SCIENTIFIC CULTURE

www.sci-cult.com

SCIENTIFIC CULTURE, Vol. 12, No. 2.1, (2026), pp. 2716-2726
Open Access. Online & Print

DOI: 10.5281/zenodo.122.126209

BRIDGING THE LOGICAL GAP: VISUAL SCRIPTING AS A SCAFFOLDING STRATEGY FOR PHYSICS-BASED GAME DEVELOPMENT IN ENGINEERING EDUCATION

Álvaro Villagómez-Palacios[1], Claudia De la Fuente-Burdiles[2] and Cristian Vidal-Silva[3*]

[1]Facultad de Ciencias e Ingenierías Universidad Estatal de Milagro Milagro, Ecuador
Email: avillagomezp@unemi.edu.ec
[2]Department of Interactive Visualization and Virtual Reality, Faculty of Engineering University of Talca
Talca, Chile. Email: claudia.delafuente@utalca.cl
[3]Department of Interactive Visualization and Virtual Reality, Faculty of Engineering University of Talca
Talca, Chile. Email: cvidal@utalca.cl

Received: 20/10/2025
Accepted: 01/12/2025

Corresponding Author: Cristian Vidal-Silva
(cvidal@utalca.cl)

ABSTRACT

The acquisition of computational thinking and programming proficiency represents a significant bottleneck for novice engineering students. Traditional syntax-heavy languages often impose a steep cognitive load that hinders the understanding of core algorithmic concepts. This study explores the efficacy of node-based visual programming environments, specifically Unity Visual Scripting, as a pedagogical scaffolding mechanism. By shifting the instructional focus from syntax memorization to visual logic flows, we aimed to enhance student engagement and technical competency. Utilizing a quasi-experimental design with a within-subjects approach ($N=22$), we analyzed development efficiency, academic performance, and code complexity. Results demonstrate that the visual paradigm facilitated a smoother transition to complex logic implementation, reducing development time by 30-50% for core mechanics and significantly improving student confidence. Crucially, participants identified Visual Scripting not as a replacement, but as a cognitive bridge that facilitates the understanding of algorithmic logic before tackling syntactic complexities.

KEYWORDS: Unity Visual Scripting, Instructional Scaffolding, Computational Thinking, Engineering Education, Node-Based Programming.

1. INTRODUCTION

The development of algorithmic and programming competencies is widely regarded as essential for modern scientific thinking and computational literacy [1]. As society transitions towards Industry 4.0, programming has evolved from a niche skill into a transversal competence required across disciplines, enabling the construction of applications ranging from data-intensive systems to simulations and videogames [2], [3]. In this context, identifying the components of "Education 4.0" becomes critical to align academic training with 21st-century skills frameworks [4]. Throughout this article we adopt the following terminology for clarity. We use the term block-based programming to refer to environments that represent instructions as interlocking blocks in a two-dimensional workspace (e.g., Scratch, Tinkercad, Alice), which are mostly used in K–12 settings. We reserve visual programming as a broader umbrella concept that includes both block-based tools and other graphical notations for specifying program behaviour. When referring specifically to tools integrated into professional game engines, such as Unity Visual Scripting or Unreal Blueprints, we use the term visual scripting and treat node-based scripting as a synonym, since programs are constructed by connecting typed nodes instead of writing textual code. In the remainder of the paper we consistently use block-based programming for K–12 tools and visual scripting for the Unity-based intervention studied here.

Novices must simultaneously wrestle with abstract logical structures (loops, conditionals) and rigid syntactical rules. This dual burden, often referred to as 'intrinsic' and 'extraneous' cognitive load, leads to high attrition rates and a superficial understanding of algorithmic logic. However, in educational contexts, specifically in engineering and computer science, introductory programming courses often act as "gatekeepers." These courses historically suffer from high dropout rates and low student motivation due to the steep learning curve associated with traditional syntax-heavy languages like C++, Java, or C# [5]. Novices are often overwhelmed by what Sim and Lau [6] describe as the "double burden": the simultaneous need to master abstract problem-solving logic (semantics) and strict language rules (syntax). Even when a conceptual task is relatively simple, such as determining whether a person is of legal age, translating that task into syntactically correct C# code requires knowledge of data types, memory management, and compiler error handling that novices do not yet possess.

This "syntactic barrier" disconnects the student's mental model from the computational implementation. As illustrated in Figure 1, while the algorithmic flow is intuitive, the textual implementation introduces extraneous cognitive load. To address this, block-based programming environments have been successfully deployed in K-12 education to foster computational thinking without the frustration of syntax errors [8], [9]. Yet, in higher education, there is a reluctance to adopt these tools, often perceiving them as insufficiently professional. The emergence of Unity Visual Scripting (formerly Bolt) challenges this perception by offering a professional-grade, node-based environment integrated directly into a market-leading game engine [10]. This article presents a quasi-experimental comparative study to determine if professional Visual Scripting tools can act as efficient accelerators in university curricula. Unlike previous studies that focus solely on satisfaction, we analyze development efficiency (time-to-prototype) and structural quality, hypothesizing that visual paradigms can bridge the gap between novice intuition and professional engineering practices.

