TIDE: An Educational Live Programming Environment to Compose Graphics with PyTamaro

Joey Bevilacqua
joey.bevilacqua@usi.ch
Software Institute, Università della Svizzera italiana
Lugano, Switzerland

Nathan Coquerel
nathan.coquerel@ens-rennes.fr
Rennes University
Rennes, France

Luca Chiodini
luca.chiodini@usi.ch
Software Institute, Università della Svizzera italiana
Lugano, Switzerland

Igor Moreno Santos
igor.moreno.santos@usi.ch
Software Institute, Università della Svizzera italiana
Lugano, Switzerland

Matthias Hauswirth
matthias.hauswirth@usi.ch
Software Institute, Università della Svizzera italiana
Lugano, Switzerland

Abstract

The PyTamaro approach to introductory programming equates *composing a program* to *composing a graphic*. Using the simple PyTamaro library for Python, beginner programmers compose function calls that produce primitive graphics, such as `rectangle(20, 10, red)`, with function calls that combine graphics into composites, such as `above(_, _)`.

To ease students into programming, several school teachers have been using TamaroCards, a paper-based visual language that can express the subset of Python needed to compose simple graphics with PyTamaro.

This paper introduces TIDE, the TamaroCards IDE, a web-based environment to complement the unplugged, paper-based TamaroCards. Given the constraints imposed by the Python programming language, the PyTamaro library, and the TamaroCards notation, the paper explores the design space for an interactive TamaroCards programming environment, describes a design that satisfies the pedagogical needs, and evaluates the implemented design using the Cognitive Dimensions framework.

CCS Concepts: • **Social and professional topics → Computer science education**.

Keywords: Novice Programmer, Visual Programming, Notional Machine, Graphics Programming, Live Programming, Composition, Expressions, Error Reporting, Evaluation

ACM Reference Format:
Joey Bevilacqua, Nathan Coquerel, Luca Chiodini, Igor Moreno Santos, and Matthias Hauswirth. 2025. TIDE: An Educational Live Programming Environment to Compose Graphics with PyTamaro.

PAINT '25, Singapore, Singapore

ACM ISBN 979-8-4007-2160-1/25/10
https://doi.org/10.1145/3759534.3762683

In *Proceedings of the 4th ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3759534.3762683

1 Introduction

PyTamaro [9] is an educational graphics library for Python. Aimed at beginner programmers, it only requires a minimal set of Python language constructs (literals, constant uses, and function calls) to create primitive graphics and to compose them. Because PyTamaro uses pure functions and immutable graphic values, there is a direct correspondence between the *composition of function calls* and the *composition of graphics*. This supports students in reasoning about the structure of code, and it helps them in learning to trace program execution [10]. This correspondence also supports students in learning to abstract, by identifying identical or similar components of graphics (clones), and by defining constants and functions to represent them. The library, together with an associated web site[1] that provides example programming exercises, has been used for more than three years in multiple Swiss high school informatics courses to teach programming in Python and has been trialed in a year-long introductory programming course in middle school.

TamaroCards (TC)[2] is a visual data-flow language for representing simple PyTamaro programs. It was developed to support teachers in the initial lessons of their PyTamaro-based programming courses. The language primarily consists of *paper cards* representing function calls, constant uses, and literals. Students place those cards on tables and connect them with hand-drawn lines to form complete expressions, as shown in Figure 1. Teachers who adopted TC in their courses generally find the cards helpful. However, some teachers

[1] https://pytamaro.si.usi.ch/
[2] https://pytamaro.si.usi.ch/tamarocards

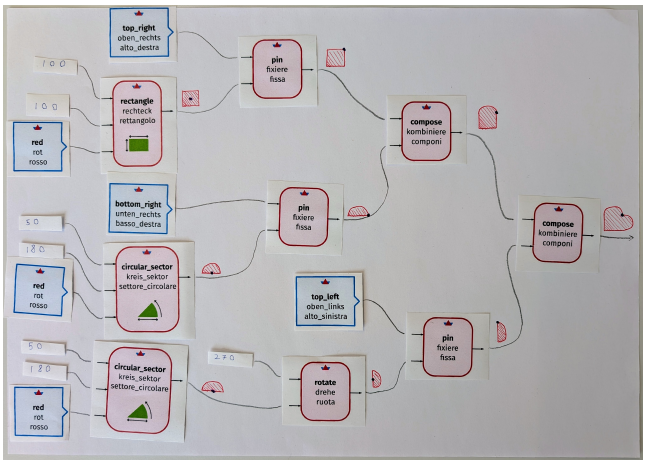

Figure 1. Use of paper TamaroCards to compose a heart. Cards are connected by drawing the links on the background paper. The evaluation of the program can be traced by drawing the graphics produced by each function call.

report that working with paper instead of a computer is perceived as *unauthentic* by some of their students.

In this paper we present the TamaroCards IDE (TIDE), a web-based environment for composing and evaluating TC programs[3]. TIDE enables teachers to start their programming course on the computer, while still enjoying the pedagogical benefits of the TamaroCards. We discuss the design of the environment, which is constrained by the existing TC notation, the PyTamaro library, the underlying general-purpose programming language, Python, and the pedagogical needs and priorities. We demonstrate the feasibility of our idea through a prototypical implementation and use the Cognitive Dimensions [14] framework to evaluate TIDE.

The remainder of this paper is structured as follows: Section 2 discusses related work, Section 3 provides the background on the TC language, Section 4 presents the design space for a TC editing environment, Section 5 describes and motivates TIDE as a specific point in that design space, Section 6 evaluates TIDE using the Cognitive Dimensions framework, Section 7 discusses limitations and future work, and Section 8 concludes.

2 Related Work

TamaroCards (TC) is a small visual language centered around pure functions and immutable data. It primarily focuses on the *composition of expressions*, a prevalent construct of programming languages that is often neglected in programming education [7].

TC supports *Babylonian programming* [24], a form of programming that tightly integrates values and code. Babylonian programming has been proposed as a promising approach to programming education [16]. TC was deliberately designed to allow tracing the evaluation of expressions by drawing intermediate values (mostly graphics) produced by each sub-expression. This juxtaposition of values and code can reduce the *split attention effect* [5].

TC can be seen as a *notional machine*, a "pedagogic device to assist the understanding of some aspect of programs or programming" [11]. Specifically, it is a form of the "expression as tree" notional machine[4] that has been used to teach and assess students' conceptual understanding of expressions and helps to shed light on their structure, typing, and evaluation [1].

The purely functional nature of beginner PyTamaro programs, characterized by the absence of control-flow and mutable state, favors the use of a *data-flow language* [15]. This contrasts with common educational visual programming languages like Scratch [25], that are based on a block paradigm that focuses on imperative structured programming.

