This version of the contribution has been accepted for publication, after peer review but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/10.1007/978-3-031-37189-9_3. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use.

Illustrating Algorithmic Design

Renata Castelo-Branco[0001−9965−9558] and António Leitão[0001−7216−4934]

INESC-ID/Instituto Superior Técnico, University of Lisbon, Portugal
{renata.castelo.branco,antonio.menezes.leitao}@tecnico.ulisboa.pt

Abstract. Architectural design is strongly based on visual and spatial reasoning, which is not easy to translate into algorithmic descriptions and, eventually, running programs, making it difficult for architects to use computational approaches, such as Algorithmic Design (AD). One of the most pressing problems is program comprehension. To overcome it, we propose an automatic illustration system for AD programs that produces annotated schemes of the program's meaning.
The illustration system focuses on a basic set of geometric elements used in most calculations to place geometry in space (points, distances, angles, vectors, etc.), and on the way they are manipulated to create more complex geometric entities. The proposed system automatically extracts the information from the AD program and the resulting illustrations can then be integrated into the AD program itself, intertwined with the instructions they intend to explain.
This article presents the implementation of this solution using an AD tool to generate the illustrations and a computational notebook to intertwine the program and the illustrations. It discusses the choices made on the system's implementation, the expected workflow for such a system, and potential future developments.

Keywords: Algorithmic Design · Illustration · Documentation · Program Comprehension.

1 Introduction

In the Algorithmic Design (AD) approach, designs are represented through algorithmic descriptions that, when executed, generate the corresponding digital model [5,32]. AD's parametric nature [3] promotes design experimentation, facilitates the evaluation of multiple and often conflicting design requirements [33,20], and promotes the production of large-scale unconventional designs [7].

However, AD is less intuitive than other design methods, and architects still struggle to understand how the AD program relates to the design it represents, particularly in programs developed by others, an increasingly common scenario in collaborative work environments [35]. To help practitioners understand AD programs, we propose an automatic illustration system that can produce annotated 2D schemes explaining parameters and other relevant relations that compose the AD. The proposed system provides illustrations for a set of basic

geometric elements, such as distances, angles, points, vectors, etc., which can then be intelligently combined to produce useful illustrations.

To support the natural evolution of the design, the planned use is for architects to piggyback the generation of illustrations on top of the AD program they are developing, making it generate not only the intended architectural model but also the illustrations explaining it. Combined with existing visual documentation techniques for AD [8,9], these illustrations can then be intertwined with the program itself. By promoting a dialog between the algorithm and the design it represents, this proposal aims to reduce AD's comprehension-related drawbacks, improving the development, maintenance, and sharing of AD programs.

2 A challenging practice

Although initially met with resistance, the paper-to-digital transition brought considerable advantages to architectural design [21,6] and the adoption of digital-based design methods rapidly increased, especially because the developed digital design tools emulated existing representation methods, only with more precision and efficiency. Currently, another big leap is taking place with the use of AD, a design approach that represents designs through algorithmic descriptions [31], i.e., computer programs with rigorous instructions for a computer to perform [3].

Even further from the hands-on nature and materiality of traditional architectural development [22], AD relies on abstractness to transcend the constraints to representation and imagination that bind prior digital design tools [22], outperforming them in terms of flexibility and expressiveness [7]. Additionally, AD can seamlessly integrate analysis and optimization in design exploration processes, allowing performance to act as a design principle [4]. This capacity is becoming increasingly critical in an era where the industry is pressed to reduce not only time and cost requirements but also its environmental impact [23].

However, with fewer analogies to traditional representation methods than previous digital design paradigms, AD has some challenges ahead. One of them is the need for programming skills. Withal, even for experienced programmers, the ability to achieve design thinking with AD remains difficult, since these are two very distinct modes of thought [15]. Design representations are meant to stimulate creativity [28] by extending architects' imagination to the physical realm and establishing a feedback loop between both types of representation (internal and external to the creator's mind) [12]. Unfortunately, with AD, the practice seems unable to intuitively allow for this mutual influence.

3 Comprehension mechanisms

The architectural design process is strongly based on visual and spatial reasoning, which is not easily translated into algorithmic descriptions. However, the challenge in converting abstract ideas to and from algorithmic representations is not a specific drawback of AD. Program comprehension is a well-known problem in computer science and several solutions have been proposed in the past.

3.1 Documentation

Donald Knuth, for instance, proposed the development of programs as literary works [17]. Other examples include the use of graphical representations to explain programs, such as flow diagrams [13] and other diagrammatic techniques [14,19], as well as animations of the program's fundamental operations [2]. Although some of these works addressed the use of images as visual explanations for textual programs, particularly those that fell under the program visualization umbrella [27,25], they were still based on a computer science outline, which completely disregarded the intrinsic visual nature of the design process itself.

