Applying a Lightweight Graphics Library to Visually Corroborate Learning in Programming Introduction Courses

Claudia De La Fuente[1], Cristian Vidal-Silva[2], Liza Jegó-Mendoza[3], Patricia Pedrero-Valenzuela[4]
Facultad de Ingeniería, Universidad de Talca, Talca, Chile[1,2,3]
Departamento de Administración de Educación Municipal, Talca, Chile[4]

***Abstract*—Learning to program in first-year courses is challenging because the link between source code and program behaviour is not immediately visible to novices. This paper reports on the deployment of UFramework, a lightweight graphics library developed in C++/Visual Studio, designed to help students visually corroborate their learning by observing the on–screen effects of their own algorithms. The experience was implemented in a Structured Programming module that combines lectures, labs, and project-based assignments. We 1) describe the design principles and architecture of the library, 2) present a portfolio of progressively scaffolded assignments (shooter prototype, grid map parsing, spatial quadrants), 3) outline the assessment rubric and its alignment with intended learning outcomes, and 4) report multi-year descriptive evidence that includes pass rates and qualitative reflections. Results show improved student engagement and higher pass rates in the most recent cohorts, together with qualitative evidence of increased motivation and clearer problem decomposition. While the findings are limited to a single institution and remain descriptive, they suggest that lightweight, visual-first workflows can lower barriers to learning programming and foster computational thinking competencies such as decomposition, abstraction, algorithmic design, and debugging. Future work should include controlled comparisons and broader validation to strengthen internal validity and explore the applicability of this approach to non-visual domains.**

Keywords*—Structured programming; visual corroboration; self-regulated learning; metacognition; program visualization; project-based learning

I. Introduction

Learning to program in first-year courses is challenging because the relationship between source code and program behaviour is not immediately visible to novices. Without timely, interpretable feedback, misconceptions persist and motivation declines [1], [2], [3]. To reduce this gap, we deployed UFramework, a lightweight graphics library in C++ (Visual Studio toolchain) that allows students to see the effect of loops, conditionals, arrays, and simple state machines with minimal setup. The design intention is to preserve working memory for algorithmic reasoning while delivering immediate, motivating, on-screen feedback [4], [5].

In our institution, the intervention was embedded in a first-year Structured Programming module that combines lectures, lab sessions, and project-based assignments. Students implemented small visual programs and documented decisions in short reflective notes. Fig. 2 provides an overview of the dominant terms appearing in reflections and forum posts—learning, students, programming, education—which is coherent with the course aims.

1) Rationale: Two strands of previous work underpin the approach. First, program visualization and animation help externalize state changes and support comprehension for novices [1], [2]. Second, designs that minimize extraneous cognitive load and support self-regulated learning enable students to devote resources to core schemas and problem solving [4], [5]. We also draw on project-based learning (PBL) to sustain engagement and cultivate productive habits for introductory programming [6], [7], and we align activities with competencies commonly associated with computational thinking [8], [9].

2) Objectives and research questions: The study seeks practical improvements in introductory programming while documenting the design so others can reuse it. We address the following questions:

RQ1. Does a lightweight, visual workflow help students connect code with behaviour in early assignments? RQ2. How do scaffolded visual tasks map to intended learning outcomes (decomposition, abstraction, testing)? RQ3. What descriptive changes are observable in pass rates and student reflections across cohorts?

TABLE I. From Challenges to Design Decisions. The Mechanisms are Intentionally Minimal to Reduce Setup Cost and Cognitive Overhead

Challenge	Design mechanism	Intended effect
Invisible program state	On-screen visual outcomes for each step	Comprehension
High setup overhead	Small, discoverable API and starter templates	Time on task
Early frustration	Short, game-flavoured tasks with rapid feedback	Motivation
Shallow assessment	Rubric beyond correctness (decomposition, docs)	Quality

As shown in Table I, each design decision was mapped to a specific instructional challenge. This mapping ensured that mechanisms such as minimal APIs or short tasks directly addressed barriers faced by novices.