Visual data-flow programming systems can be traced back to Sutherland's work on "graphical specifications of computer procedures" [27]. Nowadays, visual data-flow languages are used in a wide range of domains: scientific visualizations and simulations (e.g., Iris Explorer and Simulink, respectively), systems design (e.g., LabVIEW), image and video editing (e.g., Graphite and DaVinci Resolve Fusion, respectively), game development (e.g., Unity Visual Scripting / Bolt), and for education (e.g., teaching embedded systems programming [3]).

The paper-based TC can be seen as a manual, physical form of *live programming*. Live programming systems support "responsive and continuous feedback about how [a developer's] edits affect program execution" [18], which reduces the friction between the writing of the code and its execution, encouraging experimentation [4]. Tanimoto identifies six different levels of "liveness", that may be used to classify systems that support live programming [28]. Of course, with the paper-based TC there is no *automatic* evaluation, but the programming process can still interleave the *manual* placement and connection of cards with the *manual* evaluation of sub-expressions. The goal of TIDE is to automate the evaluation part and to provide feedback even when programs contain errors.

When programming with the paper-based TC, students are free to compose expressions in any order they want. They may build expressions bottom-up, starting with literals and constant uses. Alternatively, they may work top-down, starting with the top-level node of an expression. The bottom-up

[3] A deployed version of TIDE is available at https://pytamaro.si.usi.ch/tide

[4] https://notionalmachines.github.io/nms/ExpressionAsTree-1.html

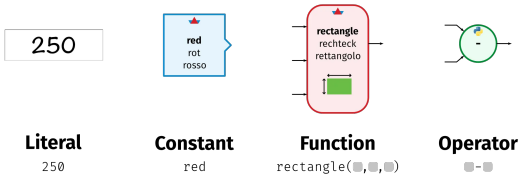

Figure 2. Four kinds of expression cards in a TC program

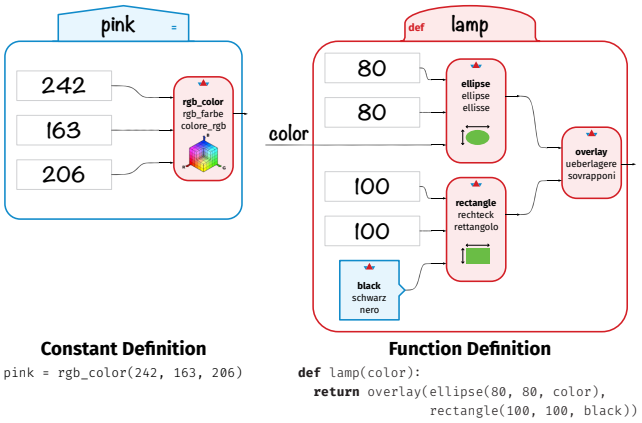

Figure 3. Two kinds of definitions in a TC program

approach leads to the existence of multiple top-level expressions during the construction process. The top-down approach leads to an *expression with holes*, an idea that has been studied in live programming systems like Hazel [21].

3 Background: The Visual Language

TamaroCards (TC) represents the small subset of Python needed in beginner PyTamaro programs. It is a language of *expressions* consisting of literals, constant uses, function calls, and operators (Figure 2). It also supports the *definition* of constants and functions (Figure 3). The language represents a pure and immutable subset of Python, without mutable state and control flow. TC's node-link diagram notation highlights the data-flow through expressions. Data flows from left to right: function and operator nodes have an *outlet port* on their right, and *inlet ports* (parameters) on their left.

TC was designed as a "tangible" language: students create programs by placing paper cards (printed and cut by teachers) on a large sheet of paper and by drawing lines from outlets to inlets to compose expressions. Literal cards are empty pieces of paper onto which students write the literal by hand. Students can use constant and function definition cards ("`pink =`" and "`def lamp`" in Figure 3) to define their own constants and functions.

More information about the language is also provided later in Section 5 when relevant to the discussion of design decisions influenced by properties that the IDE inherits from the TC visual programming language.

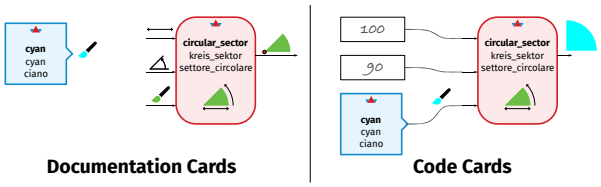

Figure 4. Documentation cards include illustrative annotations for parameters and return values. Code cards do not include those annotations to avoid conflicts with manually drawn evaluation results.

3.1 Code Cards vs. Documentation Cards

Each TC card comes in two variants: the code card and the documentation card (Figure 4). *Code cards* are the cards students use to compose expressions. *Documentation cards* represent the "library documentation". They include visual annotations on each inlet (parameter) and outlet to illustrate the semantics of the card. Each annotation represents an example value (e.g., a color, a width, a graphic), so that the set of annotations provide an illustrative usage example of the card. An annotated card is a form of purely visual documentation: it helps students to determine the meaning of a card simply by looking at the visual annotations. Code cards elide these annotations, which leaves space for students to trace the evaluation of an expression by drawing evaluation results on the right side of cards ("Code Cards" in Figure 4).

3.2 Classroom Use

The TC language was developed for a teacher training workshop on teaching Python programming with PyTamaro. Teachers participating in the workshop subsequently started using TC in their own classrooms. Figure 5 shows an exercise from a middle school programming course[5]. Students solve such exercises by placing paper cards on their table, connecting them with hand-drawn lines, and sometimes annotating each card with the value it produces (Figure 1).

Because of their unplugged paper-based nature, students and teachers can use TC in unintended ways. For example, teachers may use documentation cards instead of code cards to compose expressions, or students may just juxtapose cards without drawing any links between ports.

3.3 From Physical to Virtual

TC are physical manipulatives that students can arrange on a table. Fyfe and Nathan hypothesized that "a concreteness fading progression that includes a physical manipulative in the first stage will be more effective than one that does not" [13]. TC can be seen as a first stage in a concreteness fading progression, where the next stage is working with (more abstract) Python code. Fyfe and Nathan's hypothesis could be specialized to mean that starting programming by

[5] https://luce.si.usi.ch/composing-python/

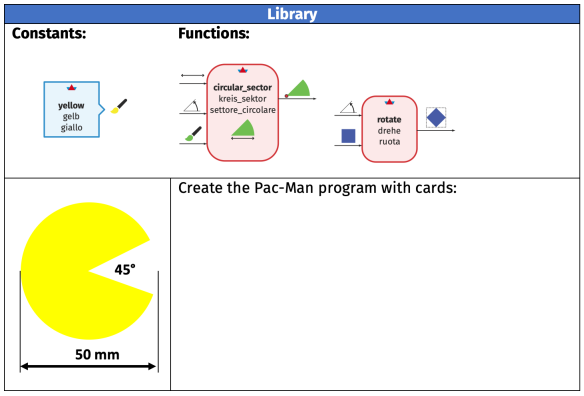

Figure 5. Middle school exercise: Use the three cards from the library to program the shown Pac-Man graphic.

using *physical*, paper-based TC would be more beneficial than starting using *virtual* TC. However, a recent study by Trory et al. found little effect of the modality on learning gains or attitude [29]. Thus, using a virtual environment to work with TC may offer interesting benefits without significant drawbacks.