The remainder of this article is structured as follows. Section 2 reviews the state-of-the-art regarding visual programming barriers, the use of game engines in academia, and current assessment models. Section 3 establishes the theoretical framework that guides this study, integrating Cognitive Load Theory, Flow Theory, and the Technology Acceptance Model. Section 4 details the quasi-experimental design, including participant demographics and the twophase instructional intervention (MonoGame vs. Unity). Section 5 presents the empirical findings, highlighting the comparative analysis of development efficiency and a qualitative taxonomy of errors. Section 6 discusses the pedagogical trade-offs and interprets the role of Visual Scripting as a didactic bridge. Finally, Section 7 summarizes the main contributions, acknowledges limitations, and outlines directions for future research.

2. RELATED WORK

The intersection of game development, visual programming, and engineering education has been a fertile ground for research. This section reviews key contributions regarding learning barriers, the use of hardware/software visual analogies, and game-based assessment.

2.1. Learning Barriers and Novice Programming

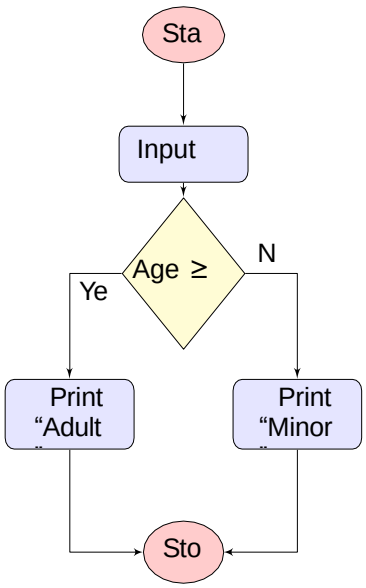
Visual Programming Languages (VPLs) have long been advocated to lower the barrier to entry for novices. Ko et al. [7] identified early on that syntax errors often discourage learners before they grasp algorithmic logic. Recent systematic reviews by Sim and Lau [6] reaffirm that syntax remains the primary source of frustration for beginners. Sun et al. [11] conducted a comparative study showing that block-based environments significantly reduce the frequency of compilation errors, allowing students to focus on "computational action." However, the transition from blocks to text remains complex. Wo¨rister and Knobelsdorf [?] argue that block-based programming facilitates the understanding of low-level computing concepts, suggesting that visual abstractions are valid even for complex engineering topics like Assembly, provided there is a clear mapping to the underlying logic.

2.2. Visual Metaphors in Robotics and STEM

The efficacy of visual programming is not limited to pure software; it has been extensively validated in physical computing. Vidal-Silva et al. [8], [12] demonstrated that block-based tools (e.g., Scratch, Tinkercad) significantly improve programming competencies in school students across Latin America by allowing them to control Arduino hardware visually. Similarly, Rojas-Valde´s et al. [9] found that removing syntactic friction allows students to solve complex hardware control problems more effectively. Tramonti et al. [13] further emphasized that design thinking combined with visual tools enhances problem-solving in educational robotics. These findings in the hardware domain support our hypothesis that visual scripting in Unity, which controls "virtual hardware" (game physics), can yield similar educational benefits.

2.3. Game Engines and Serious Games in Academia

The adoption of professional game engines in academia has shifted the focus from building engines to building with engines. Hussain et al. [14] provide a technical survey of Unity, emphasizing its versatility for both 2D and 3D development. Recent work by Maraffi [15] introduces "Level-Up Logics," leveraging game design platforms to teach coding fundamentals. This aligns with the notion of Serious Games as assessment tools; Gomez et al. [16] conducted a systematic review concluding that game-based assessment can capture competencies that traditional tests miss. Furthermore, Maxim and Arnedo-Moreno [17] identify key principles in serious game design, suggesting that the development tool itself (visual vs. text) influences the quality of the final learning artifact. Our study contributes to this body of knowledge by quantifying the efficiency gains of visual tools in this specific context.



a) ***Algorithm (Mental Model).***

Figure 1: The "Cognitive Gap" In Introductory Programming. While The Flowchart (A) Is Intuitive, The Text Implementation
(B) Introduces Extraneous Load Via Syntax Rules (Braces, Semicolons, Pointers), As Identified By Ko Et Al. [7].

```
include<stdio.h>
int main() {
int age;
printf("Enter age: ");
// SYNTAX BARRIER:
// Format specifiers, addresses (&)
scanf("%d", &age);
if (age >= 18){
printf("Adult \n");
}else{
printf("Minor \n");
a)    Algorithm (Mental Model)}
//  SEMANTIC BARRIER :
//  Return  types, scope
return 0;}
```

3. THEORETICAL FRAMEWORK

3.1. Cognitive Load Theory in Programming

Cognitive Load Theory (CLT) posits that working memory is limited. In the context of learning

programming, the load can be categorized as:

- Intrinsic Load: The inherent difficulty of the algorithm (e.g., understanding a nested loop).
- Extraneous Load: The effort required to deal with the instructional material or environment (e.g., syntax errors, IDE configuration).
- Germane Load: The effort dedicated to creating permanent schemas (learning).

