Bridging the Gap Between Text-Based and Visual Programming: A Comparative Study of Efficiency and Student Engagement in Game Development

Álvaro Villagómez-Palacios[1], Claudia De la Fuente-Burdiles[2], Cristian Vidal-Silva*[3]
Facultad de Ciencias e Ingenierías, Universidad Estatal de Milagro, Milagro, Ecuador[1]
Department of Interactive Visualization and Virtual Reality-Faculty of Engineering
University of Talca, Talca, Chile[2,3]

***Abstract*—The integration of Low-Code and No-Code (LCNC) tools in higher education challenges traditional text-based programming pedagogies. While visual environments are often relegated to K-12 education, their adoption in professional engines like Unity suggests a need to re-evaluate their role in engineering curricula. This study analyzes the effectiveness, development efficiency, and perceived utility of Unity Visual Scripting compared to traditional C# programming (MonoGame) within a "Physics for Videogames" undergraduate course. Employing a quasi-experimental design with a within-subjects approach ($N = 22$), students first developed a game using C#/MonoGame and subsequently a complex variant using Unity Visual Scripting. Metrics included development time for core mechanics, project grades, and pre/post surveys on self-efficacy. Results demonstrate a statistically significant reduction in development time (30–50% faster for core mechanics) using Visual Scripting. Furthermore, academic performance improved slightly, and students reported higher confidence levels. 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. Consequently, Visual Scripting serves as an efficient accelerator for prototyping and conceptual learning in higher education, fostering a "logic-first, syntax-second" approach.**

***Keywords*—Visual scripting; higher education; development efficiency; engineering curricula**

I. 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, the following terminology is adopted for clarity. The term *block-based programming* is used 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. *Visual programming* is reserved 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, the term *visual scripting* is employed, and *node-based scripting* is treated as a synonym, since programs are constructed by connecting typed nodes instead of writing textual code. Consequently, in the remainder of the paper, *block-based programming* is consistently used for K–12 tools, while *visual scripting* refers to the Unity-based intervention studied here.

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, introducing 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 [7], [8]. 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 [9].

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, this study analyzes *development efficiency* (time-to-prototype) and *structural quality*, hypothesizing that visual paradigms can bridge the gap between novice intuition and professional engineering practices. The main contributions of this study are:

- A quantitative analysis of development efficiency

*Corresponding author.

(time-to-prototype) comparing text-based vs. visual scripting in a higher education context.

- A validation of Visual Scripting not just as an introductory tool, but as a professional "Cognitive Bridge" for engineering students.
- A "Logic-First, Syntax-Second" pedagogical framework for integrating Game Engines into engineering curricula.

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

II. 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.

A. Learning Barriers and Novice Programming

Visual Programming Languages (VPLs) have long been advocated to lower the barrier to entry for novices. Ko et al. [10] 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. Wörister and Knobelsdorf [12] 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.

B. 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. [13], [7] 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-Valdés et al. [8] found that removing syntactic friction allows students to solve complex hardware control problems more effectively. Tramonti et al. [14] further emphasized that design thinking combined with visual tools enhances problem-solving in educational robotics. These findings in the hardware domain support the hypothesis that visual scripting in Unity, which controls "virtual hardware" (game physics), can yield similar educational benefits.

C. 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. [15] provide a technical survey of Unity, emphasizing its versatility for both 2D and 3D development. Recent work by Maraffi [16] 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. [17] conducted a systematic review concluding that game-based assessment can capture competencies that traditional tests miss. Furthermore, Maxim and Arnedo-Moreno [18] identify key principles in serious game design, suggesting that the development tool itself (visual vs. text) influences the quality of the final learning artifact. The present study contributes to this body of knowledge by quantifying the efficiency gains of visual tools in this specific context.

III. Theoretical Framework

A. 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).

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) [19], [20].