Architects have learned to think and reflect upon their designs through sketching, and ADs are no different. The hand-made drawings architects make during the design process represent their intentions for their AD programs, explaining the logic behind their conception and what they expect them to produce. They thus constitute fundamental information for the understanding of the architect's design idea and could be integrated with the algorithmic representation, illustrating and explaining it.

Some authors developed solutions to accommodate these assets in AD programs. Leitão et al. [18] proposed the inclusion of sketches, images, and renders in textual AD descriptions. However, their solution suffered from scalability issues. Grasshopper allows for the inclusion of imagery in the middle of AD programs, although, in the visual programming environment, it becomes hard to contain the clutter that these elements cause.

3.2 Computational Notebooks

Beyond documentation, other theories stress the importance of displaying not just the program's structure and behavior but also its evolution [11], which is crucial in a creative process such as AD. An interesting take on this problem has been forwarded by computational notebooks [29], which were designed to support computational narratives, allowing users to create and communicate their experiments in a comprehensible manner. To that end, they allow incremental development of programs with immediate feedback on their results, as well as the intertwining of code with textual and visual documentation.

The immediate feedback provided by computational notebooks touches on another comprehension aspect, which is liveliness [16]. Liveliness has been differently interpreted in different fields [26]. In the case of AD, it means that changes to the algorithmic description should have immediate repercussions in the generated design model so that users can relate the changes in the program to their respective impact on the model. Tools like Grasshopper [10] and LunaMoth [1] do this via live coding [26], whereas computation notebooks rely on other facets of liveliness, such as interactive evaluation and reactivity [9].

Relying on computational notebooks, previous works also explored the idea of integrating, in the AD program, hand-made drawings and other digital media produced during the design process (such as model screenshots and rendered images) [8,9]. However, there is a downside to these elements. Screenshots and

renders typically present a global view of the digital model, rarely succeeding in explaining the relevance of particular program fragments for the generation of the intended geometry. Hand-made drawings offer finer control over which aspects the architect wishes to illustrate. The problem, however, lies in their static nature. Scanned drawings incorporated in the program rapidly get outdated as the design evolves. And whereas repurposing drawings made during design ideation has little cost for the architect, taking the time to correct outdated drawings imposes a higher penalty that not many are willing to pay.

4 Automatic illustration

Many of the existing solutions in the field of program comprehension contemplate or derive from the computational thinking paradigm and do not necessarily comply with the visual demands of AD. There is still much to explain in the intricate dependencies that compose a parametric program and there is still a long way to go to make AD more akin to traditional architectural design processes. As a first step toward that goal, we turn our attention toward the essence of design experimentation and the most popular means used to iterate over design ideas: drawings.

Building upon the solution presented in section 3.2, we propose to overcome the previously-mentioned shortcomings with (1) the automatic generation of computer-made geometric illustrations explaining relevant aspects of the algorithmic description, and (2) their subsequent integration in the AD program. The envisioned workflow is for architects to generate and integrate automatic illustrations with as little extra work as possible, as well as automatically update them given any changes to the algorithmic description. To that end, the proposed system extracts as much information as it can from the existing AD description, requiring users to write additional instructions only if and when they wish to alter the system's default behavior. Since the illustrations are automatically extracted from the algorithmic descriptions, they can be regenerated at any time, ensuring the program's visual documentation is up to date.

For evaluation purposes, we implemented the proposal on top of the Khepri AD tool [30], which uses the Julia programming language and communicates with several modeling, rendering, analysis, and optimization tools. From this set, we chose two visualization tools to generate the illustrations: TikZ, a procedural drawing tool, and AutoCAD, a 2D drafting and 3D modeling tool known to most architects. As for the integration of the generated illustrations in the AD program, we follow the workflow proposed in [9], using computational notebooks to intertwine visual and textual documentation with the AD program and, thus, create a more comprehensible programming experience. The computational notebook used to exemplify the proposed system is Pluto [24].

4.1 Geometric elements

Khepri has a large set of pre-defined modeling operations capable of creating 2D and 3D geometry in multiple tools. The shape and positioning of that ge-

ometry typically depend on a series of calculations performed with the most basic geometric elements, such as points, vectors, distances, and angles. Using a multiplicity of coordinate systems (namely, cartesian, polar, cylindrical, and spherical) these elements can be used to form more complex geometric entities, such as arcs, circles, polygons, etc. These eventually get extruded, sweeped, lofted, and combined in other ways to create 3D shapes. However, more often than not, it is in these first steps that most geometric calculations are done and that most design parameters imprint their influence. These operations are thus the focus of our illustration proposal.