Visual environments are designed specifically to minimize extraneous load by preventing syntax errors entirely; two nodes simply cannot be connected if their data types are incompatible. Text-based programming often imposes a high Extraneous Load due to syntax. Visual Scripting reduces this load by preventing syntax errors by construction, potentially freeing up cognitive resources for the Intrinsic and Germane load (logic and physics concepts) [18], [19].

Figure 2 illustrates the theoretical redistribution of cognitive resources observed in this study. According to Sweller's framework, the total cognitive capacity of a novice student is limited.

In the Text-Based approach (left bar), a significant portion of this capacity is consumed by Extraneous Load (represented in red). This corresponds to the mental effort required to process syntax rules, case sensitivity, and compiler error codes, elements that are not central to the logical problem but are necessary for the code to run. Consequently, the available capacity for Germane Load (green), the processing space used for actual learning and schema construction, is compressed. In contrast, the Visual Scripting approach (right bar) drastically reduces the Extraneous Load by eliminating syntax errors through a "drag-and-drop" interface that prevents invalid connections. Since the Intrinsic Load (blue), the inherent difficulty of the game logic, remains constant across both paradigms, the reduction in extraneous noise directly liberates cognitive resources. This surplus capacity is shifted towards Germane Load, allowing students to focus deeper on the causal relationships of the physics simulation rather than debugging missing semicolons.

3.2. Flow Theory and Engagement in Coding

Beyond cognitive load, the emotional state of the learner plays a crucial role. Csikszentmihalyi's Flow Theory suggests that optimal learning occurs when the challenge level matches the student's skill. In text-based programming, syntax errors often interrupt this flow, causing frustration. Ke et al. [20] and Tsai et al. [21] have explored engagement in game-based learning, finding that visual feedback loops help maintain the "Flow" state. By removing syntax errors, Visual Scripting allows students to stay in the flow of logic construction, potentially increasing persistence as observed by Israel-Fishelson and Hershkovitz [22].

To understand the students' preference for Visual Scripting, we draw upon the Technology Acceptance Model (TAM) proposed by Davis [23]. TAM posits that Perceived Usefulness (PU) and Perceived Ease of Use (PEOU) determine the intention to use a technology. In our context, while C# might have high PU (industry standard), its low PEOU for novices hampers adoption. Visual Scripting balances this by offering high PEOU while maintaining PU through its integration into a professional engine like Unity.

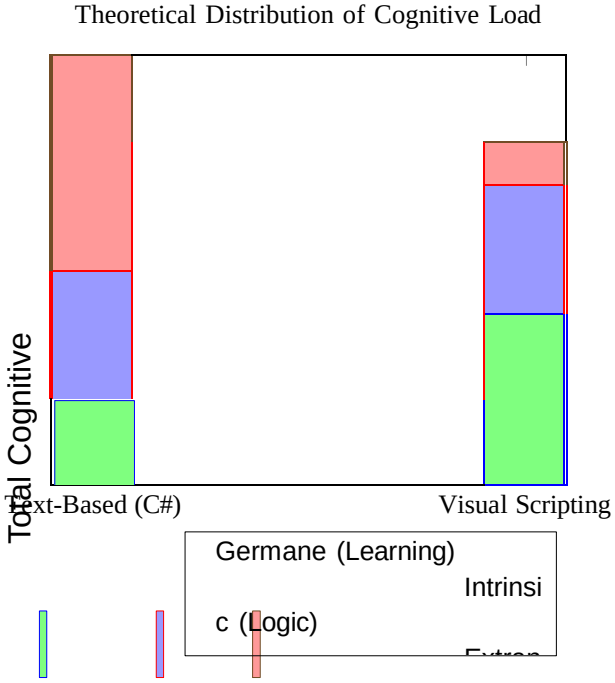


Figure 2: Comparison Of Cognitive Load Distribution. The Reduction Of Extraneous Load (Syntax) In Visual Scripting Frees Up Cognitive Capacity For Germane Load (Deep Learning), Assuming Constant Intrinsic Load (Logic).

3.3. Visual Scripting In Professional Engines

Unlike educational tools like Scratch (Figure 3), Unity Visual Scripting (Figure 5) operates on the engine's actual API. This means students are manipulating real game objects, physics bodies, and rendering pipelines, but through a node-based interface.

Unlike educational tools like Scratch (Figure 3), Unity Visual Scripting (Figure 5) operates on the engine's actual API. This means students are manipulating real game objects, physics bodies, and rendering pipelines, but through a node-based interface.

Figure 3 Block-based example in Scratch. Useful for K-12 but limited for professional simulation.

Figure 3: Block-based example in Scratch.

