Glassbox: Execution-Aware Interactive Illustrations for Algorithmic Design

Duarte Pardal[1], Inês Caetano[2], António Leitão[3]
[1,2,3]University of Lisbon
[1]duarte.pardal@tecnico.ulisboa.pt, [2]ines.caetano@tecnico.ulisboa.pt, [3] antonio.menezes.leitao@tecnico.ulisboa.pt

Abstract. *Algorithmic Design (AD) enables architects to express design intent through parametric logic to generate design alternatives. However, its abstract nature often hinders understanding, debugging, and communication of algorithmic logic. Prior works have shown that illustrations improve AD comprehension, but many do not cover the execution dynamics and internal runtime state of algorithms. This paper proposes Glassbox, an interactive, execution-aware illustration system for AD that dynamically links geometric outputs to the algorithm's structure and execution state. By capturing the history of operations leading to each output, the system generates a comprehensive dependency graph of the design, providing bidirectional traceability between output shapes and execution history. This enables novel interaction patterns: designers can select geometric elements to (1) generate illustrations reflecting the operations performed by the algorithm, (2) reveal the function call chain and local variables at the moment of the element's creation, (3) highlight shapes associated with specific execution contexts, (4) recursively inspect variable definitions, and (5) change numeric values impacting the element's shape. To evaluate the proposal, a prototype implementation was tested in comparative exploratory studies with architecture students. The results suggest that execution-aware illustrations enhance algorithmic reasoning, supporting deeper control and reflection within the AD process.*
Keywords. *Algorithmic Design, Design Representation, Interactive Visualization, Algorithm Traceability, Parametric Design, Design Cognition.*

INTRODUCTION

Computer-Aided Design (CAD) tools have changed how architects develop and present their designs, making them easier to modify, share, and analyze. Algorithmic Design (AD) builds on this by shifting the focus to programmatic generation, where users create algorithms that instantiate geometry based on input parameters. With AD, architects can move beyond the limitations of standard tools and are constrained only by the computational power of the system (Caetano et al., 2020).

Despite its expressive power, AD's adoption remains hindered by a steep learning curve, especially for designers with no programming experience. Its abstract nature makes it difficult to understand, manipulate, and communicate how geometric outcomes are produced (Castelo-Branco et al., 2022). Visual programming tools attempt to bridge this gap but often lack the debugging and

runtime inspection features common in traditional software engineering. Other approaches involve automating the illustration of algorithm logic to improve comprehension (Castelo-Branco et al., 2024). However, these illustrations are often static and fail to represent the dynamic execution flow or the internal state of the algorithm during runtime.

This paper proposes an interactive, execution-aware illustration system for AD that allows users to visually correlate algorithmic logic to geometric changes in real-time. By performing runtime introspection, the system reveals the function calls, operations, and variables involved in the creation of user-selected shapes. The information is displayed through annotations overlaid on the output, providing bidirectional traceability between the code and the geometry. To evaluate this approach, a prototype was tested with architectural design students to assess its impact on their ability to reason about algorithmic behavior and predict geometric outcomes.

RELATED WORK

Many systems have been developed to aid the understanding of algorithms through various visualization and tracing strategies. The next sections discuss the cognitive challenges of AD, intelligibility features, existing programming environments, and the use of automated illustrators.

Cognitive perspectives in AD

AD is cognitively demanding and fundamentally different from traditional creative thinking. It is difficult to learn, creating significant cognitive challenges for designers as they transition from geometric manipulation to defining concrete parametric rules (Monedero, 2000). Recent research suggests that effectively addressing these challenges requires a dual-process model that aligns design thinking with computational thinking (Kelly and Gero, 2021).