Fig. 1 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

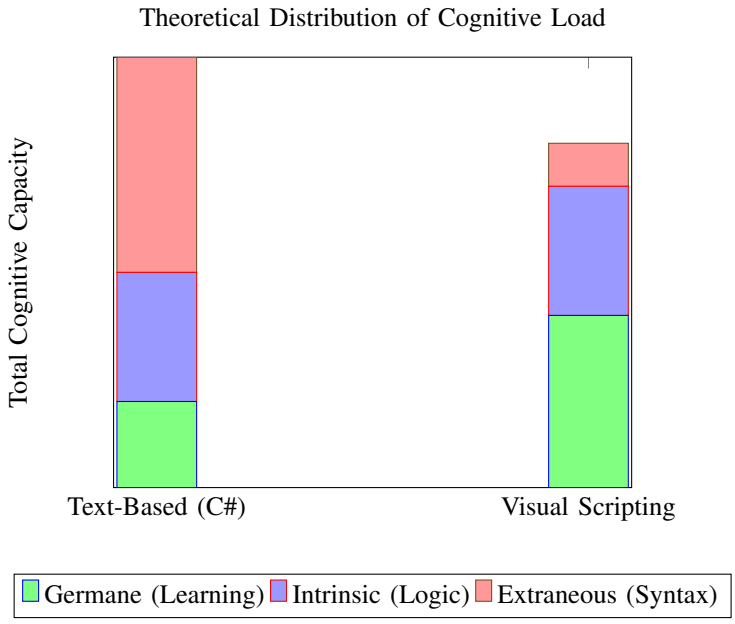


Fig. 1. Comparison of cognitive Load distribution. The reduction of extraneous load (Syntax) in visual scripting frees up cognitive capacity for germane load (Deep Learning), assuming constant intrinsic load (logic).

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.

This disparity is visually captured in the “Cognitive Gap". As illustrated in Fig. 2, while the algorithmic flow is intuitive, the textual implementation introduces extraneous cognitive load via syntax rules (braces, semicolons, pointers). This barrier often discourages learners before they can grasp the core logic [10].

B. 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. [21] and Tsai et al. [22] 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 [23].

IV. Methodology

A. 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 Object-Oriented Programming (OOP) concepts.

B. Instructional Design and Tasks

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.

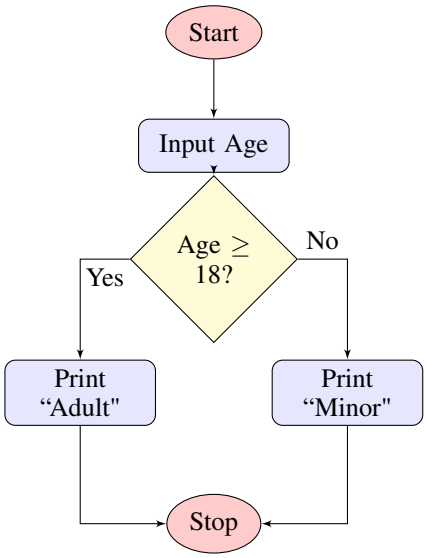
C. Experimental Design

A **Within-Subjects Quasi-Experimental Design** was adopted. The semester was divided into two phases to force a comparison within the same group of learners (see Fig. 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.

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

```
if (head.Position.X == food.Position.X &&
```



a) Algorithm (Mental Model)

```c
#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");
    }
    // SEMANTIC BARRIER:
    // Return types, scope
    return 0;
}
```

b) Implementation (Syntactic Barrier)

Fig. 2. 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. [10].

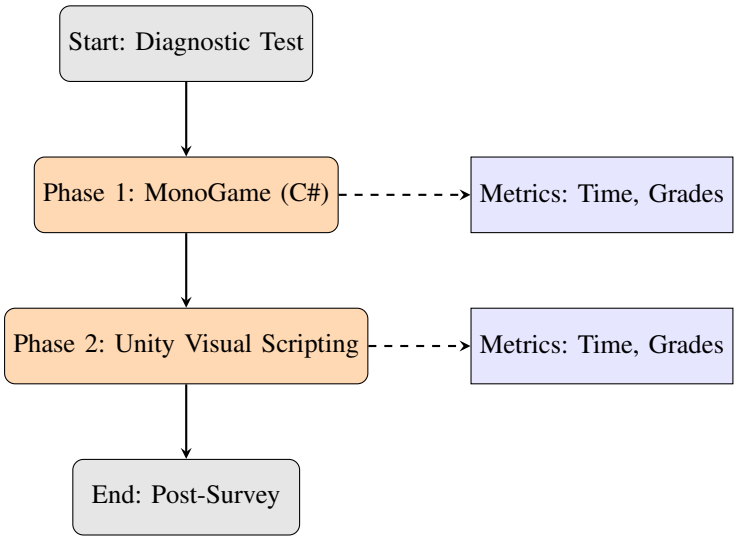


Fig. 3. Methodological workflow of the study. All students participated in both phases.

```
    head.Position.Y == food.Position.Y) {
    GrowSnake();
    SpawnFood();
}
```

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

V. Results

A. Development Efficiency Analysis

Instructors tracked the time taken to complete specific milestones in both phases. The differences were substantial.