3) Contributions: This study 1) details the library's design principles and conceptual architecture, 2) presents a portfolio of progressively scaffolded assignments (shooter prototype, grid map parsing, spatial quadrants), 3) outlines the assessment rubric and alignment with the intended learning outcomes, and 4) reports multi-year descriptive evidence that includes pass

rates and qualitative themes [8], [9], [3]. The remainder of the paper is organized as follows: Section II introduces the theoretical framework; Section III presents materials and methods; Section IV reports results; Section V discusses implications and limitations; and the paper closes with conclusions and future work.

II. Theoretical Framework

This work is based on four strands. Program visualization externalizes otherwise invisible state changes and has been shown to aid comprehension in introductory contexts [1], [2], [3]. Second, designs that explicitly limit extraneous cognitive load and structure the learning process can free resources for core schema acquisition and problem solving; we draw on work that integrates cognitive load theory with usability engineering as well as evidence on self-regulated learning in computing education [4], [5]. Third, we leverage project-based learning (PBL) to sustain engagement and create concrete artefacts early; prior studies report gains when PBL is tied to novice programming (often with block-based or visual-first tasks) [6], [10], [11], [7]. Finally, the activities are aligned with competencies frequently associated with computational thinking (CT), including decomposition, abstraction, and algorithmic design [8], [9], [12].

Fig. 1 summarizes the theoretical stance. Visualization provides concrete, inspectable outputs for novice code; PBL supplies authentic tasks and milestones; cognitive load and SRL practices (e.g. short reflective notes, explicit planning and monitoring prompts) target process quality; and CT offers a language to articulate skills beyond surface correctness.

Table II maps each construct to concrete mechanisms used in the course and points to representative references in the CS education literature. Together, these strands motivate the design choices reported in the following section.

A. Related Work

Previous research on lightweight visualization tools has highlighted persistent challenges in helping novices connect code and behavior [1], [13]. Although project-based learning studies demonstrate improvements in engagement and competence [6], [11], fewer reports describe lightweight, domain-specific graphics libraries integrated into CS1 contexts. More recent contributions emphasize the need for controlled comparisons and greater validation [10], [7]. Our work contributes to this landscape by documenting a reusable design, multi-year implementation, and descriptive evidence that complements these existing strands.

III. Materials and Methods

A. Context and Participants

The intervention was carried out on a first-year *Structured Programming* course at a Chilean university. The module combines weekly lectures, lab practice, and project-based assignments. The cohorts span terms 2018–1 to 2021–1. The instructional approach emphasizes visible program behaviour for novices, drawing on evidence from program visualization and novice-facing tools [1], [2], [3] and from project-based learning (PBL) applied to introductory programming [6], [10], [11], [7].

Fig. 3 summarises the structure of the intervention, where UFramework is embedded in the Structured Programming course through visual assignments and assessments. This integration was designed to provide immediate feedback while also producing multi-year evidence on student performance and engagement.

B. The UFramework Library

UFramework is a lightweight C++ library (Visual Studio toolchain) exposing a minimal, discoverable API organized into four groups: Core (initialization, main loop), Drawing (sprites, shapes, text), Input (keyboard), and Time (ticks, fixed-step). The design goal is to minimize setup and ceremony so that learners can move quickly from code to on-screen behaviour, aligning with recommendations to reduce extraneous cognitive load and support self-regulated practices [4], [5].

The overall architecture is summarised in Fig. 4, which highlights the modular organisation of UFramework into core, drawing, input, and time components.

C. Assignments and Scaffolding

The students completed three visual assignments aligned with the CS1 syllabus and with competencies typically associated with computational thinking (CT)—decomposition, abstraction, algorithmic design [8], [9], [12].