This difficulty stems from the need to develop computational thinking skills, such as abstraction, systematic logic, and debugging, as well as the inherent complexity of AD descriptions. These often exceed the capacity of standard design environments to provide clear feedback on whether algorithms match the architects' design intent (Leitão and Santos, 2011). Furthermore, the lack of features specifically dedicated to comprehending the internal logic of algorithms remains a major barrier in architectural education (Castelo-Branco and Leitão, 2022). Aligning algorithmic thinking with design reasoning is crucial for facilitating the learning and use of AD. However, despite the importance of framing AD tools as interpretation and communication mediators (Self, 2019), educational tools that bridge the gap between abstract logic and design intent remain underdeveloped.

Intelligibility features

Understanding the behavior of algorithms is a long-standing challenge in Computer Science, often arising in the context of debugging. The traditional approach involves inspecting program state at discrete points in time, often accompanied by interactive step-by-step execution using debuggers. Communicating how an algorithm functions often focuses on the evolution of the program state over time, usually represented through domain-specific diagrams relevant to the problem at hand (e.g., bar charts, graphs), as seen in platforms like *VisuAlgo* (Halim, 2015) and *Algorithm Visualizer* (Park, 2016).

However, in AD environments, similar techniques remain underexplored. A feature particularly relevant to AD is bidirectional traceability: the ability to query which parts of the algorithm generated specific parts of the output, or vice versa. This feature is partially implemented in Grasshopper and Dynamo and is primarily used to help users navigate large algorithms by visually linking components to their produced geometry. However, standard AD systems often fail to provide mechanisms to understand intermediate operations, inspect runtime values, or determine exactly how input variations influence specific results.

Visual programming and real-time parametric environments

A popular approach to enhancing algorithm legibility is the use of dataflow diagrams instead of code (e.g., Grasshopper, Dynamo, and Blender). This representation supports visual reasoning and alleviates the need to learn complex programming syntax, making AD more accessible to architects (Navarro-Prieto & Cañas, 2001).

This approach, however, scales poorly: as complexity increases, dataflow diagrams become less compact and more difficult to manage than equivalent code. In addition, many dataflow-based AD tools lack key programming constructs, such as functions, loops, and closures, making the representation of certain algorithms infeasible. Blender is an exception, as it supports the previously mentioned elements. Moreover, these systems often lack the ability to inspect values at runtime or to determine precisely which input values influence specific parts of the output. While some academic tools have explored interactive state inspection to facilitate this correlation (Alfaiate et al., 2017), most contemporary AD environments still lack geometric representations of runtime operations and states.

Illustrations for AD algorithms

To further aid comprehension, some systems provide explanatory illustrations of algorithmic designs.

Lopes & Leitão (2014) addressed the disconnect between code and output by embedding architectural sketches directly within the program and providing bidirectional traceability between the source code and the generated geometry. Khepriillustrator (Castelo-Branco et al., 2024), an automatic algorithm illustrator designed for Khepri (Sammer et al., 2019), extended this perspective by automatically creating static illustrations for any given algorithm and choice of its input arguments, for educational contexts—a previously labor-intensive manual process. By evaluating an algorithm and overlaying the output with explanatory annotations, such as arrows, angle arcs, and labels (Figure 1), the system illustrates all geometric operations performed by the algorithm. However, both approaches remain limited in representing the algorithms' internal state during execution as they rely on static illustrations that do not show values stored in variables.

Another approach, developed by Kelly et al. (2015), uses dimensioning annotations and interactive handles over a 3D parametric model. This allows users to modify parameters directly within the design space by dragging the handles, significantly reducing interaction friction. This approach, however, remains opaque regarding the algorithm's internal logic, as it is designed for manipulation of input arguments rather than algorithm refinement or comprehension. It also requires manual definition of the base lines used to position the annotations and handles.

Summary

Despite advances in visual programming and automated algorithm illustration, AD environments still lack execution-aware representations that explicitly connect geometric output, algorithm structure, and runtime state. This gap motivates the development of the proposed system, which provides a transparent, real-time link between an algorithm's logic and its geometric results.