Fig. 1 shows, on the left, a 3D model generated by an AD program. The building's profile mimics an Islamic pattern and is achieved with two superimposed squares and eight circles centered on each of the eight intersection points and tangent to the bounding octagon. In this case, the tangency was achieved by calculating the correct circle radius using trigonometry. The resulting 2D surface is replicated in height with decreasing size and then extruded to make slabs and glass panels. The parameters defining the base shape are explained in the geometric illustrations on the right, with the recursive distribution of the circles represented with increasing transparency (further explained in section 4.5).

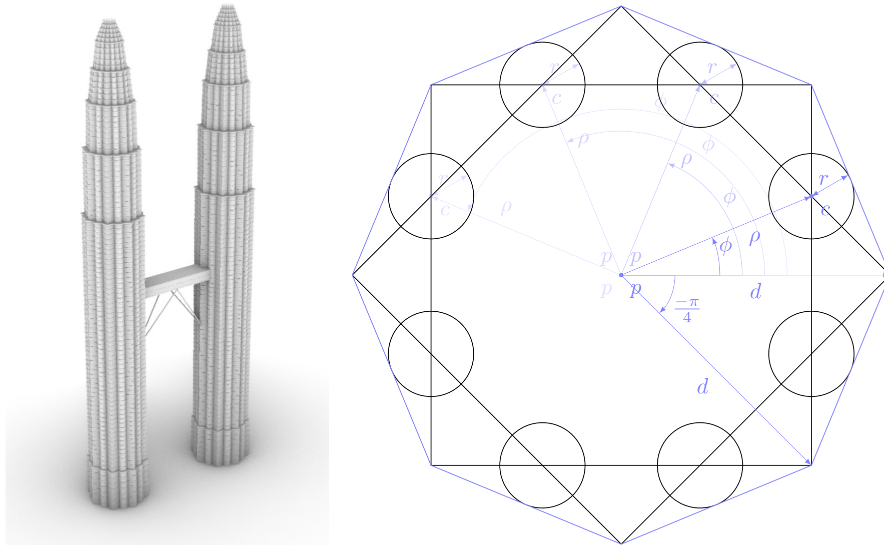


Fig. 1: Petronas Towers inspired 3D model and 2D geometric illustration generated in TikZ.

For the illustration process to piggyback the AD program itself with as little additional programming as possible, we capitalize on the parameters that users must already provide to create each element. For instance, in order to create a circle, users must specify a center and a radius (c and r, in the previous

example), whereas to create a regular polygon, besides center (p) and radius (d), an initial angle ($-\pi/4$) and the number of sides are also required. The automatic illustration of each element is, in most cases, simply comprised of the parameters required to create it. Naturally, beyond the automation, users are also given mechanisms to include additional information they may find relevant to the explanation (e.g., the 'invisible' octagon in the previous example).

4.2 The evaluator

The illustration system is implemented as a specialized Julia evaluator, which recognizes specific program patterns in the evaluated program and acts upon those, generating the appropriate illustration, before sending the instructions to the standard Julia evaluator, who resumes the normal evaluation cycle producing the program's originally intended results. The resulting behavior is similar to that of code injection in selected parts of the program.

The patterns recognized by the evaluator include operations between points and vectors in the same or different coordinate systems, and the basic geometry creation operations mentioned above. For each case, the evaluator explores the expressions used as arguments in the corresponding function calls to produce the annotation labels for the illustration. For example, in the definition of the arrow shape shown in Fig. 2, the evaluator intercepts the point-vector sums and draws the elements involved in each one, using the name of the points for their labels. Polar vectors, however, are represented with two elements - a distance and an angle - whose labels the evaluator fetches from the vectors' expressions (e.g., ρ and α, or θ and $\alpha + \pi - \beta$).

The illustrations intend to show the underlying process that leads to the final result. Since the same geometric outcome may be achieved using different program patterns, each of these will yield specific illustrations. Fig. 3 shows two versions of a function that places four tangent circles around a point. The first uses cartesian coordinates and the second uses polar coordinates. Accordingly, the automatic illustration of these instructions will produce different drawings, although the intended output of the functions is the same.

4.3 Programming style

Illustration allows us to visualize the entire generation process, whereas without it we would only see the final result. If not for illustration, users would likely choose between possible implementation styles on a whim. However, knowing that the way they write their program impacts its illustration will likely lead them to a more conscious choice, perhaps even to a new programming style.