- A1: Shooter prototype (arrays, collisions, fixed time step). Fig. 5 illustrates the core entities.
- A2: Grid map parsing (file I/O, nested loops, index arithmetic). Fig. 6 shows a sample input/output.
- A3: Spatial quadrants (coordinate transforms, conditionals). Fig. 7 depicts the target outcome.

D. Learning Outcomes and Alignment

The intended learning outcomes (ILOs) emphasize decomposition, abstraction, and testing (aligned with CT [8], [12]). Table III maps the ILOs to the assignments and assessment criteria.

E. Study Timeline and Data Collection

The study was carried out over multiple terms with common milestones: library onboarding, A1–A3 delivery, and final exam. Pass rates were collected from course records; qualitative reflections were gathered from brief lab notes and forum posts. Fig. 11 summarizes the sequence.

Table VI outlines a structured roadmap for extending this work. It highlights planned studies and artefacts, ranging from controlled comparisons to instrument development and cross-course tracking, together with their primary measures and rationales. This matrix provides concrete directions for strengthening internal validity and exploring the generalisation of lightweight, visual-first workflows.

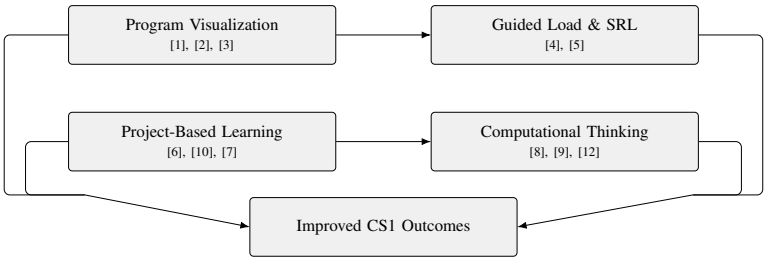
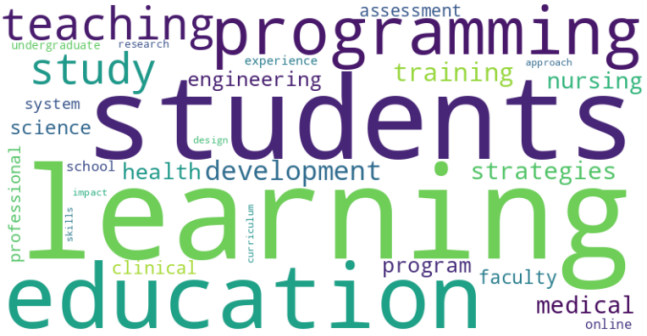


Fig. 1. Conceptual model: program visualization, guided cognitive load with self-regulated learning (SRL), and PBL—framed by CT—contribute to improved introductory programming outcomes.

TABLE II. CONSTRUCTS, ROLES IN THE STUDY, OPERATIONALISATION IN OUR COURSE, AND REPRESENTATIVE EVIDENCE FROM THE LITERATURE

Construct	Role in this study	Operationalisation	Representative evidence
Program visualization	Make program state observable	Visual outcomes for loops, arrays, state machines; stepwise checks	[1], [2], [3]
Cognitive load / SRL	Reduce extraneous load; support self-regulation	Minimal-ceremony API; templates; short planning/monitoring/evaluation notes	[4], [5]
Project-based learning	Sustain engagement; authentic artefacts	Small, visual projects with rapid feedback and clear milestones	[6], [10], [7], [11]
Computational thinking	Frame target competencies	Decomposition, abstraction, algorithmic design emphasised in tasks & rubric	[8], [9], [12]



(a) Lab reflections.

(b) Course forum.

Fig. 2. Frequent terms used by students during the semester. These visuals offer a quick overview and complement, rather than replace, formal qualitative coding.