4 Design Space for an Environment

There are two key facets to designing a visual programming environment: the design of the *visual language*, and the design of the *interactive environment*. Given that the language (TC) already exists in a paper-based form, and has been used in educational practice in schools, we are constrained by the existing language and focus our work on the design of the environment. Even though TC is a small visual language, based on a straightforward node-link diagram notation, a designer of an interactive environment faces a multitude of design questions. In this section we present those questions to outline the design space.

Node creation How does the user specify the kind of node to create? Is there a *palette* of available nodes from which to drag and drop? If a large number of nodes is provided, how can the user find the right node quickly? How are literal nodes (e.g., numeric or string literal) created?
Connection How does the user specify links between nodes? Do links have to be created in the direction of data-flow, from an outlet to an inlet?
Layout Is the user responsible for laying out the nodes or is the layout done automatically? Is the membership of a node in a constant/function definition determined by its placement (within the area of the definition) or by its connectivity (within the subgraph corresponding to the definition)?
Selection Does the editor allow selecting a node, a link, a subgraph? What further actions are enabled by providing selection?
Copying Does the editor allow copy-pasting a node, a link, a subgraph?
Deletion Does the editor allow the deletion of an individual node, an individual link, a selected subgraph?
Edit History Are undo and redo operations allowed?
Abstraction The TC language supports two forms of abstraction: user-defined *constants* and *functions*. How does the user create a new abstraction? Does the editor provide support for abstracting by refactoring (e.g., an "extract constant" refactoring, similar to Figure 6 of [19])? Can a program contain subgraphs that are not within any definition (top-level expressions)? Does the editor provide a separate canvas for each constant/function definition or are they all shown on a single canvas? Can function/constant definitions contain other function/constant definitions?
Well-formedness checking or enforcement (Problems that have no direct correspondence in Python.) Does the editor allow diagrams that are not well-formed, or does it prevent them? Can one connect two outlets or two inlets? Are cycles allowed? Can an inlet have multiple incoming links? Can an outlet have multiple outgoing links (sharing of sub-expressions)? If well-formedness violations are allowed, what feedback does the editor provide?
Syntax checking or enforcement (Problems that can be related to a `SyntaxError` in Python.) Does the editor allow syntactically incorrect programs? Can an inlet have no incoming link (can an expression have holes [22])? If syntax violations are allowed, what feedback does the editor provide?
Name analysis or enforcement (Problems that can be related to a `NameError` in Python.) Does the editor allow the use of constants or functions that have not been defined? Does the editor allow the definition of constants or functions with names that are not legal Python identifiers? Does the editor allow multiple definitions of the same name? Does the editor consider that constant names and function names share the same name space? Does the editor allow renaming? Does the editor provide support for navigating between definition and uses of a name? Can name definitions contain cyclical dependencies? If name binding violations are allowed, what feedback does the editor provide?
Type checking or enforcement (Problems that can be related to a `TypeError` in Python.) Does the editor allow programs with type errors? How should the type system used by the editor relate to the type system of Python? If type errors are allowed, what feedback does the editor provide?
Literal checking or enforcement (Problems corresponding to lexical errors with respect to literal tokens.) The TC language supports different types of literal nodes (e.g., floats and strings). Is there a separate kind of literal node for each type? How does the user input the value of a literal node? Does the user specify the type of the literal or does the editor automatically determine it based on the user input? Does the editor allow illegal literals (e.g., literals that would not be valid lexical tokens in Python)? If illegal literals are allowed, what feedback does the editor provide?

Program Evaluation Does the editor automatically evaluate the TC expressions and annotate nodes with their values? Where and when are values shown? Does the editor show values of all types? How are values of different types visualized? How do errors (due to well-formedness, syntax, name, type, or evaluation errors in preceding nodes) affect evaluation, and what kind of feedback does the editor provide?
Annotation Does the editor support annotations? Does it allow annotations of individual nodes, links, subgraphs, and free-standing annotations at arbitrary locations? How are annotations created? Do annotations of nodes, links, or subgraphs move when the nodes, links, or subgraphs move?

5 Design

We now describe the design of TIDE, which represents a specific point in the design space defined in Section 4. Moody states that design rationale is conspicuously absent in the design of visual notations in software engineering [20]. We address this issue by providing a rationale for our design decisions, based on the existing visual language (TC), the underlying programming language (Python), the library (PyTamaro), and the educational goals. These goals include teaching a purely functional approach to programming, where pieces of programs are independent and thus can easily be composed. Moreover, they include teaching the benefits of static typing [23], despite the underlying language, Python, not being statically typed.

5.1 Node Management

Node Creation TIDE provides a palette from which the user drags nodes onto the canvas (left side of Figure 6). The palette shows documentation cards (with their explanatory annotations), while the canvas shows code cards. The set of nodes to be shown in the palette can be configured. To quickly find a node, a search field allows for incremental search by name. The palette also includes nodes for literals, which are pre-populated with common default values (one for type `float`, one for `str`, and two for the two `bool` values).
Connection Users have to explicitly establish connections between ports by dragging from one port to the other. The direction in which the connection is established does not matter, because we believe that the educational benefit of students having to drag connections in the direction of data-flow is dwarfed by the user dissatisfaction when unsuccessfully trying to establish a connection in the opposite direction, an interaction that is commonly possible in node-link diagram editors. The tips of the arrows always point towards the right, clearly indicating the direction in which data flows as the expression is evaluated. In the middle of Figure 6, various cards are connected together to compose a graphic of a stylized house.
Layout In TIDE, users freely arrange nodes. Automatic layout of nodes is not supported: early feedback from teachers indicated that there is value in having students think about properly placing nodes on the canvas. Cards used in constant definitions can be also arranged freely: the boundaries grow dynamically to encompass the entire tree connected to the terminal of the definition, as seen for the definition of the `arm` constant in Figure 7.
Selection Users can select a single node or link by clicking on it (e.g., the `rectangle` node of Figure 6 is shown "raised" to indicate that it is currently selected), and they can select a subgraph by rubber-banding. Selected pieces can be freely moved around on the diagram. Besides its purpose for diagram manipulation, selection can also have a pedagogical benefit. During a live-coding session, an instructor may select a group of components to highlight them during an explanation.
Copy & Paste TIDE allows users to create a copy of the selected nodes and links. This is beneficial for creating graphics that consist of multiple similar components, or to create alternative variants of a program that are juxtaposed on the same canvas. We are acutely aware that copying a part of a (graphical or textual) program means introducing *code clones*. We plan to explore ways to exploit this action to suggest refactorings to use language abstractions (such as extracting common sub-expressions into constants or functions).
Deletion To support experimentation, we allow students to delete individual links, nodes, and the selected subgraph.
Edit History TIDE supports undoing and redoing edits. This addresses the problem of accidental deletion, especially in the case a larger subgraph was selected. It also allows students to explore alternatives, and to go back to a previous, working solution. However, TIDE does not provide advanced features, such as a list of prior edits, or the maintenance of several alternative branches in the edit history. While these might be beneficial in some cases, the additional complexity of their user interface would distract from the essential task of composing programs.