THE GLASSBOX SYSTEM

Algorithms are often defined as instructions that operate on values of a variety of data types. To make this process transparent, the Glassbox system provides real-time explanations of how each value is created and processed, capturing both the concrete values at runtime and the relationship between instructions and geometric outputs. The system augments Khepriillustrator's approach of generating illustrations automatically with bidirectional traceability, a dynamic stack trace, and the ability to interactively modify numeric arguments to immediately observe their impact on the final geometry/design.

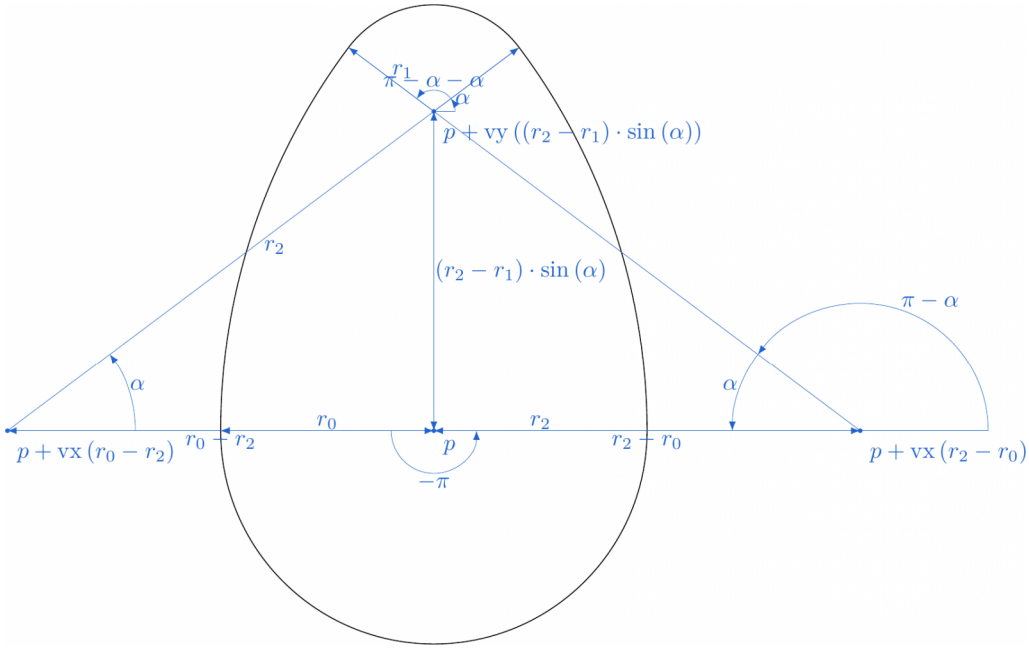


Figure 1
An illustration generated by KheprilIlustrator for an algorithm that draws a parameterizable egg curve.

The system structure is independent of specific programming languages or AD tools, though the implemented prototype uses the Julia programming language integrated with the Khepri AD tool (Sammer et al., 2019). The system comprises two major components: a **Tracer** and an interactive **Visualizer** .

Tracer

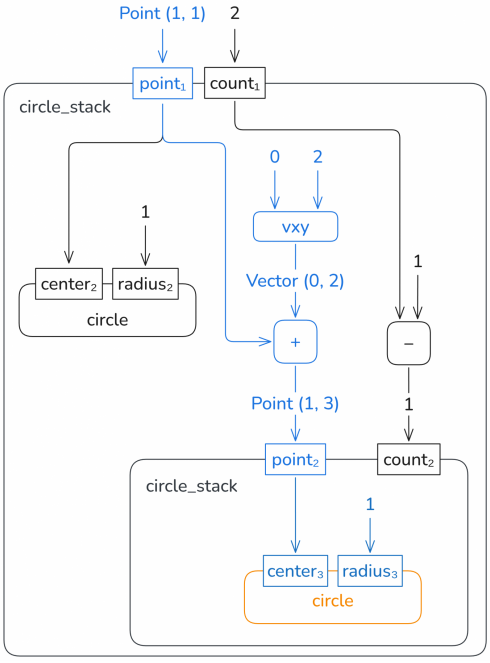
The **Tracer** performs runtime introspection on the executing algorithm, recording data, such as operations performed, operands used, and resulting values, from a live execution. This process generates a dependency graph that captures instantiated values and operations performed.