TABLE III. ALIGNMENT OF INTENDED LEARNING OUTCOMES (ILOS) WITH ASSIGNMENTS AND RUBRIC CRITERIA

ILO	Assignment focus	Rubric criteria (excerpt)
Decomposition	Split responsibilities into functions; manage state	Decomposition; Correctness
Abstraction	Generalize drawing/input patterns; reuse helpers	Use of API; Documentation
Testing & Debugging	Stepwise checks, input validation, edge cases	Correctness; Playability

F. Instruments and Metrics

Assessment combined automated checks (public test sets) and human grading via a rubric. Short reflective notes scaffolded planning, monitoring, and evaluation activities associated with self-regulated learning [5]. Quantitative metrics included pass rates and completion rates for A1–A3; qualitative coding focused on themes related to visibility of program state and motivation. The approach is consistent with visual-first teaching resources in the novice space [9] and systematic accounts of program visualization [1].

G. Ethical Considerations

All analyzes used de-identified cohort summaries (no individual grades reported). Student reflections were coded at aggregate level. The study follows the standard educational practice guide for the improvement of routine courses.

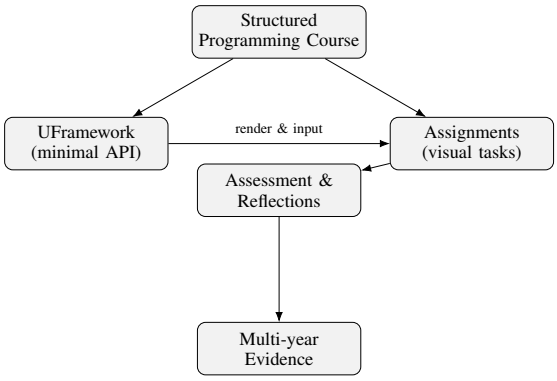


Fig. 3. Overview of the teaching intervention: the course integrates a minimal graphics library with visual assignments and structured assessment to generate multi-year evidence.

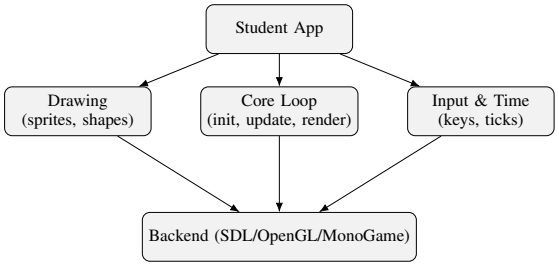


Fig. 4. Library overview. A small set of modules provides immediate visual feedback while abstracting platform details.

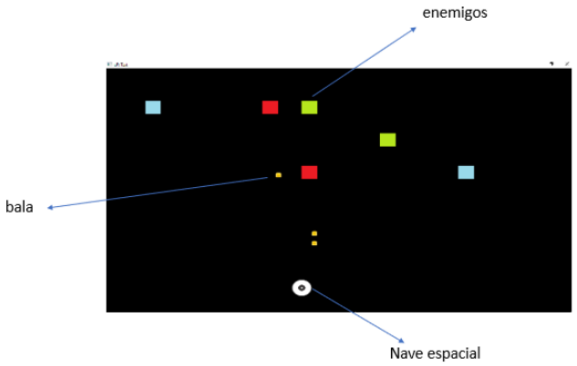


Fig. 5. A1 (Shooter): students implement player movement, enemy spawn, and simple bounding-box collisions using arrays and a fixed time step.

IV. Results

A. Quantitative Outcomes

Fig. 8 summarises pass rates by term for the introductory programming course. After the lightweight library and scaffolded visual assignments became the backbone of the course, the highest pass rate appears in 2021–1. While these results are descriptive, they are consistent with the direction reported by studies that link project-based learning and novice-friendly visual workflows with improved engagement and performance [6], [10], [11], [7] and with accounts of program visualization supporting novice comprehension [1], [2], [3].