Take the two definitions in Fig. 4 for an egg shape. The first one calculates the center points for the arcs directly within the arc calls, whereas the second one defines the center points as local variables, giving them specific names. The illustration below corresponds to the definition on the left, where the egg's larger radius is annotated with the local name r_2 instead of its corresponding but less readable definition $(r_0 - r_1 * cos(\alpha))/(1 - cos(\alpha))$. The illustration of the

```
arrow(p, ρ, α, θ, β) =
    let p1 = p + vpol(ρ, α)
        p2 = p1 + vpol(θ, α + π + β)
        p3 = p1 + vpol(θ, α + π - β)
        line(p, p1, p2, p3, p1)
    end
```

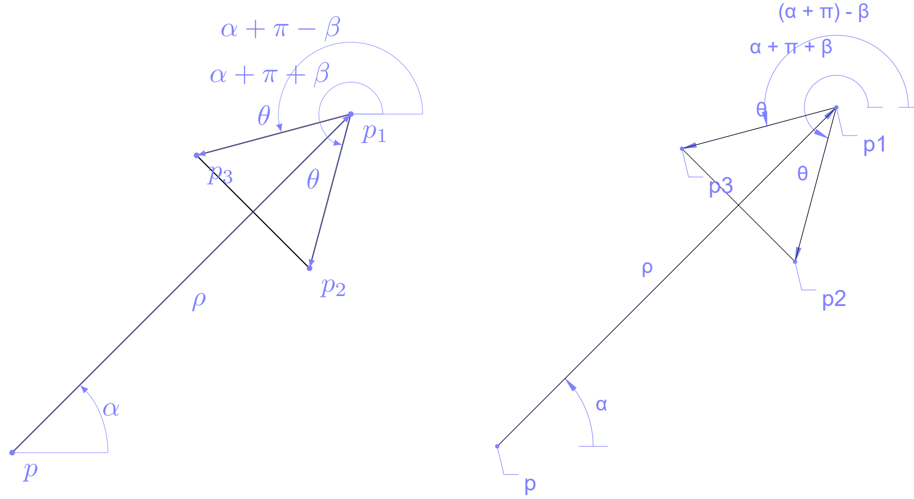


Fig. 2: Arrow shape illustration generated in TikZ (left) and AutoCAD (right).

second case would be even more intelligible since the same happens with all the lengthy expressions representing the points (an illustration corresponding to this definition can be found in Fig. 5). Despite the added trouble, having the points as separate entities from the arcs makes both the program and the illustration easier to read. As a general rule, properly naming variables over which we may want to operate is a good programming practice.

4.4 Too Much Information (TMI)

Some of the examples shown portrayed rather simple computations, with just the right amount of information to produce an intelligible illustration. However, illustrating all identified patterns, in most cases, will likely result in an excessively cluttered drawing with superimposed information.

To avoid overlapping annotations, the evaluator keeps track of all the elements generated in a single illustration, through their insertion points, lengths, angles, labels, etc. If it identifies collisions, the system distributes the colliding elements through the available space. For instance, coincident radii will be placed at different angles on the circle, coincident angles will be given different radii for their arcs (see Fig. 3 right), coincident lengths will be given different offsets from the line, and coincident labels will be placed at different angles and distances from the insertion point. Cases where both point and label coincide

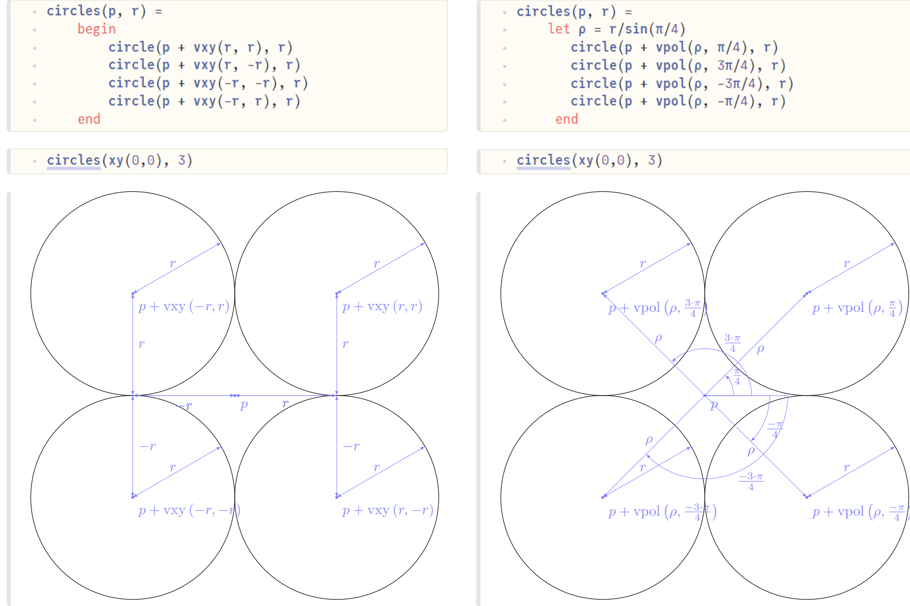


Fig. 3: Two possible definitions to place four tangent circles around a point: using cartesian (left) and polar coordinates (right). Both algorithmic description and illustration are shown in the Pluto notebook.

will simply not be generated. Naturally, not all elements possess such flexibility. Vectors, for instance, cannot change place, but their labels can.