Unlike a dataflow diagram, which represents the algorithm itself, our graph represents its *execution history*. For example, if a recursive function or a loop executes multiple times, the **Tracer** generates unique nodes for each repetition, preserving the entire computational history of every individual geometric element.

The graph is constructed dynamically as the algorithm runs. Whenever a new value is created, a new node is appended to the graph. If the new value is the result of an operation, its node is connected to the nodes that represent its operands. Initializing a variable (including a function parameter) with a value also creates a new node, connected to the node of the assigned value. For simplicity, this description ignores optimizations.

This graph is illustrated in figure 2 for a recursive Khepri algorithm that stacks unit circles along the y axis (**Algorithm 1**). Note that, in Khepri, points are created using the *xy()* function and vectors are created using the *vxy()* function.

Figure 2
Visual representation of the dependency graph created when calling **Algorithm 1** with *point = xy(1, 1)* and *count = 2*: subscripts indicate recursion depth; rounded rectangles represent function calls and basic operations; and sharp rectangles represent variables.




```
Algorithm 1:
circle_stack(point, count) = begin
  circle(point, 1)
  if count > 1
    circle_stack(point + vxy(0, 2),
                 count - 1)
  end
end
```

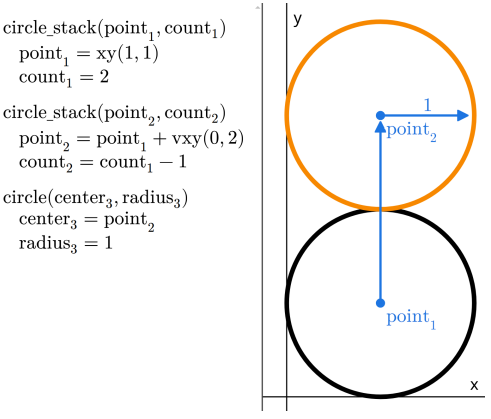
In this example, the point with coordinates (1, 1) and the number 2 are instantiated and passed to the initial call of the *circle_stack()* function as the *point* and *count* arguments, respectively. The built-in *circle()* function is then invoked, receiving the *point* argument from the *circle_stack()* function as the *center* and a newly instantiated value of 1 for the *radius*. The point argument is then added with the vector with coordinates (0, 2) in the expression *point + vxy(0, 2)*. The resulting value is passed to the *point* argument of the subsequent *circle_stack()* call, generating a node that represents the addition operation. Similarly, the result of subtracting *1* from the initial *count* argument is passed to the recursive call, which then calls the *circle()* function as before, terminating the recursion because the count condition is no longer met.

Interactive visualizer

The **Visualizer** transforms the dependency graph created by the **Tracer** into human-understandable representations using annotations overlaid on the geometric output and a complementary call chain listing all intermediate values and operations performed, as shown in figure 3. Because the graph stores information about the execution process (including where and when each function was called), the **Visualizer** provides bidirectional traceability. Backward traceability is achieved by traversing the dependency graph starting from the selected shape, while forward traceability is achieved by iterating through the execution history to identify all shapes resulting from a specific line of code.