5.2 Abstraction

TIDE currently supports abstraction through constant definitions, but does not yet support function definitions. The user defines a constant by selecting a sub-expression and picking "extract constant" from a sub-menu. All definitions, as well as top-level expressions, co-exist on one canvas. Whether or not a node is part of a constant definition is determined by connectivity: if a node's outlet is transitively connected to the terminal of the constant definition, the node is considered part of that definition, and the definition's visual rectangle automatically expands to surround that node. Definitions cannot be nested; all definitions are top-level definitions shown on the same canvas. Figure 7 illustrates the definition of a constant (`arm`): the expression that produces the value is connected to the terminal of the constant definition (blue semicircle on its right). Then, the name can be used in a

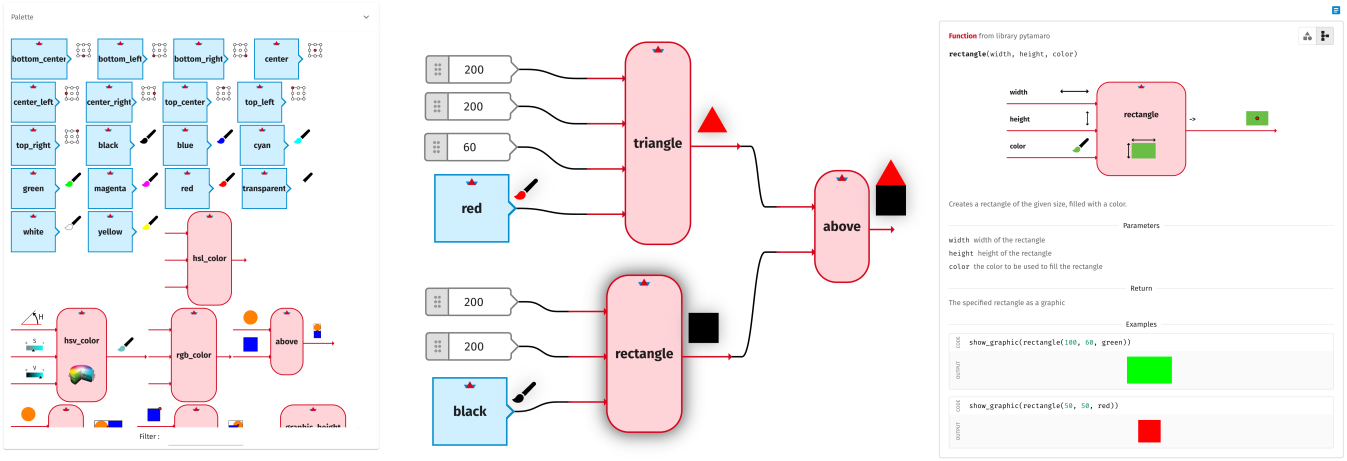

Figure 6. The TamaroCards IDE. Left: *palette* from which the user can drag cards onto the canvas. Center: *canvas* with the cards composing the expression `above(triangle(200, 200, 60, red), rectangle(200, 200, black))` to draw a house. Right: *documentation* for the selected card (the function `rectangle`).

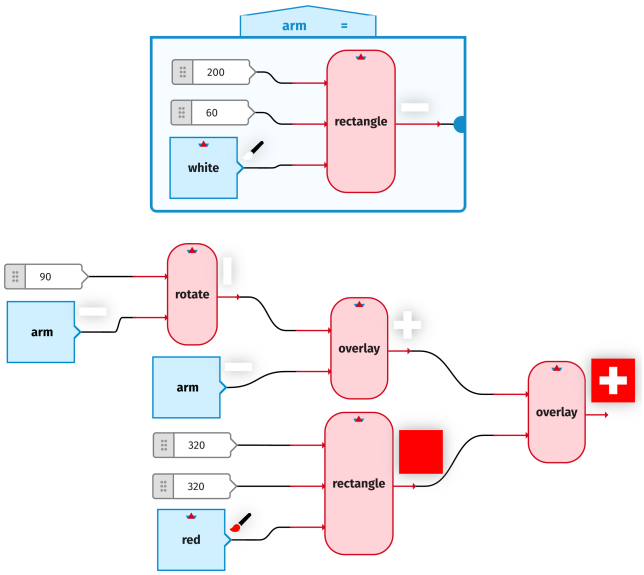

Figure 7. The arms of the cross in the Swiss flags are abstracted to the `arm` constant. The name is then used twice to compose the flag.

bigger expression to compose a more complex graphic (e.g., the cross of the Swiss flag).

5.3 Well-Formedness Checking or Enforcement

TIDE prevents users from establishing connections that would result in a diagram that is not well-formed. Students have been shown to make well-formedness mistakes in expression tree diagrams, such as creating cycles, building incomplete trees, and connecting multiple sibling expressions to the same inlet [1]. However, we decided to prevent these kinds of mistakes to focus the students' learning on problems that can also occur in Python.

A well-formed TC program must correspond to a set of top-level expressions with possible holes and a set of constant definitions. Any link must connect one outlet to one inlet (or to the terminal of a constant definition). It is illegal to connect multiple outlets to an inlet. While connecting one outlet to multiple inlets would be a practical way to avoid code duplication within the language of the diagram, TIDE disallows this because there is no corresponding construct in Python. It is important for students to learn to avoid code duplication [6], but we want them to learn to use the facilities of Python (e.g., constants) to do so.

A well-formed expression must not be cyclic. TIDE ensures that expressions are acyclic by preventing the creation of links from an outlet to an inlet of the same node or its predecessors.

A well-formed TC program does not necessarily need to be complete. We do not classify an open *inlet* or *terminal* as a well-formedness mistake, but we treat them as holes and report them as syntax errors (see Section 5.4). Moreover, we allow open *outlets* without generating any error: they correspond to top-level expressions in the program. Allowing holes enables users to construct expressions top-down, and allowing open outlets enables them to construct expressions bottom-up.