4. METHODOLOGY

4.1. Course Context, Participants and DemographIcs

The study was conducted during the "Physics for Videogames" course (3rd semester) at the University of Talca. The participants (N = 22) were engineering students with basic prior knowledge of Java/C but no experience in Unity Visual Scripting. The demographic distribution was 77% male and 23% female, with an age range of 19-22 years. Prior to the course, 100% of students had taken "Introduction to Programming" (C/Java), but a diagnostic survey revealed that only 15% felt "confident" with ObjectOriented Programming (OOP) concepts.

4.2. Instructional Design and Tasks

The intervention was structured to isolate the variable of syntax. By implementing the exact same algorithmic challenges (Snake game mechanics) in both paradigms, we could measure the 'friction' caused by text-based coding versus the 'flow' enabled by visual scaffolding.

The intervention was structured over 8 weeks:

- Weeks 1-4 (MonoGame/C#): Students implemented a Snake game from scratch. This involved creating a GameLoop, handling SpriteBatch rendering, and manually calculating AABB (Axis-Aligned Bounding Box) collisions.
- Weeks 5-8 (Unity Visual Scripting): Students recreated the game with added complexity (obstacles, particle effects). Instead of writing code, they used the Bolt/Visual Scripting graph to manipulate Rigidbody2D and BoxCollider2D components.

This structure ensures that students faced the exact same algorithmic challenges (movement vectors, collision response) in both paradigms.

4.3. Experimental Design

We adopted a Within-Subjects Quasi-Experimental We adopted a Within-Subjects Quasi-Experimental

TABLE 2: Selected Post-Survey Results (Scale 1–10).

Statement Mean

Design. The semester was divided into two phases to force a comparison within the same group of learners.

4.3.1. 4.3. Experimental Design

We adopted a Within-Subjects Quasi-Experimental

TABLE 2: Selected Post-Survey Results (Scale 1–10)

Statement Mean

Design. The semester was divided into two phases to force a comparison within the same group of learners.

4.3.1. Phase 1: Text-Based Approach (MonoGame). Students developed a classic *Snake* game. This framework requires writing the game loop, handling inputs, and calculating collisions manually in C# code.

"Visual Scripting helped me understand

programming logic."

"It is useful for learning fundamentals before moving to text."

8.9

8.8

Listing 1: Snippet of C# collision logic required in Phase 1.

```
i f ( head . P o s i t i o n . X == food . P o s i t i o n . X && head . P o s i t i o n . Y == food . P o s i t i o n . Y) { GrowSnake ( ) ;
SpawnFood ( ) ;
}
```

*4.3.2. Phase 2: Visual Approach (Unity). Students developed an enhanced *Snake* with physics obstacles. They used Unity Visual Scripting graphs.*

4.4. Results

4.4.1. Development Efficiency Analysis

Instructors tracked the time taken to complete specific milestones in both phases. The differences were substantial. As seen in Figure 6, the most significant gain was in Collision Logic (50 min vs 30 min). In MonoGame, students had to debug mathematical bounding box errors. In Unity, they simply connected an 'OnCollisionEnter' node, allowing them to focus on the consequence of the collision rather than its detection.

4.5. Academic Performance

Table 1 details the grade distribution. While the mean increase is modest (6.15 to 6.46), the Standard Deviation decreased, indicating that fewer students were "left behind" when using visual tools.

Table 1: Detailed Comparison Of Project Grades (Scale 1.0–7.0).

Statistic	Phase 1 (C#)	Phase 2 (Visual)
Mean Grade	6.15	6.46 (**+5%**)
Std. Deviation	0.55	0.49
Min Grade	5.0	5.5
Max Grade	6.9	6.9

4.5.1. Student Perception and Self-Efficacy

The post-intervention survey (Table 2) revealed high satisfaction. Crucially, students did not view Visual Scripting as a "toy," but as a legitimate learning tool.

Table 3 presents a qualitative taxonomy of errors observed during laboratory sessions. The contrast is striking. In the MonoGame phase, instructors spent the majority of their time helping students fix "Syntactic" errors, issues that prevent compilation but have no bearing on the game's logic.

This confirms the observations of Sim and Lau [6] regarding novice frustration. In contrast, the Visual Scripting phase shifted the error distribution almost entirely to "Logic/Physics". While students still made mistakes (e.g., calculating the wrong vector direction), these were *productive failures*. The visual environment allowed them to "see" the error in the graph's data flow (Fig. 5), transforming the debugging process from a code-inspection task into a logic-inspection task. This shift is critical for developing Computational Thinking as defined by Wing [1].

4.6. Discussion

4.6.1. Cognitive Mediation

A core contribution of this study concerns the idea of visual environments as cognitive mediators. Far from being reduced versions of programming, they create an intermediate representational layer. Students can validate their intuition about control flow by directly observing the execution path on screen. The results support the hypothesis that Visual Scripting is significantly more efficient for prototyping (Figure 6).