As seen in Fig. 5, 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.

B. Academic Performance

Table I 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.

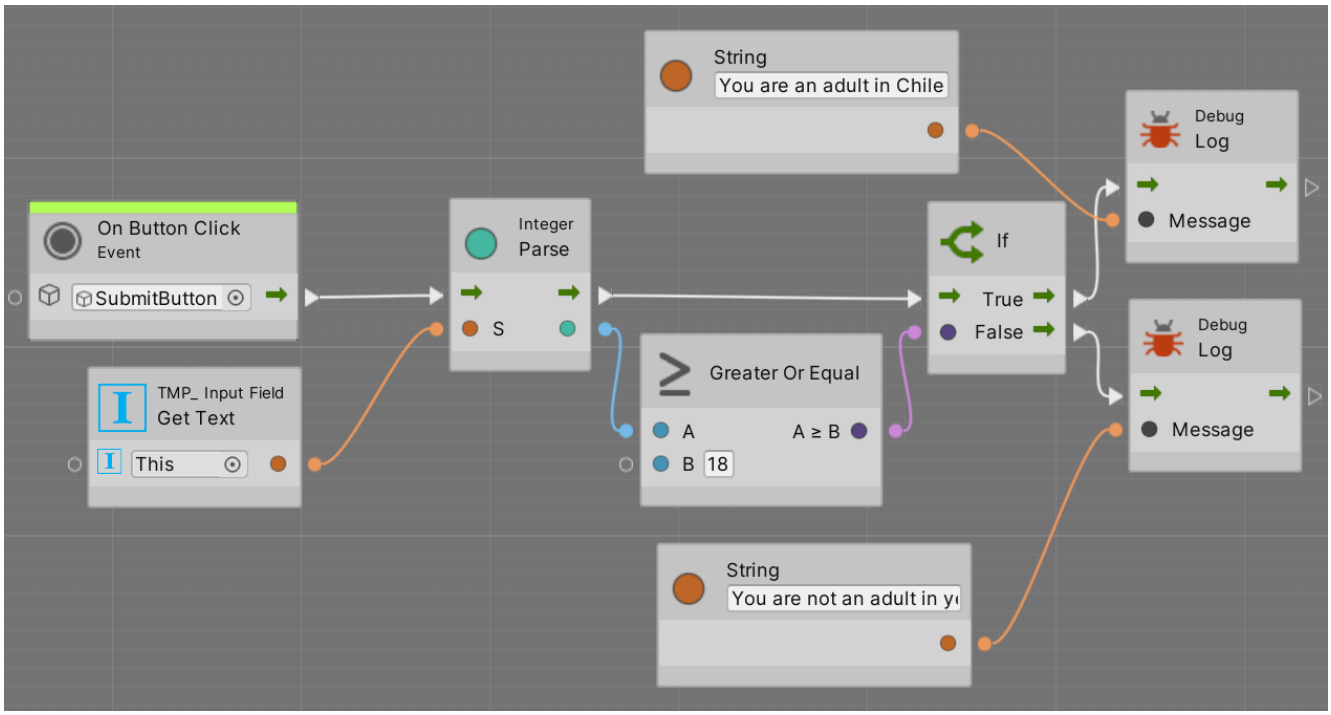


Fig. 4. Unity visual scripting solution. Nodes represent events and data flow, abstracting the syntax.