5.4 Syntax Checking or Enforcement

TIDE allows users to create well-formed but syntactically incorrect programs. Unlike the well-formedness errors, which TIDE disallows, syntax errors can happen in TC as well as in Python. Thus, students making syntax errors in TC will be able to transfer their familiarity with these kinds of problems

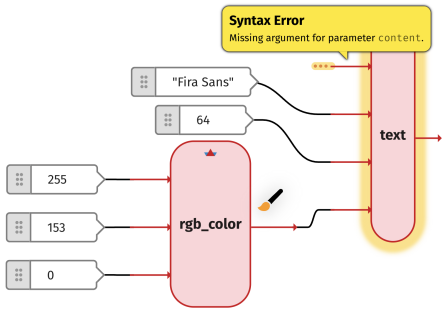

Figure 8. Example of a `SyntaxError`: one of the arguments of the `text` function is missing. The first inlet shows an ellipsis (`...`). Hovering over the ellipsis shows an error that indicates a missing argument for the `content` parameter.

to their future work in Python. TIDE reports several kinds of syntax errors:

Holes in Expressions The presence of a hole (a missing argument for a parameter, e.g., in `overlay(■, arm)`) is considered a syntax error. Our evaluation strategy, upon finding expression holes, prevents the evaluation of the expression. According to Omar et al., this is the standard approach for programming languages [22]. TIDE highlights holes with an ellipsis to show the missing sub-expression, a yellow highlight of the corresponding inlet, and a tooltip with the error message (Figure 8). The text in the tooltip deliberately uses wording similar to the error messages Python provides (e.g., the term "Syntax Error"), to familiarize students with the error messages they will be confronted with in Python. The ellipsis (`...`) was chosen because the Python language includes an ellipsis that sometimes is used to denote a hole in expressions (most teachers developing assignments for PyTamaro follow this convention). The color of the tooltip and the error highlight (yellow) was chosen to be easily distinguishable from the fill colors of TC nodes (light red, green, and blue).
Allowing this kind of mistake enables students to construct expressions in a top-down manner, starting with a top-level function call, and incrementally building sub-expressions for each argument. Without the ability to have incomplete expressions with holes, TIDE would prevent such a top-down problem decomposition approach.
Holes in Constant Definitions A hole can similarly appear when a constant is defined without an initialization expression. This is equivalent to a Python assignment statement without an expression (e.g., `arm = ■`). TIDE shows a syntax error if no nodes are connected to the terminal of the constant definition.
Invalid Names The definition or use of invalid names (e.g., `answer? = 42`) is also reported as a syntax error. TIDE checks that constant names follow the convention for valid Python identifiers.

Python source code can have many other kinds of syntax errors as well, such as missing or superfluous parentheses, but the TC language (similar to a block-based language) makes such mistakes impossible by construction.

5.5 Name Analysis or Enforcement

TIDE supports the definition of constants. In a constant *definition*, the name is shown in the "roof", to the left to the equals sign (Figure 7). In a constant *use*, the name is the sole content of the node.

When declaring a new constant, the user is asked to provide its name. The use of illegal identifiers is reported as an error. Specifically, the editor marks the usage of names that have already been defined (e.g., `red`) and Python keywords[6] (e.g., `return`) as name and syntax errors, respectively. The user can rename the constant by clicking on the name in the constant definition or the constant use. TIDE deliberately does not provide a rename refactoring: this teaches students that names are what ties uses and definitions together, and thus that when they rename a definition, they need to appropriately rename all the uses as well.

However, if the user opens a context menu on a constant, it is possible to navigate to the constant definition.

Cyclic dependencies are reported as a "Name Error" by TIDE. The definition of a constant `a` cannot depend on another constant `b` if the definition of `b` itself depends on `a`. This would normally not be possible in Python, because statements are ordered and executed sequentially, while the TC visual language does not have the concept of an order. Figure 9 shows how TIDE reports such an error.

In Python, a `NameError` is "raised when a local or global [unqualified] name is not found"[7]. TIDE reports a "Name Error" if it cannot find a constant definition for a name used in a constant use. Similarly, it signals a "Name Error" if multiple constant definitions define the *same* name. This is necessary because, unlike the full Python language, our sub-language only allows the definition of names once (no multiple definitions or assignments).

5.6 Type Checking or Enforcement

The type system of TC as implemented in TIDE is entirely monomorphic and only supports a predetermined set of types: Python's `float`, `str` and `bool`, plus `Graphic`, `Color`, and `Point` from the PyTamaro library.

Following the pedagogical idea that students learn from making mistakes, and that mistakes in TIDE are easier to understand than mistakes made in Python, TIDE allows programs with type errors.

According to the Python specification, a `TypeError` is "raised when an operation or function is applied to an object

[6] The Python Language Reference » 2. Lexical analysis » 2.3. Identifiers and keywords
[7] The Python Standard Library » Built-in Exceptions » `NameError`

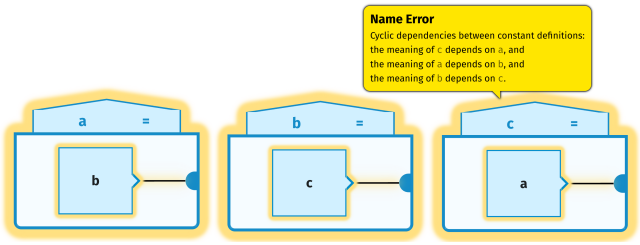

Figure 9. Example of a `NameError`: a cyclic dependency. Variable a depends on b, which depends on c, which in turn depends on a. This error is shown on all nodes. Hovering over one shows a popup that highlights the chain of dependencies.

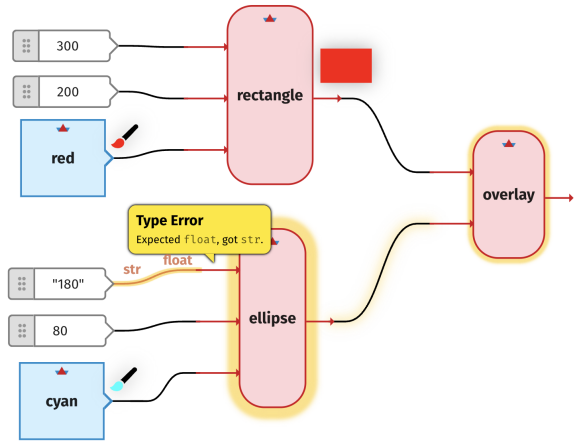

Figure 10. The type of the first argument of `ellipse` is incorrect, resulting in a `TypeError`. The `ellipse` node is highlighted with a thick yellow border. Hovering over the inlet of the function shows a popup that describes the error referring to the mismatching types. As a consequence of this error, the card for the `overlay` function cannot be evaluated and is shown with a thinner yellow border.

of inappropriate type" and when "passing arguments of the wrong type (e.g. passing a `list` when an `int` is expected)"[8].

TIDE checks that the types at both ends of a link (the outlet of a literal, constant, or function, and the inlet of a function) agree, and it reports type errors for every disagreement. Links with type mismatches are highlighted in yellow. The error tooltip shown next to the inlet reports both the actual type at the outlet and the expected type at the inlet (Figure 10).