By abstracting the "boilerplate" code, students entered a state of "Flow" more easily. However, some advanced students noted in open-ended comments that for complex mathematical algorithms, visual graphs could become "spaghetti code," confirming that text is superior for density, while visual is superior for architecture and events.

4.6.2. The "Didactic Bridge" Effect

The high score (8.8) on the bridging question suggests that Visual Scripting acts as a scaffold. Seeing the logic laid out spatially helps build the mental model required for text-based coding. This challenges the notion that visual languages hinder the transition to "real" coding; instead, they appear to prepare the cognitive ground for it.

Figure 7 conceptualizes the role of Visual Scripting in the curriculum. Previous research often treats Visual Scripting as a distinct, alternative path. However, our findings align with Vinueza et al. [24], suggesting it is a transitional stage.

- Stage 1 (K-12): Tools like Scratch maximize abstraction to engage children [8].
- Stage 2 (The Bridge): Unity Visual Scripting

exposes the Game Object Component model and professional.

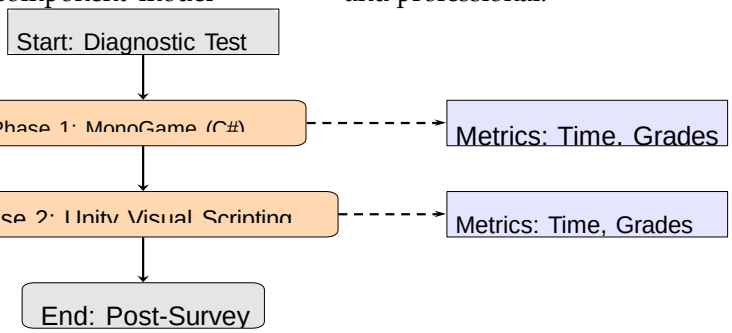
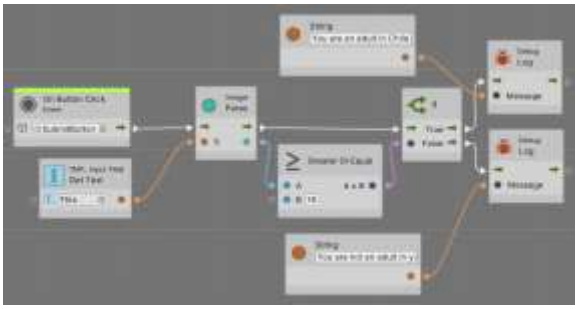


Figure 4: Methodological Workflow Of The Study. All Students Participated In Both Phases.

Figure 5: Unity Visual Scripting solution. Nodes represent events and data flow, abstracting the syntax.



API structure (Logic) but retains the visual safety net (Visual Syntax).

- Stage 3 (Professional): Once the API structure is understood visually, the student transitions to C# solely for fine-grained control and optimization.

This model explains why students reported high "Perceived Usefulness" in our survey; they intuitively understood they were learning professional concepts without the syntactic penalty

.Implications for Engineering Curricula

The efficiency gains observed in this study suggest that Visual Scripting should not be viewed merely as a prototyping tool, but as a strategic pedagogical scaffold. CaeiroRodr´ıguez et al. [25] emphasize the need for teaching soft skills and adaptability in engineering; by mastering Visual Scripting, students learn "systemic thinking" before getting bogged down in "syntactic details."

Furthermore, as highlighted by Fahmideh et al. [26] in the context of IoT and Software Engineering, the industry.