The backwards traceability process is illustrated in figure 3: selecting the second circle (in orange) triggers dynamic annotations, in this case a dot

Figure 3
Interactive illustration created when selecting the second circle (in orange): the arrows, dots, and labels (in blue) represent the values and operations involved in the creation of the shape; the left panel displays the corresponding call chain, showing the function calls in which the shape was created.



labeled as *point*$_2$ and an arrow labeled with 1 representing the circle's center and radius, respectively. This is complemented by an arrow representing the translation operation performed on the bottom circle's center (*point*$_1$). These illustration elements correspond directly to the nodes highlighted in blue in figure 2, obtained by traversing the graph starting from the second *circle()* call. This live link between geometry and source code allows the user to identify the line of code and runtime variables responsible for a specific shape.

Additionally, clicking on the name of a variable—either in the illustration or in the call chain—automatically replaces it with the expression that defines it. For example, in figure 3, clicking on *point* $_2$ would replace it with *point* $_1$ + *vxy(0, 2)* .

The system also provides a mechanism to highlight all shapes produced within a specific function invocation. By selecting the desired invocation in the call chain, the **Visualizer** highlights the corresponding geometric group, mapping the scope of a function call to its geometric manifestation in the generated design.

Besides the illustration and inspection functions, the **Visualizer** also provides a way to smoothly change numeric arguments and literals through mouse-based interaction, as shown in **figure 4** .

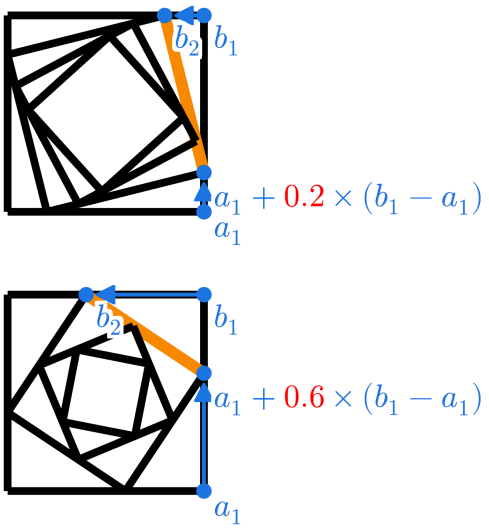


Figure 4
By dragging with the mouse on top of the number literal in the expression (highlighted in red), the user can modify it, causing the output to update to match the new value.

EVALUATION

To evaluate the impact of real-time interactive illustrations on algorithmic comprehension, the Glassbox system was integrated into Visual Studio Code as an extension that provides a real-time interactive preview that updates as the code is edited (Figure 5). The evaluation was conducted in an introductory AD course within the Integrated Master's in Architecture at Instituto Superior Técnico, University of Lisbon. The study involved 21 students from the aforementioned course with little to no prior programming experience, divided between two shifts: one (the experimental group) used the Glassbox system with all features enabled, while the other (the control group) used a restricted version of the system with the illustration overlay and the dynamic call chain pane deactivated. The experiment spanned three weeks, consisting of two sessions per week for each group, totaling six sessions.

In both groups, the system also provided automated visual feedback by color-coding the geometric output: elements aligning with the target design were highlighted in green, while discrepancies were highlighted in red. This real-time validation was intended to assist students in self-correcting their logic during the exercises.

The experimental procedure began with a standardized introduction phase, where the students familiarized themselves with the system's interface through a series of preparatory exercises.

The students were then tasked to complete a series of exercises of increasing complexity:

- Finding the arguments of a given algorithm that make the output match a target design.