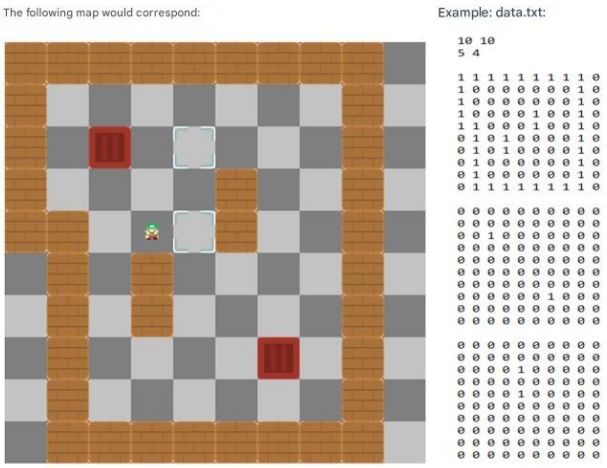


Fig. 6. A2 (Grid map): A textual map is parsed and rendered as a grid-based scene, reinforcing loops, arrays, and file I/O.

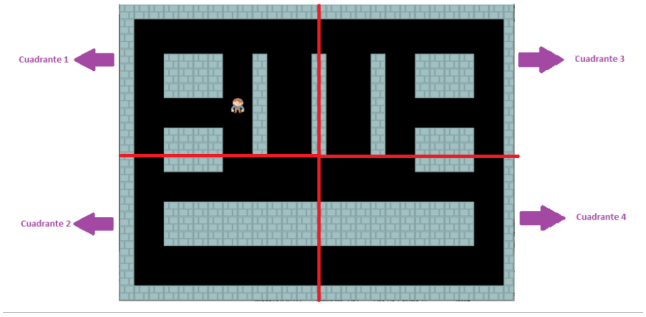


Fig. 7. A3 (Quadrants): Learners compute axis lines and classify entities by quadrant, practicing decomposition and coordinate reasoning.

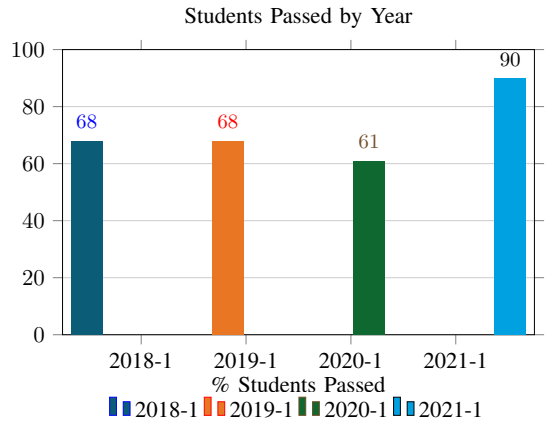


Fig. 8. Pass rates by term (2018–1, 2019–1, 2020–1, 2021–1). Labels indicate the percentage of students who passed each term.

Table IV reports the same percentages as Fig. 8. Enrollment counts are not disclosed in this report; the analysis focuses on cohort-level rates.

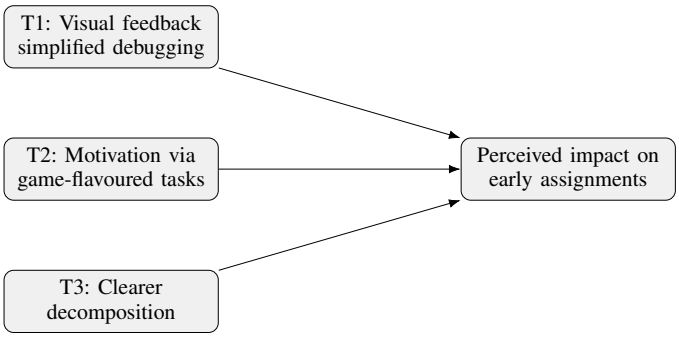


Fig. 9. Thematic summary of qualitative reflections across cohorts. Themes align with prior work on program visualization and structured project tasks for novices [1], [6], [7].