5.7 Literal Checking or Enforcement

The TC language contains literals for the supported Python types (Section 5.6). The paper-based literal card is simply an empty white card, on which a student can freely write. Similarly, the literal nodes of TIDE allow users to enter arbitrary text. TIDE checks the text to automatically determine the type of the literal value. If the type of a literal cannot be determined (e.g., for `"hi`), a "Syntax Error" is shown.

While this could be prevented by providing a more restrictive user interface (e.g., a node with pre-written quotation marks around the text, a slider to choose a number, and a checkbox for a Boolean literal respectively), we deliberately allow users to make the same kinds of mistakes they might make when writing Python source code.

5.8 Program Evaluation

TIDE is a live programming environment. It automatically analyzes and evaluates the program after each change, and shows the corresponding values and errors, as seen in Figure 6. The main purpose of the PyTamaro library is to allow students to work with values that are algebraic graphics (a few primitive graphics and ways to compose them). By looking at a value of type `Graphic` (e.g., a house), students immediately see its structure (e.g., a roof placed above a wall), which often quite closely corresponds to the structure of the expression that produced it. Graphics are shown above the outlet of function calls that produce them, helping students understand how the expression is composed.

Graphics can have an arbitrary size. TIDE shows all graphics at the same scale, so that component graphics have the same size when shown in isolation and when shown as part of a bigger graphic.

Currently, TIDE only shows values of types `Graphic` (by showing the graphic itself) and `Color` (by showing a colored paintbrush icon). Although it would be possible to show other types of values, additional annotations would clutter the diagram and thus reduce the emphasis on the composition of graphics.

As a live programming environment, TIDE keeps all values up-to-date when the user edits the program. To avoid jarring effects when entering literal values, which are often used to define sizes, the update of values is debounced.

Violating a function's precondition leads to an evaluation error, and thus the resulting graphic is not shown. TIDE does not currently report runtime errors, so it does not show any error annotation for a function precondition violation.

An important feature of TIDE is that programs containing errors of any kind (syntax, name, or type errors) are still partially evaluated. An error only prevents the evaluation of the parts of an expression that depend on the erroneous nodes (Figure 10 shows an example).

In the context of the framework proposed by Tanimoto, used to classify the "liveness" of a programming environment [28], TIDE sits on the fourth level (Informative, significant, responsive and live).

[8] The Python Standard Library » Built-in Exceptions » `TypeError`

5.9 Annotations

TIDE uses annotations for various purposes. It annotates function nodes that produce graphics or colors with the value they produce. If nodes cannot be evaluated, they are instead highlighted as an error and annotated with tooltips that are shown when selecting or hovering over the node itself. Unlike the canvas, which shows code cards, the palette shows documentation cards with their additional annotations that help illustrate the purpose of each parameter and include exemplar values [24]. In case of type errors, the links are annotated with the mismatching types.

TIDE does not support user-defined annotations. The absence of such a feature forces students to use a clean layout to communicate clearly, and to introduce proper abstractions (named constants) with good names, a skill that will transfer to writing code in Python later on.

6 Evaluation

We evaluate TIDE using the Cognitive Dimensions [14] framework. While we focus on evaluating the interactive environment, the evaluation necessarily is affected by the existing visual TC language, the PyTamaro library, and the underlying Python language. The Cognitive Dimensions framework consists of 13 dimensions. For each dimension, we begin the discussion by citing its short description [14, Section 3].

6.1 Abstraction Gradient

What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?

The declaration of constant names to allow the re-use of values is the only supported form of *abstraction* in TIDE. Using constants is not strictly necessary, because all expressions in TC are pure and thus names can always be substituted with the expression they stand for. This puts TIDE in what Green and Petre define as the *abstraction-tolerant* group. Providing the ability to move along the abstraction gradient (from a single deeply-nested expression devoid of named constants to a program providing named constants for meaningful subexpressions) is essential for an environment where students are supposed to learn to abstract.

6.2 Closeness of Mapping

What 'programming games' need to be learned?

The only programming constructs used by TCs are functions, constants and literals. All TC functions are pure: there are no side effects. One can reason about a TC function as a mathematical function: it consumes some values to produce another value. The "programming world" (functions) of the TC language is close to its "program world" (graphics). A function either produces a primitive graphic or composes two graphics together. The composition operations between graphics can be easily reasoned about even without considering the semantics of a programming language.

6.3 Consistency

When some of the language has been learned, how much of the rest can be inferred?

The very limited number of constructs available in the TC visual language makes it easy for someone to learn the *building blocks* of the language and compose new programs. One only needs to learn how to apply functions by connecting cards together, assuming the types match. TIDE uses Judicious [8] to provide the documentation for each card that represents a function, including a brief description of what the function does, names and types for each parameter, the return type of the function, and simple examples with the corresponding outputs.

TIDE uses a consistent way to report errors, using the same yellow color for highlighting cards and links, and for presenting tooltips with error messages. Moreover, it uses the same error names (syntax, name, and type error) students will encounter once they move to Python.

6.4 Diffuseness

How many symbols or graphic entities are required to express a meaning?

The number of symbols required to express a computation using TCs in TIDE is consistently less of that of Python. A number of tokens, such as parentheses (required to express precedence) and commas (used to separate arguments), are no longer required due to the visual nature of the TC language.

6.5 Error-Proneness

Does the design of the notation induce 'careless mistakes'?

TIDE prevents well-formedness mistakes (Section 5.3) that would only exist in the visual environment and have no correspondence in Python. It deliberately focuses the possible mistakes to those that are pedagogically meaningful (Sections 5.4 to 5.6). Unlike Scratch, with its "failsoft" approach that avoids all error messages [17], TIDE allows students to make errors in a simple environment and provides targeted feedback to help students learn from them.

6.6 Hard Mental Operations

Are there places where the user needs to resort to fingers or pencilled annotations to keep track of what's happening?

TIDE's main benefit is its Babylonian approach to live programming by interleaving code and values. This reduces the otherwise hard mental operation of program tracing into small, tractable steps.

When expressions are composed further and further, they become large and hard to work with. TIDE provides a mechanism to deal with this problem: by giving names to specific sub-expressions, a user can decompose expressions and then use those names to compose simpler expressions.

This approach falls short in case of sub-expressions that are similar but not identical. Handling this case requires allowing abstraction using user-defined functions, a construct that is not currently supported in TIDE.

6.7 Hidden Dependencies

Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?

The TC language does not allow any form of impurity: all functions are pure and names cannot be redefined. This guarantees that all the dependencies within an expression are clearly indicated by the arrows that connect the cards. Cyclic dependencies within an expression are not allowed (Section 5.5).

However, TIDE does currently not visualize the connection of name uses to their definitions (such as the "binding arrows" shown by Dr. Racket and its predecessor Dr. Scheme, as illustrated in Figure 7 of Findler et al. [12]). This makes it harder to see the dependency structure of a program with definitions.

