16.

Sajaniemi, J. and Kuittinen, M. 2005. An experiment on using roles of variables in teaching introductory programming. *Comp. Sci. Educ. 15,* 1, 59–82.

Sajaniemi, J., Byckling, P., and Gerdt, P. 2007. Animation metaphors for object-oriented concepts. In *Electronic Notes in Theoretical Computer Science*, vol. 178, 15–22.

Sajaniemi, J., Kuittinen, M., and Tikansalo, T. 2008. A study of the development of students' visualizations of program state during an elementary object-oriented programming course. *J. Educ. Res. Comp. 7,* 4, 1–31.

Scheiter, K., Gerjets, P., and Catrambone, R. 2006. Making the abstract concrete: Visualizing mathematical solution procedures. *Comp. Hum. Behav. 22,* 1, 9–25.

Schulte, C. and Bennedsen, J. 2006. What do teachers teach in introductory programming? In *Proceedings of the 2nd International Workshop on Computing Education Research (ICER'06)*. 17–28.

Scott, A., Watkins, M., and McPhee, D. 2008. Progranimate - A Web enabled algorithmic problem solving application. In *Proceedings of the International Conference on E-Learning, E-Business, Enterprise Information Systems, & E-Government (EEE'08)*. 498–508.

Seppälä, O. 2004. Program state visualization tool for teaching CS1. In *Proceedings of the 3rd Program Visualization Workshop (PVW'04)*. 118–125.

Shaffer, C. A., Naps, T. L., and Fouh, E. 2011. Truly interactive textbooks for computer science education. In *Proceedings of the 6th Program Visualization Workshop (PVW'11)*. 97–106.

Sherry, L. 1995. A model computer simulation as an epistemic game. *SIGCSE Bull. 27,* 2, 59–64.

Shinners-Kennedy, D. 2008. The everydayness of threshold concepts: State as an example from computer science. In *Threshold Concepts within the Disciplines*, R. Land and J. H. F. Meyer Eds., SensePublishers, 119–128.

Simon. 2011. Assignment and sequence: why some students can't recognize a simple swap. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research (KoliCalling'11)*. 16–22.

Sirkiä, T. and Sorva, J. 2012. Exploring programming misconceptions: An analysis of student mistakes in visual program simulation exercises. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research (KoliCalling'12)*. 19–28.

Sivula, K. 2005. A qualitative case study on the use of Jeliot 3. Master's thesis, Department of Computer Science, University of Joensuu.

Sleeman, D., Putnam, R. T., Baxter, J., and Kuspa, L. 1986. Pascal and high school students: A study of errors. *J. Educ. Comp. Res. 2,* 1, 5–23.

Smith, P. A. and Webb, G. I. 1991. Debugging using partial models. In *Proceedings of the 4th Australian Society for Computer in Learning in Tertiary Education Conference (ASCILITE'91)*. 581–590.

Smith, P. A. and Webb, G. I. 1995a. Reinforcing a generic computer model for novice programmers. In *Proceedings of the 7th Australian Society for Computer in Learning in Tertiary Education Conference (ASCILITE'95)*.

Smith, P. A. and Webb, G. I. 1995b. Transparency debugging with explanations for novice programmers. In *Proceedings of the 2nd Workshop on Automated and Algorithmic Debugging (AADEBUG'95)*. 105–118.

Smith, P. A. and Webb, G. I. 2000. The efficacy of a low-level program visualization tool for teaching programming concepts to novice c programmers. *J. Educ. Comp. Res. 22,* 2, 187–215.

Sorva, J. 2010. Reflections on threshold concepts in computer programming and beyond. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (KoliCalling'10)*. 21–30.

Sorva, J. 2012. Visual program simulation in introductory programming education. Doctoral dissertation, Department of Computer Science and Engineering, Aalto University.

Sorva, J. 2013. Notional machines and introductory programming education. *ACM Trans. Comput. Educ. 13*, 4.