Collision avoidance strategies work wonders at small scales, but they cannot perform miracles when there are simply too many elements to illustrate. As such, user-configurable flags are also available for users to control which elements get generated in each illustration, and if they wish to visualize the illustration step by step. In the latter case, the evaluator generates multiple illustrations, one step at a time, until the AD program fragment is completely evaluated. This results in a group of images that can then be combined in an animation. As an example, by applying this mechanism to the egg shape, we obtain a step-by-step illustration where the comprehension of each individual arc is improved (Fig. 5).

4.5 Repeated Illustrations

Loop instructions, such as `for` and `while` cycles, and, more importantly, recursive definitions are strong candidates for the generation of cluttered illustrations. If we repeat the illustration of the geometric elements for each iteration of the loop or each recursive call, we are likely to get not only repeated information but also superimposed geometry, which will be difficult to differentiate. To avoid this, the system increases the transparency of the annotations with each loop.

```
egg(p, r0, r1, h) =
    let α = 2*atan(r0-r1, h-r0-r1),
        r2 = (r0-r1*cos(α))/(1-cos(α))
        arc(p, r0, 0, -π)
        arc(p + vx(r0-r2), r2, 0, α)
        arc(p + vx(r2-r0), r2, π-α, α)
        arc(p + vy((r2-r1)*sin(α)),
            r1, α, π-α-α)
    end
```

```
egg(p, r0, r1, h) =
    let α = 2*atan(r0-r1, h-r0-r1),
        r2 = (r0-r1*cos(α))/(1-cos(α))
        p1 = p + vx(r0-r2)
        p2 = p + vx(r2-r0)
        p3 = p + vy((r2-r1)*sin(α))
        arc(p, r0, 0, -π)
        arc(p1, r2, 0, α)
        arc(p2, r2, π-α, α)
        arc(p3, r1, α, π-α-α)
    end
```

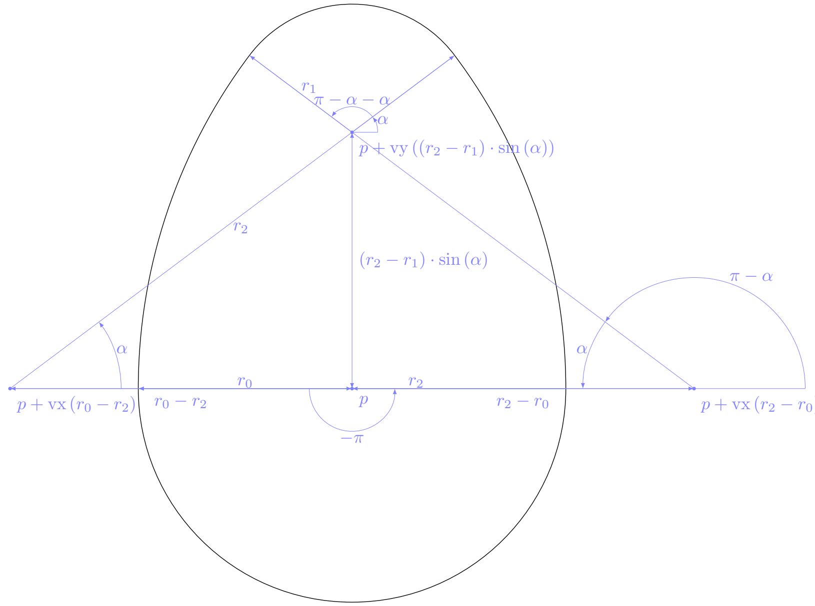


Fig. 4: Two definitions of an egg shape, either calculating the center points for the arcs directly within the arc calls (left) or defining the center points as local variables (right). The illustration below corresponds to the first definition.

If the illustration remains cluttered, users can also choose to illustrate a limited number of steps. As an example, consider the recursive definition of a spiral in Fig. 6. On the left, we see an unconstrained illustration with increasing transparency. On the right, we only illustrate the first recursive step. Other mechanisms, such as the step-by-step illustration option presented above, may be equally useful to illustrate the execution of a recursive program.

In most cases, our approach is enough to get an idea of the process and understand where the errors lie when the program is not producing the expected result. Take, for instance, the recursive diamond-shaped pattern in Fig. 7. The leftmost definition contains a bug that is easily perceived in the illustration. The one on the right is correct. Either way, illustrations might require style decisions on the user's part, as the system's default behavior is unlikely to suit all cases.