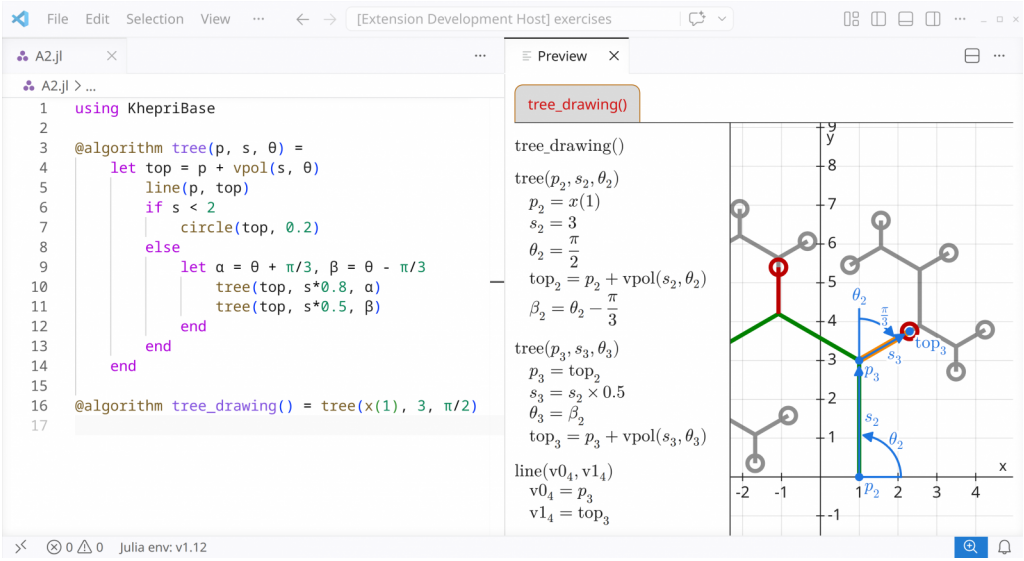


Figure 5
An exercise where students modified a given algorithm (on the left) to match the target design (on the right, in gray). The algorithm's output shapes are either green or red, depending on whether they match the target design or not. The blue annotations represent the operations performed by the algorithm and the the middle panel contains to the dynamic call chain.

- Modifying or correcting a given algorithm, which is either incomplete or presents errors, to behave according to a given specification.
- Creating an algorithm from scratch that behaves according to a given specification.

After completing each task, students rated the perceived difficulty using a Likert scale in the range of 1 to 5. Additionally, each class concluded with a verbal debriefing, in which students were encouraged to provide feedback on the illustration system. This allowed for qualitative data collection regarding the system's usability and the perceived effectiveness of the execution-aware illustrations.

RESULTS

A total of 122 data points were collected, each representing a student's subjective difficulty score measured on a 5-point scale. The time each student took to solve each exercise was also recorded. However, this metric was excluded from the analysis as its reliability was severely compromised by unplanned student breaks.

Quantitative results

To evaluate the impact of the illustration system, a statistical permutation test, stratified by exercise, was conducted on the subjective difficulty scores. The test statistic used was the average of the differences of means for each exercise, weighted by the number of data points per exercise.

The results indicate that students in the experimental group (with access to interactive illustrations) rated the tasks 0.67 points less difficult on average than the control group. The test yielded a p-value = 0.009, disfavoring the null hypothesis that the presence of illustrations has no effect on the perceived difficulty of algorithmic tasks. The 95% confidence interval for the average difference in means, calculated using statistical bootstrapping, is [0.22, 1.12].

The interpretation of these results must account for confounding factors correlated with

the group assignments. These include variations in pedagogical instructions/explanations (different teachers), as well as the time of day and day of the week when the sessions occurred. While these factors may have influenced the absolute scores, these preliminary results warrant further investigation into the role of interactive illustrations in reducing cognitive load and improving the intelligibility of AD descriptions.

Qualitative feedback

The qualitative data gathered during the debriefing sessions provided further insight into the system's performance. In general, students reported that the annotations were helpful for understanding the output. However, some experienced cognitive friction when interpreting the dynamic call chain, suggesting a need for further refinement in the representation.

Bidirectional traceability was used less frequently than expected. This may derive from the moderate complexity of the assigned exercises, which may not justify its use: as the algorithms were relatively small, the students could often mentally map the code to the geometry without assistance from the system.

While not directly related to the goal of this project, students showed appreciation of the integrated real-time visualization and automated verification, allowing them to see live updates within the programming environment rather than in external CAD software (e.g., AutoCAD). According to the students, this accelerated the iterative design loop and reduced overall task duration.