6.8 Premature Commitment

Do programmers have to make decisions before they have the information they need?

In TIDE, users can refactor sub-expressions into constants. The order in which constants are created and placed on the canvas is irrelevant. Moreover, the purity of the language allows for sub-problems to be solved independently, a property inherited from PyTamaro. This frees the users from having to clearly define how to tackle the composition of a graphic at the very beginning, encouraging exploration and refactoring of complex expressions.

Moreover, TIDE's undo history reduces the risk of editing, and the ability to copy and paste entire subgraphs allows the convenient exploration of alternatives.

6.9 Progressive Evaluation

Can a partially-complete program be executed to obtain feedback on 'How am I doing'?

TIDE shows the results of evaluating expressions and their sub-expressions. When information is missing because the expression is incomplete (e.g., a missing argument) or at least one of the sub-expressions has in turn an error, an expression cannot be fully evaluated. In such situations, the program is partially evaluated: partial results are rendered for complete sub-expressions, while localized error information is given as a form of feedback to guide the user towards a state where a program can be fully evaluated. TIDE also highlights the links that propagate the error, helping students to find the root cause quickly. The node that is the source of the error is highlighted with a thick yellow border (e.g., the `ellipse` node in Figure 10). All nodes that can no longer be evaluated as a consequence of errors in other nodes are instead highlighted with a thinner yellow border (e.g., the `overlay` node in Figure 10).

When showing a prototype of TIDE to a teacher who extensively uses TC in their classroom, their immediate response was that seeing the values, including the intermediate ones, was great. Their second response was that a teacher needs to be able to disable the live evaluation for pedagogical reasons. Automatically evaluating the code deprives students of a learning opportunity: they should learn to "trace" through a program, i.e., to evaluate it by themselves (a form of *visual program simulation* [26]). When students work with the paper version of TC, one of the key activities they do is tracing through the evaluation of the expression by drawing the graphics produced by each card. If TIDE always immediately shows the evaluation results, there is a risk that students stop reasoning about the code and resort to a trial-and-error approach until the resulting graphic looks as expected.

6.10 Role-Expressiveness

Can the reader see how each component of a program relates to the whole?

Different language constructs have different visual notations. Functions are represented in red, constants in blue, literals in white (grayscale), and errors in yellow. The same blue is used both in the definitions and uses of constants. A consistent yellow is used to signal errors; the thickness of the yellow border distinguishes between the node that is the source of the error and the nodes that consequently error. Constant use and literal cards have similar visual outlets, which are different from the outlet of functions: functions need to be *called* to be used, while literals and constants are used as-is. The arrows in a function card clearly show the direction in which data flows to see how the program works.

6.11 Secondary Notation

Can programmers use layout, color, other cues to convey extra meaning, above and beyond the 'official' semantics of the language?

TIDE does not offer means to annotate a TC program beyond the live evaluation results and errors. However, users can freely arrange cards on the canvas, and they can organize code into named constants. This teaches students to pick descriptive names to denote meaningful subexpressions, a skill that will be valuable in Python.

6.12 Viscosity

How much effort is required to perform a single change?

TIDE has moderate viscosity: performing edits in a graph-like structure may require rearranging the positions of the nodes. Most of the time, local changes do not require much rearranging, as expressions grow left to right. TIDE does not offer features to automatically layout the nodes on the

canvas, but it allows to operate on multiple nodes at the same time. This makes it easier to change the position of a group of nodes. Users can select multiple nodes (and their links) either individually or by selecting all nodes within a certain rectangular area. Selected elements may then be moved, copied or deleted. Moving a constant definition will also move all the nodes that are part of the definition. Which nodes are part of a constant definition is defined by (indirect) connection to the terminal of the definition. Simply connecting (or disconnecting) nodes will bring them inside (or outside) of a constant definition group. It is also possible to select a subtree or its root and refactor it into a constant definition in one single action, similar to Figure 6 of [19].

6.13 Visibility

> *Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to juxtapose any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?*

It is possible to freely move and zoom across the canvas to adjust the view of the program. The *palette* is shown as a popup panel that can be opened on demand. The *documentation* for a function represented by a card can be shown by selecting it. *Errors* show up as tooltips near the relevant cards and are also grouped in a collapsible list at the top-right corner of the canvas. Clicking on an error in that list zooms the canvas to the part of the diagram relevant for the error. When a constant is user-defined, it is possible to jump from a constant use to its definition through a context menu.

Juxtaposability is an important aspect of visibility. Tools such as Code Bubbles [2] allow users to create bubbles containing code fragments, which they can place on a canvas. TIDE allows nodes to be freely placed; given that nodes need to be connected by links, the layout of an expression is usually not too dissimilar from its tree structure. By moving around sub-trees, users can place related sub-trees near each other. More importantly, however, by extracting sub-trees into constants, the constant definitions become disconnected from their uses, and the definitions (similar to a code bubble), can be freely placed where they are most helpful. This extra freedom of placement afforded by creating constants provides an additional motivation for students to practice abstraction by creating constants. Moreover, the "bubble" representing a constant shows the constant's name in its header, to reinforce the need for choosing good names.

7 Limitations and Future Work

The paper-based TC language includes user-defined functions, but the current implementation of TIDE does not yet support it. Function definitions are an essential abstraction mechanism, and we plan to extend TIDE to support user-defined functions.

In the current implementation of TIDE, all numbers are treated as floating-point numbers rather than having multiple numerical types. The only PyTamaro function that has parameters annotated with type `int` is the `rgb_color` function, which expects three integer numbers (between 0 and 255) as the three color components, but its implementation still works when provided with arguments of type `float` (the numbers are rounded to the nearest `int` value). In the future, we plan to extend our type system to add an integer number type. All functions that produce graphics in PyTamaro always take floating-point values as numerical arguments.

TIDE does not display annotations for runtime errors such as `ValueError`s that would be produced, for example, when providing a primitive graphic function with a negative size.

While the paper-based TC provides cards for Python operators, TIDE still lacks the corresponding functionality.

The availability of TIDE provides the basis for empirical evaluations that could provide insights into several open questions: (1) Is a physical or a virtual implementation of a visual language like TC more beneficial for learning? (2) In which ways does automatic live evaluation help or hinder learning? (3) Which affordances are most effective in teaching for transfer from a visual language like TC to textual code in a language like Python?

8 Conclusions

We described the design space for an educational visual programming environment focused on the composition of graphics that is supported by the PyTamaro library. Starting with the paper-based visual TC language, and driven by the corresponding pedagogical goals, we characterized a promising point in the design space, implemented it in the form of TIDE, and evaluated it using the Cognitive Dimensions framework.