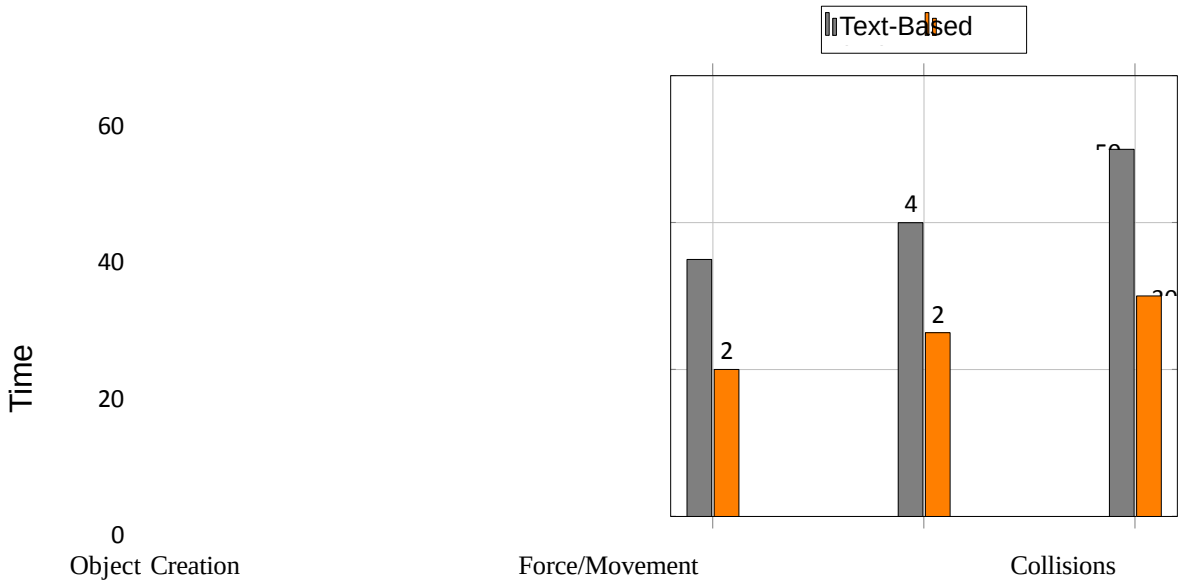


Figure 6: Average Development Time per Task. Visual Scripting Provided A 30-40% Reduction In Time-To-Prototype.

Table 3: Taxonomy of Frequent Errors: Text-Based vs. Visual Approach

is moving towards higher levels of abstraction. Integrating tools like Unity Visual Scripting aligns

with Industry 4.0 trends [4], where the ability to rapidly configure and connect logic blocks is becoming as valuable as writing raw code. However, we agree with Perera et al. [27] that textbased languages remain fundamental. Therefore, we propose a "Hybrid Syllabus" for introductory game development courses: start with Visual Scripting to build confidence and mental models (Weeks 1-6), and transition to C# scripting for performance optimization and complex architecture (Weeks 7-16).

5. CONCLUSION

This study demonstrates that Unity Visual Scripting is not merely an accessibility tool for non-programmers, but an efficient pedagogical accelerator for engineering students. The empirical results show a statistically significant reduction in development time (30–40%) for core mechanics compared to the text-based MonoGame approach, without compromising academic grades. Crucially, students reported higher self-efficacy and lower frustration, identifying Visual Scripting as a "cognitive bridge" that allows them to visualize abstract logic before committing to syntax.

5.1. Implications For Education

We recommend a "Hybrid Scaffolded Curriculum" for introductory game development and engineering courses. Educators should leverage Visual Scripting in the early stages to teach high-level concepts, such as Game Loops, Event-Driven Programming, and Physics Interactions, allowing students to build mental models free from syntactic noise. Once these models are solidified, the curriculum should transition to C #, defining it as a tool for optimization, architectural refinement, and granular control. This approach "Logic-First, Syntax-Second," aligns with the requirements of Industry 4.0 [4], prioritizing systemic thinking over rote memorization.

From a curriculum and policy perspective, this hybrid model has concrete implications for program design. At the institutional level, integrating visual scripting as an explicit component of early-semester courses requires aligning syllabi, learning outcomes, and assessment rubrics with the idea of a gradual transition from graphical to textual representations. This may involve, for example, dedicating

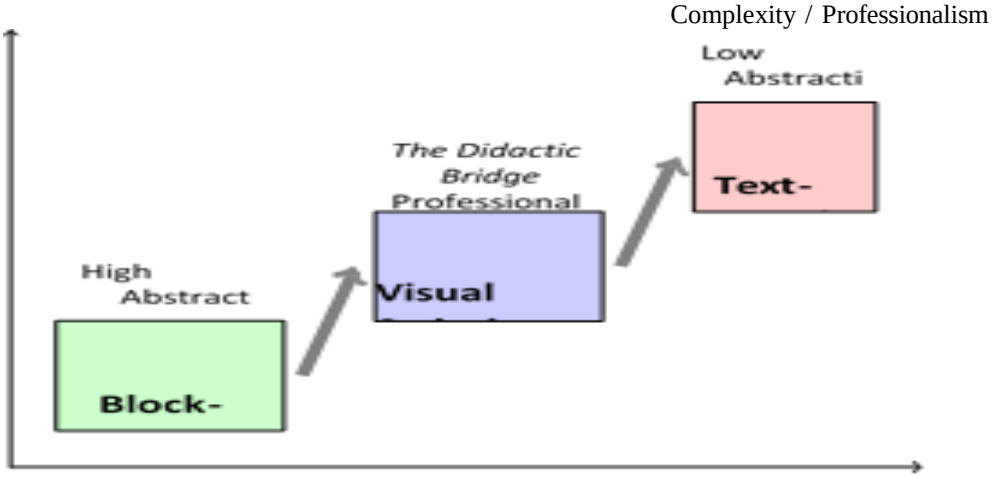


Figure 7: The Proposed "Pedagogical Bridge Model". Visual Scripting Serves As The Necessary Scaffold Between K-12 Block Tools And Professional Text Coding, Aligning With Vinueza Et Al. [24].

specific modules to Unity Visual Scripting in courses such as "Introduction to Programming" or "Game Development I", and mapping those modules to accreditation criteria related to computational thinking, design of interactive systems, and teamwork. At the faculty level, the adoption of visual tools calls for targeted professional development so that lecturers can design tasks that go beyond simple drag-and-drop exercises and connect the graphs with the underlying C# constructs. Finally, on a broader policy scale, the results support curricular guidelines that recognize visual environments as legitimate vehicles for engineering education, rather than remedial tools, which is particularly relevant for institutions seeking to widen participation and reduce early attrition in programming-heavy degrees.