DISCUSSION

The results of this study indicate that the use of execution-aware environments may help smooth the AD learning curve in educational settings. The observed average decrease of 0.67 points in difficulty scores suggests a potential reduction in the cognitive load traditionally associated with algorithmic tasks.

In traditional AD workflows, designers are often required to maintain a complex internal mental model of program state. By externalizing this state, the proposed system acts as a cognitive scaffold that allows students to visualize operations as they occur. The statistical analysis indicates that, within the context of this specific cohort, the presence of dynamic annotations contributed to a more accessible learning experience.

The limited application of bidirectional traceability may reflect the moderate complexity of the assigned exercises rather than a lack of utility in the tool itself. For relatively simple recursive algorithms, mental mapping remains a viable strategy for many students. However, we hypothesize that the value of execution-aware traceability will become more apparent in high-complexity scenarios, where manually verifying the algorithm's logic is no longer feasible.

The difficulty students experienced in interpreting the call chain can be attributed to both the inherent conceptual complexity of the representation and the high volume of information displayed. While the **Tracer** successfully captures the full computational history, presenting this data in a way that minimizes confusion without omitting critical details remains a challenge. Addressing this issue will require extensive usability testing to determine the optimal balance between transparency and legibility.

Lastly, the students' positive response to immediate feedback, as opposed to the traditional visualization of results in external CAD tools, motivates research from a broader design perspective. By reducing the temporal gap between algorithmic development and result visualization, the system facilitates a more exploratory and dynamic AD process. This allows users to discover the algorithm's behavior through real-time interaction, potentially aiding the creative process (Victor, 2012). Nevertheless, this perspective requires further investigation to determine if such

immediacy leads to better design outcomes or simply accelerates the completion of AD tasks.

CONCLUSION

This paper presented *Glassbox*, an execution-aware illustration system designed to bridge the gap between abstract algorithmic logic and geometric output in architectural design. By implementing a **Tracer** capable of capturing live runtime data and a **Visualizer** that provides bidirectional traceability and spatial annotations, the system moves beyond static documentation toward a live, bidirectional link between algorithmic logic and geometric outcomes.

Our initial user study indicates that such an environment can smooth the learning curve for AD, as evidenced by the observed statistical results on the perceived difficulty of solving algorithmic tasks. While challenges remain regarding data density and the legibility of complex call chains, the positive feedback from students suggests that immediate, execution-aware feedback is valuable in pedagogical settings.

Future work will primarily focus on improving the usability of the system for both students and professional users. Specific areas for development include:

- Improving the legibility of illustrations, possibly by implementing optimization algorithms to minimize overlapping annotations and self-intersections.
- Exploring the use of heuristics to reduce information overhead and "visual noise" in large-scale architectural models, configurable according to the design context and the user's expertise.
- Investigating the impact of real-time dynamic annotations on the quality and creativity of design outcomes, moving beyond simple task-based efficiency.
- Testing the system in other contexts, especially with practitioners already familiar with AD.
- Exploring how to effectively illustrate algorithms in 3D.

ACKNOWLEDGEMENTS

Work supported by national funds through Instituto Superior Técnico (IST) under grant 1018P.06759.1.01 and through Fundação para a Ciência e a Tecnologia, I.P. (FCT) under projects UID/50021/2025 (DOI: https://doi.org/10.54499/UID/50021/2025) and UID/PRR/50021/2025 (DOI: https://doi.org/10.54499/UID/PRR/50021/2025).