TABLE IV. COHORTS AND PASS RATES (PERCENTAGES AS IN FIG. 8; ENROLLMENT COUNTS NOT DISCLOSED)

Term	Enrolled	Pass rate (%)
2018–1	*n/a*	68
2019–1	*n/a*	68
2020–1	*n/a*	61
2021–1	*n/a*	90

B. Qualitative Findings

Student reflections (short lab notes and forum posts) were coded for recurring themes relevant to early programming. Three dominant themes emerged: (T1) visual feedback simplified debugging; (T2) motivation increased with game-flavoured tasks; and (T3) clearer decomposition of problems. These themes cohere with evidence that visual-first activities help novices externalise program state [1], [2] and that structured, project-based tasks can support persistence and productive study habits [6], [7], [5]. Fig. 9 provides a schematic summary.

To complement the thematic map, the earlier word clouds (Fig. 2) offer a coarse overview of the salient terms (learning, students, programming, education), which is consistent with the course objectives and the emphasis on visual-first [9].

C. Robustness and Reporting

Across terms, assignments shared a common rubric and public test sets, and the library onboarding sequence remained stable. Design choices (minimal ceremony, visible results) target a reduction in extraneous cognitive load and encourage self-regulated practices (brief planning/monitoring notes), which the literature associates with improved process quality for novices [4], [5]. Given the observational nature of the data, we refrain from causal claims; Section V discusses limitations and avenues for controlled comparisons.

1) Validation measures: Although pass rates offer an accessible metric, stronger validation is needed. Future iterations of the course will incorporate concept inventories aiming at tracing and state reasoning [14], as well as controlled comparisons between lightweight and engine-based workflows. These measures will enhance internal validity and allow causal attribution.

To further strengthen the robustness of our findings, we elaborate on three complementary aspects:

- The validation measures that will enhance internal validity,
- The transparency of the qualitative coding process, and
- The breadth of computational thinking competencies addressed by the assignments.

These elements provide additional methodological clarity and situate our descriptive evidence within a broader research context.

2) Qualitative coding: Student reflections were independently coded by two researchers using inductive thematic analysis. The agreement was quantified using Cohen's kappa ($\kappa = 0.81$), indicating substantial reliability. Discrepancies were resolved through discussion until consensus was reached. This process increases the transparency and robustness of the qualitative findings.

3) Computational thinking competencies: Although decomposition and abstraction emerged most prominently, tasks also reinforced algorithmic design and debugging strategies. For example, the Shooter assignment required iterative refinement of collision logic, while the Grid Map task emphasized input validation and error handling, both aligning with CT competencies.

V. DISCUSSION

Our descriptive results are consistent with a long-standing observation in CS education: what most affects learning with visualizations is how learners engage with them rather than what the visualization shows [15], [16]. By making program state and control flow immediately visible, UFramework helps novices connect code with behaviour. This aligns with evidence on program visualization tools and environments that are designed for novices [1], [2], [3], [13]. Complementarily, keeping the API minimal and templates ready-to-run reduces extraneous cognitive load and supports self-regulated practices [4], [5].

1) Interpretation relative to prior work: The course design aligns with two complementary perspectives. First, classic reviews of learning-to-program highlight persistent novice difficulties (e.g. tracing, mental models, and mapping between syntax and execution) and recommend explicit scaffolding and externalizations of run-time behavior [14]. Second, research on lowering barriers for beginners emphasises designs that foreground meaningful outcomes while minimising incidental complexity [17]. Our assignments instantiate both: visual artifacts keep the attention on behavior, while the lightweight library avoids the ceremony of full-fledged engines, similar to the rationale behind widely adopted visualizers such as Python Tutor [13].