5.2. Limitations

The study has limitations inherent to its quasiexperimental design. First, the sample size is modest (N = 22) and restricted to a single course at a Chilean university, which constrains the statistical power of the analyzes and limits the extent to which the findings can be generalized to other institutions, disciplines, or educational systems. The cohort also

corresponds to students who had already passed an introductory programming course and enrolled in a videogame-oriented course; their attitudes and prior experiences may not be representative of more heterogeneous populations or of students with low interest in game development.

Second, the within-subject structure of the intervention introduces a potential learning effect: performance in the Unity Visual Scripting phase may partly reflect the consolidation of conceptual understanding acquired during the earlier MonoGame phase. Although the second project deliberately incorporated additional mechanics to increase task difficulty, this design cannot fully disentangle the impact of the tool from that of accumulated practice. Finally, all measures were collected within a single semester and context, so the study does not capture long-term retention or transfer to subsequent text-based courses. These constraints should be considered when interpreting the results and suggest caution when extrapolating them to different institutional realities.

This study confirms that Visual Scripting effectively reduces the barrier to entry for engineering students. Future work will investigate the long-term retention of these skills, but current data suggests that visual scripting offers an essential pathway that helps students cross the initial conceptual barrier and eventually engage more deeply with professional software development practices.

5.3. Future Work

Future research avenues are twofold. First, we aim to explore the integration of Artificial Intelligence assistants within visual environments, investigating how AI-driven suggestions might further reduce cognitive load in serious game design [28]. Second, drawing from software product line engineering, we plan to analyze how visual modules can be reused to manage variability in student projects, potentially automating the assessment of game mechanics [29], [30]. Longitudinal studies are needed to track whether the conceptual gains from Visual Scripting translate into better long-term performance in advanced text-based programming courses.

REFERENCES

A. Hussain, H. Shakeel, F. Hussain, N. Uddin, and T. Ghouri, "Unity game development engine: A technical survey," University of Sindh Journal of Information and Communication Technology, vol. 4, 10 2020.

A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in 2004 IEEE Symposium on Visual Languages Human Centric Computing, 2004, pp. 199–206.

Available: https://doi.org/10.1007/s11423-023-10328-8

Available: https://doi.org/10.1016/j.compedu.2016.03.011

B. Chathuranga, "A systematic mapping of introductory programming languages for novice learners," IEEE Access, vol. 9, pp. 88 121– 88 136, 2021.

C. Vidal-Silva, "Applying the block-based programming language alice for developing programming competencies in university students," IEEE Access, vol. 13, pp. 21 471–21 485, 2025.

C. Vidal-Silva, J. Ca´rdenas-Cobo, M. Tupac-Yupanqui, J. SerranoMalebra´n, and A. Sa´nchez Ortiz, "Developing programming competencies in school-students with block-based tools in chile, ecuador, and peru," IEEE Access, vol. 12, pp. 118 924–118 936, 2024.

C. Vidal-Silva, J. Serrano-Malebran, and F. Pereira, "Scratch and arduino for effectively developing programming and computingelectronic competences in primary school children," in 2019 38th International Conference of the Chilean Computer Science Society (SCCC), 2019, pp. 1–7.

C.-Y. Chen, "Effects of worked examples with explanation types and learner motivation on cognitive load and programming problemsolving performance," ACM Trans. Comput. Educ., vol. 25, no. 2,

D. Benavides, "Exploiting the enumeration of all feature model configurations: A new perspective with distributed computing," in Proceedings of the 20th International Systems and Software Product Line Conference, ser. SPLC '16. Beijing, China: ACM, 2016, pp. 74–78. [Online]. Available: http://doi.acm.org/10.1145/2934466.2934478

D. Benavides, S. Segura, and A. Ruiz-Corte´s, "Automated analysis of feature models 20 years later: A literature review," Journal Information Systems, vol. 35, no. 6, pp. 615–636, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.is.2010.01.001

D. Sun, C.-K. Looi, Y. Li, C. Zhu, C. Zhu, and M. Cheng, "Block-based versus text-based programming: a comparison of learners' programming behaviors, computational thinking skills and attitudes toward programming," Educational Technology Research and Development, vol. 72, no. 2, pp. 1067–1089, 2024. [Online].

F. D. Davis, "Perceived usefulness, perceived ease of use, and user acceptance of information technology," MIS Q., vol. 13, no. 3, pp. 319–340, Sep. 1989. [Online]. Available: http:

F. Ke, K. Xie, and Y. Xie, "Game-based learning engagement: A theoryand data-driven exploration," British Journal of Educational Technology, vol. 47, no. 6, pp. 1183–1201, 2016. [Online]. Available: https://doi.org/10.1111/bjet.12314

I. Mekterovic´, L. Brkic´, B. Milas˘inovic´, and M. Baranovic´, "Building a comprehensive automated programming assessment system," IEEE Access, vol. 8, pp. 81 154–81 172, 2020.