We plan to deploy TIDE to make it publicly available, allowing teachers to adopt it in their classrooms as an alternative to paper-based TC. Based on a session using our initial TIDE implementation, the teacher who originally encouraged the creation of an interactive environment to complement her use of paper-based cards, expressed her intention to adopt TIDE in the upcoming school year. This real-world adoption will provide the basis for future studies of the educational benefits of aspects such as live evaluation and support for abstraction.

Acknowledgments

This work was partially funded by the Swiss National Science Foundation project 200021_184689.

References

[1] Joey Bevilacqua, Luca Chiodini, Igor Moreno Santos, and Matthias Hauswirth. 2024. Using Notional Machines to Automatically Assess Students' Comprehension of Their Own Code. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2.* ACM, Portland OR USA, 1572–1573. https://doi.org/10.1145/3626253.3635524

[2] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. 2010. Code Bubbles: A Working Set-Based Interface for Code Understanding and Maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. Association for Computing Machinery, New York, NY, USA, 2503–2512. https://doi.org/10.1145/1753326.1753706

[3] Anke Brocker, René Schäfer, Christian Remy, Simon Voelker, and Jan Borchers. 2023. Flowboard: How Seamless, Live, Flow-Based Programming Impacts Learning to Code for Embedded Electronics. *ACM Trans. Comput.-Hum. Interact.* 30, 1 (March 2023), 2:1–2:36. https://doi.org/10.1145/3533015

[4] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. 2013. It's Alive! Continuous Feedback in UI Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 95–104. https://doi.org/10.1145/2491956.2462170

[5] Paul Chandler and John Sweller. 1992. The Split-Attention Effect as a Factor in the Design of Instruction. *British Journal of Educational Psychology* 62, 2 (1992), 233–246. https://doi.org/10.1111/j.2044-8279.1992.tb01017.x

[6] Luca Chiodini, Joey Bevilacqua, and Matthias Hauswirth. 2025. The Toolbox of Functions: Teaching Code Reuse in Schools. In *Proceedings of the 6th European Conference on Software Engineering Education (ECSEE '25)*. Association for Computing Machinery, New York, NY, USA, 185–189. https://doi.org/10.1145/3723010.3723029

[7] Luca Chiodini, Igor Moreno Santos, and Matthias Hauswirth. 2022. Expressions in Java: Essential, Prevalent, Neglected?. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on SPLASH-E (SPLASH-E 2022)*. ACM, New York, NY, USA, 41–51. https://doi.org/10.1145/3563767.3568131

[8] Luca Chiodini, Simone Piatti, and Matthias Hauswirth. 2024. Judicious: API Documentation for Novices. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on SPLASH-E (SPLASH-E 2024)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/3689493.3689987

[9] Luca Chiodini, Juha Sorva, and Matthias Hauswirth. 2023. Teaching Programming with Graphics: Pitfalls and a Solution. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E (SPLASH-E 2023)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3622780.3623644

[10] Luca Chiodini, Juha Sorva, Arto Hellas, Otto Seppälä, and Matthias Hauswirth. 2025. Two Approaches for Programming Education in the Domain of Graphics: An Experiment. *The Art, Science, and Engineering of Programming* 10, 1 (Feb. 2025), 14:1–14:48. https://doi.org/10.22152/programming-journal.org/2025/10/14

[11] Sally Fincher, Johan Jeuring, Craig S. Miller, Peter Donaldson, Benedict du Boulay, Matthias Hauswirth, Arto Hellas, Felienne Hermans, Colleen Lewis, Andreas Mühling, Janice L. Pearce, and Andrew Petersen. 2020. Notional Machines in Computing Education: The Education of Attention. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR '20)*. Association for Computing Machinery, New York, NY, USA, 21–50. https://doi.org/10.1145/3437800.3439202

[12] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1997. DrScheme: A Pedagogic Programming Environment for Scheme. In *Programming Languages: Implementations, Logics, and Programs*, Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Hugh Glaser, Pieter Hartel, and Herbert Kuchen (Eds.). Vol. 1292. Springer Berlin Heidelberg, Berlin, Heidelberg, 369–388. https://doi.org/10.1007/BFb0033856

[13] Emily R. Fyfe and Mitchell J. Nathan. 2019. Making "Concreteness Fading" More Concrete as a Theory of Instruction for Promoting Transfer. *Educational Review* 71, 4 (July 2019), 403–422. https://doi.org/10.1080/00131911.2018.1424116

[14] T. R. G. Green and M. Petre. 1996. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing* 7, 2 (June 1996), 131–174. https://doi.org/10.1006/jvlc.1996.0009

[15] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in Dataflow Programming Languages. *ACM Comput. Surv.* 36, 1 (March 2004), 1–34. https://doi.org/10.1145/1013208.1013209

[16] Eva Krebs, Toni Mattis, Patrick Rein, and Robert Hirschfeld. 2023. Toward Studying Example-Based Live Programming in CS/SE Education. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT 2023)*. Association for Computing Machinery, New York, NY, USA, 17–24. https://doi.org/10.1145/3623504.3623568

[17] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education* 10, 4 (Nov. 2010), 1–15. https://doi.org/10.1145/1868358.1868363

[18] Sean McDirmid. 2007. Living It up with a Live Programming Language. *SIGPLAN Not.* 42, 10 (Oct. 2007), 623–638. https://doi.org/10.1145/1297105.1297073

[19] Michael J. McGuffin and Christopher P. Fuhrman. 2020. Categories and Completeness of Visual Programming and Direct Manipulation. In *Proceedings of the 2020 International Conference on Advanced Visual Interfaces (AVI '20)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3399715.3399821

[20] D. Moody. 2009. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering* 35, 6 (Nov. 2009), 756–779. https://doi.org/10.1109/TSE.2009.67

[21] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2018. Live Functional Programming with Typed Holes. https://doi.org/10.48550/arXiv.1805.00155 arXiv:1805.00155 [cs]

[22] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. Association for Computing Machinery, New York, NY, USA, 86–99. https://doi.org/10.1145/3009837.3009900

[23] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, Mass.

[24] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-Style Programming. *The Art, Science, and Engineering of Programming* 3, 3 (Feb. 2019), 9:1–9:39. https://doi.org/10.22152/programming-journal.org/2019/3/9

[25] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. https://doi.org/10.1145/1592761.1592779

[26] Juha Sorva. 2012. *Visual Program Simulation in Introductory Programming Education*. Ph. D. Dissertation. Aalto University, Espoo, Finland.

[27] William Robert Sutherland. 1966. *The On-Line Graphical Specification of Computer Procedures*. Thesis. Massachusetts Institute of Technology.

[28] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming (LIVE '13)*. IEEE Press, San Francisco, California, 31–34.

[29] Anthony Trory, Kate Howland, Judith Good, and Benedict Du Boulay. 2024. Physical vs. Virtual Representations Within Concreteness Fading for Primary School Computing. In *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE Press, Liverpool, United Kingdom, 71–80. https://doi.org/10.1109/VL/HCC60511.2024.00018