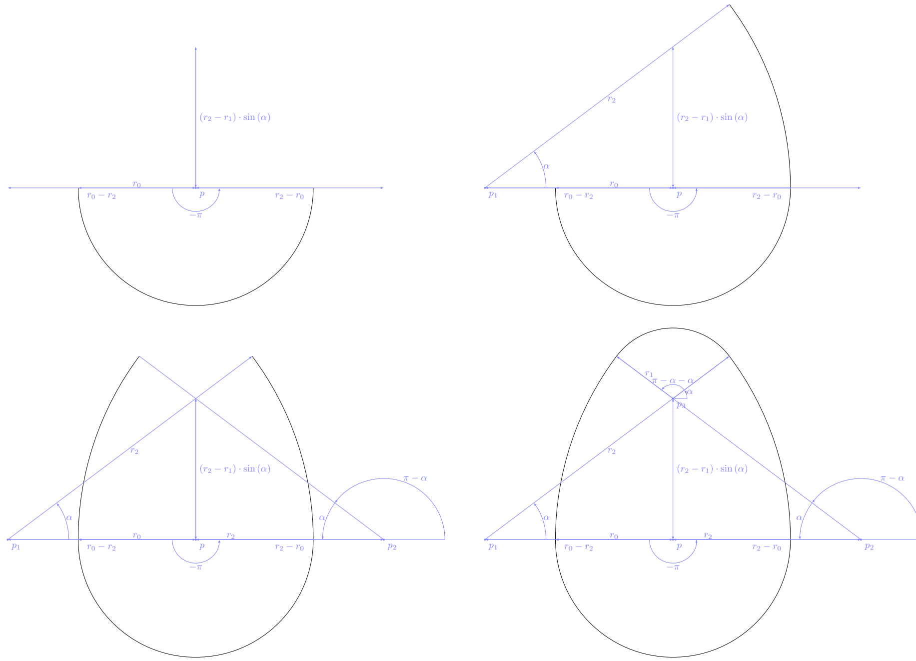


Fig. 5: Step-by-step illustration of the egg, using the second definition in Fig. 4. The arc centers are annotated with the names of the corresponding local variables.

5 Discussion and follow-ups

Explanations are all about simplifying or breaking complex problems down into smaller bits. In architecture, we frequently resort to simplified 2D plan or section depictions, 2D line-based schematics, etc. to explain a complex design idea in simpler terms. Capitalizing on this concept, the proposed system focused on the creation of 2D geometric illustrations explaining the behavior of AD program fragments, based on the operations and parameters used in it.

We foresee two main scenarios that strongly benefit from this proposal: (1) collaborative work endeavors involving shared AD programs, and (2) learning environments, including not only novice programmers trying to get a grip on their creations, but also teachers and learning-content creators.

Withal, the research presented here is a work in progress, with the potential to integrate more ideas. There are many fields left to explore. For instance, AD programs do not describe geometry modeling operations only. They frequently integrate descriptions of simulation and optimization routines. Graphically explaining these routines and subsequent processing of the output data is an en-

```
spiral_arc(p, r, α, Δα) = arc(p, r, α, Δα)

spiral(p, r, α, Δα, ω, f) =
    if ω-α < Δα
        spiral_arc(p, r, α, ω-α)
    else
        spiral_arc(p, r, α, Δα)
        spiral(p + vpol(r*(1 - f), α+Δα), r*f, α+Δα, Δα, ω, f)
    end
```