J. A. Galindo, M. Acher, J. M. Tirado, C. Vidal, B. Baudry, and

J. H. Berssanette and A. C. de Francisco, "Cognitive load theory in the context of teaching and learning computer programming: A systematic literature review," IEEE Transactions on Education, vol. 65, no. 3, pp. 440–449, 2022.

J. M. Wing, "Computational thinking," Commun. ACM, vol. 49, no. 3, p. 33–35, Mar. 2006. [Online]. Available: https://doi.org/10. 1145/1118178.1118215

J. Pe´rez, M. Castro, and G. Lo´pez, "Serious games and ai: Challenges and opportunities for computational social science," IEEE Access, vol. 11, pp. 62 051–62 061, 2023.

Jun. 2025. [Online]. Available: https://doi.org/10.1145/3732791

L. Castillo-Salvatierra, J. Ca´rdenas-Cobo, C. de la Fuente-Burdiles, and C. Vidal-Silva, "Programming competencies in university students through game development," Frontiers in Education, vol. 10, 2025. [Online]. Available: https://doi.org/10.3389/feduc.

L. I. Gonza´lez-Pe´rez and M. S. Ram´ırez-Montoya, "Components of education 4.0 in 21st century skills frameworks: systematic review," Sustainability, vol. 14, no. 3, p. 1493, 2022.

M. A. Kuhail, S. Farooq, R. Hammad, and M. Bahja, "Characterizing visual programming approaches for end-user developers: A systematic review," IEEE Access, vol. 9, pp. 14 181–14 202, 2021.

M. Caeiro-Rodr´ıguez, M. Manso-Va´zquez, F. A. Mikic-Fonte,

M. Fahmideh, A. Ahmad, A. Behnaz, J. Grundy, and W. Susilo, "Software engineering for internet of things: The practitioners' perspective," IEEE Transactions on Software Engineering, vol. 48, no. 8,

M. J. Gomez, J. A. Ruipe´rez-Valiente, and F. J. G. Clemente, "A systematic literature review of game-based assessment studies: Trends and challenges," IEEE Transactions on Learning Technologies, vol. 16, no. 4, pp. 500–515, 2023.

M. Llamas-Nistal, M. J. Ferna´ndez-Iglesias, H. Tsalapatas, O. Heidmann, C. V. De Carvalho, T. Jesmin, J. Terasmaa, and L. T. Sørensen, "Teaching soft skills in engineering education: An european perspective," IEEE Access, vol. 9, pp. 29 222–29 242, 2021.

M. Tramonti, A. M. Dochshanov, and A. S. Zhumabayeva, "Design thinking as an auxiliary tool for educational robotics classes," Applied Sciences, vol. 13, no. 2, 2023. [Online]. Available: https://www.mdpi.com/2076-3417/13/2/858

M. Vinueza-Morales, J. Ca´rdenas-Cobo, J. Cabezas-Quinto, and

M.-J. Tsai, L.-J. Huang, H.-T. Hou, C.-Y. Hsu, and G.-L. Chiou, "Visual behavior, flow and achievement in game-based learning," Computers & Education, vol. 98, pp. 115–129, 2016. [Online].

N. Wongta and J. Natwichai, "End-to-end data pipeline in games for real-time data analytics," in Advances in Internet, Data and Web Technologies, ser. Lecture Notes on Data Engineering and Communications Technologies, L. Barolli, J. Natwichai, and T. Enokido, Eds. Cham: Springer, 2021, vol. 65.

P. Perera, G. Tennakoon, S. Ahangama, R. Panditharathna, and

P. Rojas-Valde´s, C. Vidal-Silva, and C. d. L. Fuente, "Successful development of problem-solving and computing programming competences in children using arduino," in 2022 International Symposium on Measurement and Control in Robotics (ISMCR), 2022, pp. 1–6.

pp. 2857–2878, 2022.

R. I. Maxim and J. Arnedo-Moreno, "Identifying key principles and commonalities in digital serious game design frameworks: Scoping review," JMIR Serious Games, vol. 13, p. e54075, Mar 2025. [Online]. Available: https://games.jmir.org/2025/1/e54075

R. Israel-Fishelson and A. Hershkovitz, "Persistence in a gamebased learning environment: The case of elementary school students learning computational thinking," Journal of Educational Computing Research, vol. 58, no. 5, pp. 891–918, 2020. [Online]. Available: https://doi.org/10.1177/0735633119887187

T. Maraffi, "Level-up logics: Leveraging three game design platforms to teach coding," in ACM SIGGRAPH

2024 Educator's Forum, ser. SIGGRAPH '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10. 1145/3641235.3664434

T. Y. Sim and S. L. Lau, "Review on challenges and solutions in novice programming education," in 2022 IEEE International Conference on Computing (ICOCO), 2022, pp. 55–61. //dx.doi.org/10.2307/249008 2025.1585602