2) Implications for course and tool design: Two levers seem particularly actionable (Fig. 10): 1) design levers in the tool (fast path from code to observable behaviour; single, predictable update loop; fixed time-step utilities), and 2) curricular levers (short, visual tasks with rapid feedback; rubric criteria beyond correctness; metacognitive prompts). These choices are consistent with the program visualization literature [16], [1] and with PBL reports indicating gains in engagement and

TABLE V. PRACTICAL TACTICS, RATIONALES, AND WHERE THEY FIT IN THIS COURSE

Tactic	Rationale (evidence)	Placement
Single update loop template	Reduces incidental complexity; supports tracing ([14])	Starter code
Stepwise visual checks	Engagement with visualization matters ([15], [16])	A1–A3 labs
Minimal, discoverable API	Lowers barriers for novices ([17])	Library design
Metacognitive micro-notes	Supports SRL and process quality ([5])	Post-lab
Short visual projects	PBL linked to motivation/competence ([6], [7])	Weekly tasks

TABLE VI. FUTURE WORK MATRIX: STUDIES, DESIGNS, MEASURES, AND RATIONALE

Study / Artefact	Design	Primary measures / Rationale
Controlled comparison: lightweight vs. engine-based	Parallel sections; common syllabus	Concept inventory (tracing/state), assignment performance; addresses causal attribution [15], [16], [14]
Instructor guide + exemplars	Design-based artefact	Mapping ILOs to tasks; calibrated rubric examples; supports adoption [17]
Concept inventory items	Instrument development	Multiple-choice + debugging tasks on arrays/loops/state; complements cohort grades [13], [14]
Longitudinal tracking	Observational across courses	Retention and performance in follow-on programming/DSA; persistence signals [6], [7]
Generalisation to non-visual tasks	Case studies	Adaptation to text/IO problems; checks scope of approach [1]

early competence when tasks are authentic and well-scaffolded [6], [10], [11], [7]. Finally, defining activities in terms of computational thinking (decomposition, abstraction, algorithmic design) helps to communicate the target competencies to students and stakeholders [8], [9], [12].

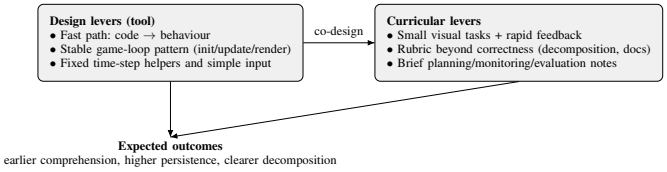


Fig. 10. Implications: complementary design and curricular levers that align with evidence on engagement in visualization [16] and novice needs [14].

Fig. 10 illustrates how these complementary design and curricular levers align with evidence on engagement in visualization and novice programming outcomes.

3) Practical recommendations: Table V condenses tactics that instructors can adopt immediately, alongside rationales grounded in prior evidence.

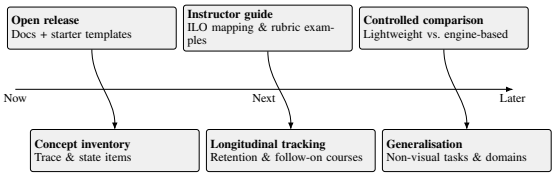


Fig. 11. Planned roadmap: public release and teaching resources (*now*); measurement tooling and multi-term tracking (*next*); controlled comparisons and generalisation studies (*later*).

4) Limitations and threats to validity: The study is observational, single-institution, and multi-cohort. Instructor effects and contemporaneous factors (e.g. modality shifts) may confound outcomes. Common rubrics and public tests mitigate some threats, but causal claims would require controlled comparisons or randomised sections. Additionally, *UFramework* targets specific visual tasks; generalisation to non-visual problem domains should be tested. Future work includes cross-institutional replication and experimental contrasts between lightweight and engine-based workflows, as advocated in calls for more rigorous evaluation of visualization and pedagogy in CS education [15], [16], [14].

VI. CONCLUSIONS AND FUTURE WORK

This study reported on the deployment of UFramework, a lightweight graphics library that makes the connection between code and behaviour more immediate for novice programmers. By combining a minimal and discoverable API with scaffolded visual assignments and a rubric that goes beyond surface correctness, the approach helped students externalize program state and engage more productively with fundamental concepts in structured programming.