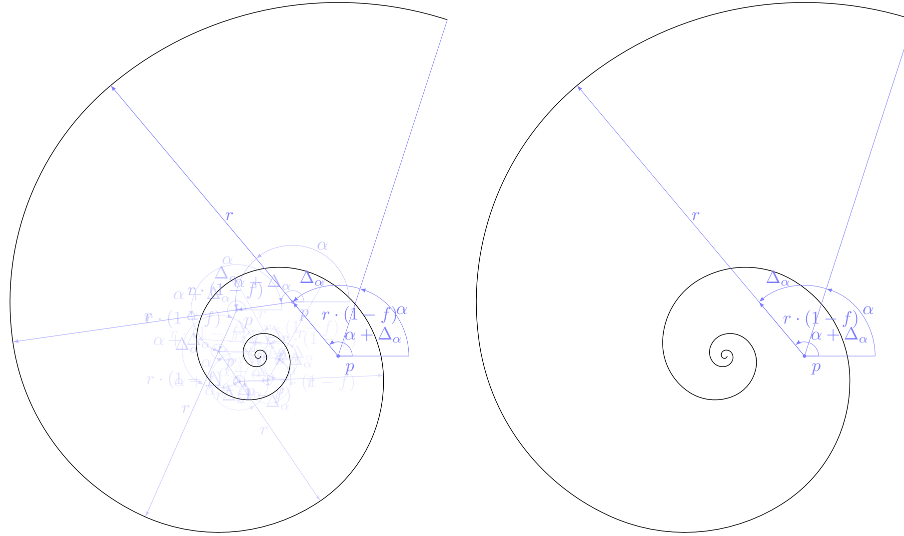
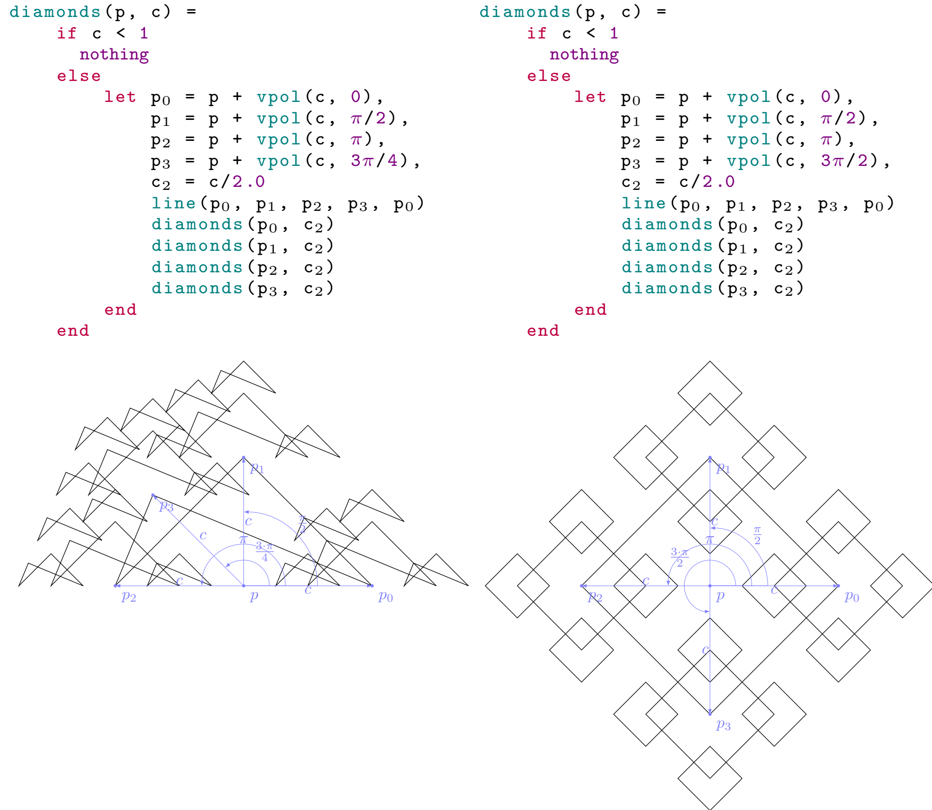


Fig. 6: Recursive definition of a spiral (top), with an unconstrained illustration of the entire process (left), and an illustration limited to the first recursive step (right).

tirely new illustration challenge. We leave below some unexplored research paths that we believe to be logical next steps in this investigation.

Geometric constraints In in the Petronas example (Fig. 1), the radius that guarantees the *tangency* between the circles and the octagon was mathematically calculated in the program without a single reference to either the octagon or the term 'tangent'. As a result, since it is not explicitly expressed in the program, this geometric relation cannot easily be inferred by the evaluator either. We made the octagon visible using the mechanisms available in the illustrator to add additional information to the images. However, it remains a far cry from an expressive illustration of geometric constraints. Research work has been done on ways to explicitly state these relations in the program in order to facilitate the associated calculations, prevent errors, and make the program more intelligible [34]. Building upon these principles, we plan to extend our illustration library to include geometric constraint concepts.



```
diamonds(p, c) =
    if c < 1
      nothing
    else
        let p₀ = p + vpol(c, 0),
            p₁ = p + vpol(c, π/2),
            p₂ = p + vpol(c, π),
            p₃ = p + vpol(c, 3π/4),
            c₂ = c/2.0
            line(p₀, p₁, p₂, p₃, p₀)
            diamonds(p₀, c₂)
            diamonds(p₁, c₂)
            diamonds(p₂, c₂)
            diamonds(p₃, c₂)
        end
    end
```

```
diamonds(p, c) =
    if c < 1
      nothing
    else
        let p₀ = p + vpol(c, 0),
            p₁ = p + vpol(c, π/2),
            p₂ = p + vpol(c, π),
            p₃ = p + vpol(c, 3π/2),
            c₂ = c/2.0
            line(p₀, p₁, p₂, p₃, p₀)
            diamonds(p₀, c₂)
            diamonds(p₁, c₂)
            diamonds(p₂, c₂)
            diamonds(p₃, c₂)
        end
    end
```


Fig. 7: Recursive placing of diamond shapes. The illustration of the program on the left shows that the angle used to calculate the position of point $p3$ is wrong. The definition on the right is correct.