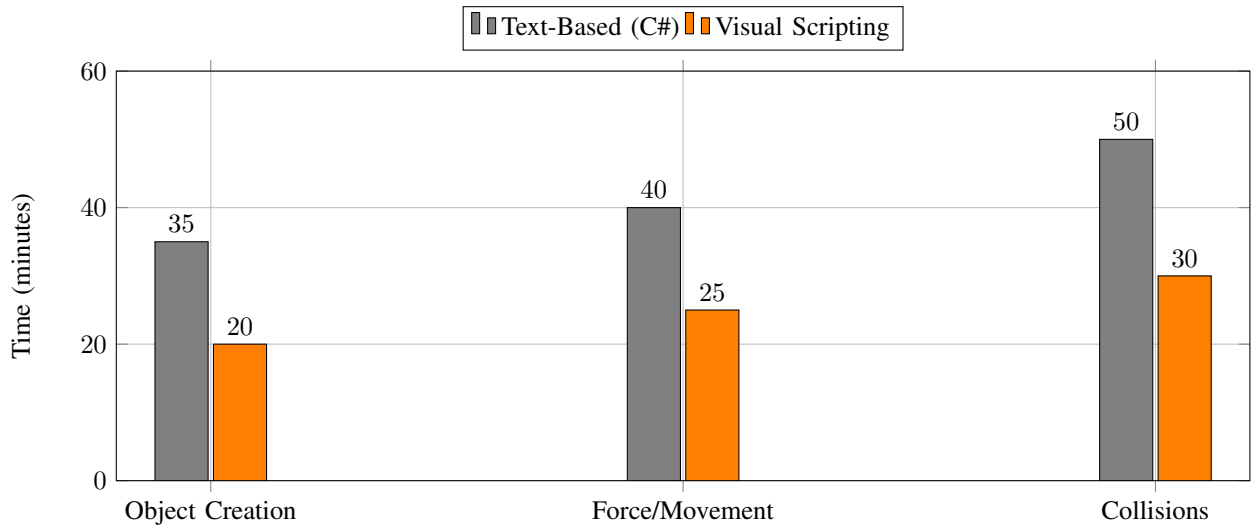


Fig. 5. Average development time per task. Visual scripting provided a 30-40% reduction in time-to-prototype.

TABLE I. 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

C. Student Perception and Self-Efficacy

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

TABLE II. SELECTED POST-SURVEY RESULTS (SCALE 1–10)

Statement	Mean
"Visual Scripting helped me understand programming logic."	8.9
"It reduced the difficulties I had with textual syntax."	8.7
"It is useful for learning fundamentals before moving to text."	8.8
"I would recommend it to beginners."	**9.0**

Table III 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. 4), 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].

VI. Discussion

A. Efficiency vs. Control Trade-off

The results support the hypothesis that Visual Scripting is significantly more efficient for prototyping (Fig. 5). 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.

B. 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.

The “Pedagogical Bridge Model” (Fig. 6) is proposed as a formal framework for curriculum design. Previous research often treats Visual Scripting as a distinct, alternative path. However, the 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 [7].
- Stage 2 (The Bridge): Unity Visual Scripting exposes the *Game Object Component* model and professional 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 the survey; they intuitively understood they were learning professional concepts without the syntactic penalty.

C. 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. Caeiro-Rodríguez 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 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, agreement is found with Perera et al. [27] that text-based languages remain fundamental. Therefore, a “Hybrid Syllabus" is proposed 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).

VII. 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.

A “Hybrid Scaffolded Curriculum" is recommended 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, 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 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.

TABLE III. TAXONOMY OF FREQUENT ERRORS: TEXT-BASED VS. VISUAL APPROACH

Error Category	MonoGame (C#)	Unity Visual Scripting
Syntactic	**High Frequency (65%)** Missing semicolons, mismatched braces, case sensitivity (e.g., `gameTime` vs `GameTime`).	**Null / Negligible** Syntax is enforced by node shapes. Connections are validated instantly by the UI.
Type Safety	**Moderate (20%)** Casting errors (e.g., `float` to `int`), null reference exceptions at runtime.	**Low (10%)** Visual conversion nodes are auto-suggested. Data ports are color-coded (e.g., Boolean is purple).
Logic / Physics	**Low Visibility (15%)** Logic errors buried inside valid syntax. Hard to visualize vector math.	**High Visibility (90%)** Errors like "Snake passing through walls" are immediately visible in the graph flow execution.