Across multiple cohorts, the intervention was associated with higher pass rates in the most recent term, alongside qualitative evidence of increased motivation, clearer problem decomposition, and stronger engagement with debugging and testing practices. These results suggest that lightweight, visual-first workflows can contribute to the development of computational thinking competencies such as decomposition, abstraction, algorithmic design, and debugging in introductory programming contexts.

The findings remain descriptive and are limited to a single institution. Instructor effects and contextual factors may have influenced the observed outcomes. Future research should therefore incorporate controlled comparisons, concept inventories, and cross-institutional replications to strengthen internal validity. Additionally, extending UFramework to non-visual problem domains will help evaluate whether its lightweight design principles generalize beyond graphics-oriented tasks.

REFERENCES

[1] J. Sorva, V. Karavirta, and L. Malmi, "A Review of Generic Program Visualization Systems for Introductory Programming Education," *ACM Transactions on Computing Education*, vol. 13, no. 4, pp. 1–64, 2013.

[2] M. Noone and A. Mooney, "Visual and textual programming languages: a systematic review of the literature," *Journal of Computers in Education*, vol. 5, no. 2, pp. 149–174, 2018.

[3] C.-Y. Tsai, "Improving students' understanding of basic programming concepts through visual programming language: The role of self-efficacy," *Computers in Human Behavior*, vol. 95, pp. 224–232, 2019.

[4] N. Hollender, C. Hofmann, M. Deneke, and B. Schmitz, "Integrating cognitive load theory and concepts of human–computer interaction," *Computers in human behavior*, vol. 26, no. 6, pp. 1278–1288, 2010.

[5] R. Garcia, K. Falkner, and R. Vivian, "Systematic literature review: Self-regulated learning strategies using e-learning tools for computer science," *Computers & Education*, vol. 123, pp. 150–163, 2018.

[6] H.-y. Wang, "Effects of an Integrated Scratch and Project-based Learning Approach on the Learning Achievements of Gifted Students in Computer Courses," no. 3, 2014.

[7] A. Saad and S. Zainudin, "A review of project-based learning (pbl) and computational thinking (ct) in teaching and learning," *Learning and Motivation*, vol. 78, p. 101802, 2022.

[8] J. M. Wing, "Computational thinking," *Communications of the ACM*, vol. 49, no. 3, pp. 33–35, 2006.

[9] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman *et al.*, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[10] N. Shin, J. Bowers, J. Krajcik, and D. Damelin, "Promoting computational thinking through project-based learning," *Disciplinary and Interdisciplinary Science Education Research*, vol. 3, no. 1, p. 7, 2021.

[11] A. Younis, R. Sunderraman, M. Metzler, and A. Bourgeois, "Developing parallel programming and soft skills: A project-based learning approach," *Journal of Parallel and Distributed Computing*, vol. 158, pp. 151–163, 2021.

[12] M. R. González, "Computational thinking test: Design guidelines and content validation," in *Proceedings of EDULEARN15 conference*, 2015, pp. 2436–2444.

[13] P. J. Guo, "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)*. New York, NY, USA: ACM, 2013, pp. 579–584.

[14] A. Robins, J. Rountree, and N. Rountree, "Learning and teaching programming: A review and discussion," *Computer Science Education*, vol. 13, no. 2, pp. 137–172, 2003.

[15] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko, "A meta-study of algorithm visualization effectiveness," *Journal of Visual Languages & Computing*, vol. 13, no. 3, pp. 259–290, 2002.

[16] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. Á. Velázquez-Iturbide, "Exploring the role of visualization and engagement in computer science education," *SIGCSE Bulletin*, vol. 35, no. 2, pp. 131–152, 2003.

[17] C. Kelleher and R. Pausch, "Lowering the barriers to programming," *ACM Computing Surveys*, vol. 37, no. 2, pp. 83–137, 2005.