REFERENCES

Alfaiate, P., Caetano, I., & Leitão, A. (2017). Luna Moth: Supporting Creativity in the Cloud. In T. Nagakura, S. Tibbits, M. Ibanez, & C. Mueller (Eds.), *Disciplines & Disruption: Proceedings of the 37th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA)* (pp. 72–81). https://doi.org/10.52842/conf.acadia.2017.072

Caetano, I., Santos, L., & Leitão, A. (2020). Computational design in architecture: Defining parametric, generative, and algorithmic design. *Frontiers of Architectural Research*, *9*(2), 287–300. https://doi.org/10.1016/j.foar.2019.12.008

Castelo-Branco, R., Caetano, I., & Leitão, A. (2022). Digital Representation Methods: The Case of Algorithmic Design. *Frontiers of Architectural Research*, *11*(3), 527–541. https://doi.org/10.1016/j.foar.2021.12.008

Castelo-Branco, R., Caetano, I., & Leitão, A. (2024). Algorithmic design explained: decomposing parametric 3D problems into 2D visual illustrations. *ACCELERATED DESIGN, Proceedings of the 29th International Conference of the Association for Computer-Aided Architectural Design Research in Asia (CAADRIA)*, *3*, 9–18. https://doi.org/10.52842/conf.caadria.2024.3.009

Castelo-Branco, R., & Leitão, A. (2022). Comprehending Algorithmic Design. In D. Gerber, E. Pantazis, B. Bogosian, A. Nahmad, & C.

Miltiadis (Eds.), *Computer-Aided Architectural Design. Design Imperatives: The Future is Now* (pp. 15–35). Springer, Cham. https://doi.org/10.1007/978-981-19-1280-1_2

Halim, S. (2015). VisuAlgo - Visualising data structures and algorithms through animation. *Olympiads in Informatics*, *9*, 243–245. https://doi.org/10.15388/ioi.2015.20

Kelly, N., & Gero, J. S. (2021). Design thinking and computational thinking: a dual process model for addressing design problems. *Design Science*, *7*, e8. https://doi.org/10.1017/dsj.2021.7

Kelly, T., Wonka, P., & Mueller, P. (2015). Interactive Dimensioning of Parametric Models. *Computer Graphics Forum*, *34*(2), 117–129. https://doi.org/10.1111/cgf.12546

Leitão, A., Lopes, J., & Santos, L. (2014). Illustrated Programming. In D. Gerber, A. Huang, & J. Sanche (Eds.), *Design Agency: Proceedings of the 34th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA)* (pp. 291–300). https://doi.org/10.52842/conf.acadia.2014.291

Leitão, A., & Santos, L. (2011). Programming Languages for Generative Design: Visual or Textual? In T. Zupanci, M. Juvancic, S. Verovsek, & A. Jutraz (Eds.), *Respecting Fragile Places: Proceedings of the 29th Education and research in Computer Aided Architectural Design in Europe (eCAADe) Conference* (pp. 549–557). https://doi.org/10.52842/conf.ecaade.2011.549

Monedero, J. (2000). Parametric design: a review and some experiences. *Automation in Construction*, *9*(4), 369–377. https://doi.org/10.1016/S0926-5805(99)00020-5

Navarro-Prieto, R., & Cañas, J. J. (2001). Are visual programming languages better? The role of imagery in program comprehension. *International Journal of Human-Computer Studies*, *54*(6), 799–829. https://doi.org/10.1006/ijhc.2000.0465

Park, J. (2016). *Algorithm Visualizer*. https://algorithm-visualizer.org/

Sammer, M. J., Leitão, A., & Caetano, I. (2019). From Visual Input to Visual Output in Textual Programming. In M. H. Haeusler, M. A. Schnabel, & T. Fukuda (Eds.), *Intelligent & Informed: Proceedings of the 24th International Conference of the Association for Computer-Aided Architectural Design Research in Asia (CAADRIA)* (Vol. 1, pp. 645–654). https://doi.org/10.52842/conf.caadria.2019.1.645

Self, J. A. (2019). Communication through design sketches: implications for stakeholder interpretation during concept design. *Design Studies*, *63*, 1–36. https://doi.org/10.1016/j.destud.2019.02.003

Victor, B. (2012). *Inventing on Principle*. https://vimeo.com/906418692