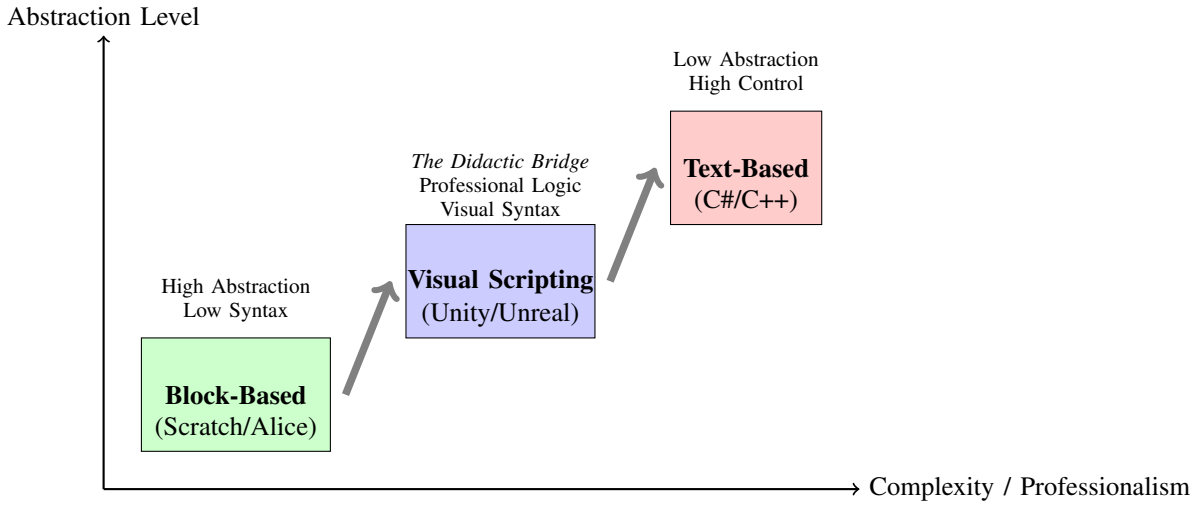


Fig. 6. The Proposed "Pedagogical Bridge Model". Visual Scripting serves as the necessary scaffold between K-12 block tools and professional text coding, aligning with Vinueza et al. [24].

The study has limitations inherent to its quasi-experimental 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.

Future research avenues are twofold. First, future work aims 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. Second, drawing from software product line engineering, the reuse of visual modules to manage variability in student projects will be analyzed, potentially automating the assessment of game mechanics. 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.

Ultimately, this study advocates for a paradigm shift in engineering education. By legitimizing Visual Scripting as a rigorous component of the curriculum, institutions can reduce the "syntactic friction" that drives attrition, fostering a new generation of engineers who are proficient in algorithmic logic before they even write their first line of syntax. This "Bridge" model offers a scalable path for adopting Industry 4.0 tools in higher education.

REFERENCES

[1] 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

[2] 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.

[3] 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.

[4] L. I. González-Pérez and M. S. Ramírez-Montoya, "Components of education 4.0 in 21st century skills frameworks: systematic review," *Sustainability*, vol. 14, no. 3, p. 1493, 2022.

[5] I. Mekterović, L. Brkić, B. Milašinović, and M. Baranović, "Building a comprehensive automated programming assessment system," *IEEE Access*, vol. 8, pp. 81 154–81 172, 2020.

[6] 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.

[7] C. Vidal-Silva, J. Cárdenas-Cobo, M. Tupac-Yupanqui, J. Serrano-Malebrán, and A. Sánchez 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.

[8] P. Rojas-Valdés, 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.

[9] L. Castillo-Salvatierra, J. Cárdenas-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.2025.1585602

[10] 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.

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

[12] F. Wörister and M. Knobelsdorf, "A block-based programming environment for teaching low-level computing," in *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research (Koli Calling '23)*. ACM, 2023, pp. 1–7.

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

[14] 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

[15] 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.

[16] 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

[17] M. J. Gomez, J. A. Ruipérez-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.

[18] 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

[19] 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.

[20] C.-Y. Chen, "Effects of worked examples with explanation types and learner motivation on cognitive load and programming problem-solving performance," *ACM Trans. Comput. Educ.*, vol. 25, no. 2, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3732791

[21] F. Ke, K. Xie, and Y. Xie, "Game-based learning engagement: A theory- and 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

[22] 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]. Available: https://doi.org/10.1016/j.compedu.2016.03.011

[23] R. Israel-Fishelson and A. Hershkovitz, "Persistence in a game-based 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

[24] M. Vinueza-Morales, J. Cárdenas-Cobo, J. Cabezas-Quinto, and 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.

[25] M. Caeiro-Rodríguez, M. Manso-Vázquez, F. A. Mikic-Fonte, M. Llamas-Nistal, M. J. Fernández-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.

[26] 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, pp. 2857–2878, 2022.

[27] P. Perera, G. Tennakoon, S. Ahangama, R. Panditharathna, and B. Chathuranga, "A systematic mapping of introductory programming languages for novice learners," *IEEE Access*, vol. 9, pp. 88 121–88 136, 2021.