Sorva, J. and Sirkiä, T. 2010. UUhistle - A software tool for visual program simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (KoliCalling'10)*. 49–54.

Sorva, J. and Sirkiä, T. 2011. Context-sensitive guidance in the UUhistle program visualization system. In *Proceedings of the 6th Program Visualization Workshop (PVW'11)*. 77–85.

Sorva, J., Lönnberg, J., and Malmi, L. accepted. Students' ways of experiencing visual program simulation. *Comp. Sci. Educ*.

Stasko, J. T. and Patterson, C. 1992. Understanding and characterizing software visualization systems. In *Proceedings of the IEEE Workshop on Visual Languages (VL'92)*. 3–10.

Stützle, T. and Sajaniemi, J. 2005. An empirical evaluation of visual metaphors in the animation of roles of variables. *Inform. Sci. J. 8*, 87–100.

Sundararaman, J. and Back, G. 2008. HDPV: Interactive, faithful, in-vivo runtime state visualization for C/C++ and Java. In *Proceedings of the 4th ACM Symposium on Software Visualization (SoftVis'08)*. 47–56.

Sutinen, E., Tarhio, J., Lahtinen, S.-P., Tuovinen, A.-P., Rautama, E., and Meisalo, V. 1997. Eliot - An algorithm animation environment. Teaching and Learning rep. A-1997-4, Department of Computer Science, University of Helsinki.

Terada, M. 2005. ETV: A program trace player for students. *SIGCSE Bull. 37,* 3, 118–122.

Thomas, L., Ratcliffe, M., and Thomasson, B. 2004. Scaffolding with object diagrams in first year programming classes: Some unexpected results. *SIGCSE Bull. 36,* 1, 250–254.

Thuné, M. and Eckerdal, A. 2010. Students' conceptions of computer programming. Tech. rep. 2010-021, Department of Information Technology, Uppsala University.

Urquiza-Fuentes, J. and Velázquez-Iturbide, J. Á. 2007. An evaluation of the effortless approach to build algorithm animations with WinHIPE. In *Electronic Notes in Theoretical Computer Science*, vol. 178, 3–13.

Urquiza-Fuentes, J. and Velázquez-Iturbide, J. Á. 2009. A survey of successful evaluations of program visualization and algorithm animation systems. *ACM Trans. Comp. Educ. 9,* 2, 1–21.

Urquiza-Fuentes, J. and Velázquez-Iturbide, J. A. 2012. Comparing the effectiveness of different educational uses of program animations. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'12)*. 174–179.

Vagianou, E. 2006. Program working storage: A beginner's model. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research (KoliCalling'06)*. 69–76.

Vainio, V. and Sajaniemi, J. 2007. Factors in novice programmers' poor tracing skills. *SIGCSE Bull. 39,* 3, 236–240.

Velázquez-Iturbide, J. Á., Pérez-Carrasco, A., and Urquiza-Fuentes, J. 2008. SRec: An animation system of recursion for algorithm courses. *SIGCSE Bull. 40,* 3, 225–229.

Virtanen, A. T., Lahtinen, E., and Järvinen, H.-M. 2005. VIP, a visual interpreter for learning introductory programming with C++. In *Proceedings of the 5th Koli Calling Conference on Computer ACM Transactions on Computing Education (KoliCalling'05)*. 125–130.

Wang, P., Bednarik, R., and Moreno, A. 2012. During automatic program animation, explanations after animations have greater impact than before animations. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research (KoliCalling'12)*. 100–108.

Weber, G. and Brusilovsky, P. 2001. ELM-ART: An adaptive versatile system for web-based instruction. *Int. J. Art. Intell. Educ. 12*, 351–384.

Yehezkel, C., Ben-Ari, M., and Dreyfus, T. 2007. The contribution of visualization to learning computer architecture. *Comp. Sci. Educ. 17,* 2, 117–127.

Received May 2012; revised November 2012; accepted May 2013