Organizing labels The proposed mechanism to infer if there is any information juxtaposition currently considers label insertion points only. For most geometric elements this approach will suffice. However, if the illustration contains long labels, the chances of juxtaposition increase and the system is none the wiser. We could further develop the existing label placement algorithm to consider the bounding boxes of previously generated labels as well.

Illustration visualizer In this implementation, we used a CAD tool (AutoCAD) and a dedicated drawing program (TikZ) to generate the illustrations to be inserted into the Pluto computational notebook. Since this insertion is not literal, but rather a reference to a file in a folder, updating the illustration does not require a new insertion; if we re-generate the image with the same name, Pluto will fetch the updated version automatically. We could, nevertheless, explore other options for image generation, such as browser-based visualizers that

produce the graphs directly in the notebook (e.g., Plotly). A downside to this approach is that users have to effectively run the program to visualize the illustrations, whereas the approach we chose to pursue keeps a version of the illustrations available for anyone to see, even without executing the program.

3D The best explanations are often simplifications or depictions of isolated parts of the problem; hence, our initial focus on 2D geometry. However, we have already begun extending the system to 3D illustrations. We are currently studying label positioning techniques that consider the rendering viewpoint, automatic sectioning methods, and step-by-step decompositions particularized to some geometry modeling operations (for instance, in a sweep operation it may be interesting to show not only the extrusion of the section along the path step-by-step but also any scaling or rotation factor applied to the section in a separate animation).

6 Conclusion

This article proposed an automatic illustration system to produce annotated 2D geometric schemes explaining Algorithmic Design (AD) programs. The aim was to improve the comprehension, and thus, learning, development, maintenance, and sharing of AD programs among architects.

The illustrations comprise a set of basic geometric elements, such as points, distances, angles, vectors, etc., which can help users understand the geometric calculations done in the AD program, the meaning of the symbols used, and the impact they have on the overall geometry if changed. The proposed system allows architects to piggyback the illustration process on top of AD programs, making them generate not only the intended architectural models but also the illustrations explaining them. The resulting illustrations are then integrated into the AD program, intertwined with the instruction they intend to explain.

The proposal builds upon previous work regarding the inclusion of visual documentation in AD programs, now contemplating two types of documentation: hand-made drawings and computer-generated illustrations. Comparing one with the other at any stage of the process can also help users understand if the program is producing the expected results.

This research addresses and considerably simplifies the hardworking task of illustrating AD programs, frequently automating it completely. Nevertheless, this is ongoing research and several logical follow-ups to this proposal were discussed, such as extending the illustration library to include more complex concepts like geometric constraints, a more holistic approach to annotation placement that is sensitive to the space occupied by pre-existing labels, considering other visualizers for the generation of the images, and extending the system to 3D illustrations.

Acknowledgments This work was supported by national funds of *Fundação para a Ciência e a Tecnologia* (FCT) with references UIDB/50021/2020, PTDC/ART-DAQ/31061/2017, and DFA/BD/4682/2020.

References

1. Alfaiate, P., Caetano, I., Leitão, A.: Luna Moth: Supporting creativity in the cloud. In: Proceedings of the 37th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA). pp. 72–81. Cambridge, Massachusetts, USA (2017)
2. Brown, M.H., Najork, M.A.: Algorithm animation using 3D interactive graphics. In: Proceedings of the 6th annual symposium on User Interface Software and Technology (UIST'93). pp. 93–100. ACM (1993)
3. Burry, M.: Scripting Cultures: Architectural Design and Programming. Architectural Design Primer, John Wiley & Sons, Inc. (jan 2013). https://doi.org/10.1002/9781118670538
4. Caetano, I., Garcia, S., Pereira, I., Leitão, A.: Creativity inspired by analysis: An algorithmic design system for designing structurally feasible façades. In: Proceedings of the 25th International Conference of the Association for Computer-Aided Architectural Design Research in Asia (CAADRIA). vol. 1, pp. 599–608. Bangkok, Thailand (2020)
5. Caetano, I., Santos, L., Leitão, A.: Computational design in architecture: Defining parametric, generative, and algorithmic design. Frontiers of Architectural Research **9**(2), 287–300 (2020). https://doi.org/10.1016/j.foar.2019.12.008
6. Carpo, M.: The Alphabet and the Algorithm. The MIT Press, 1st edn. (2011)
7. Castelo-Branco, R., Caetano, I., Leitão, A.: Digital representation methods: The case of algorithmic design. Frontiers of Architectural Research **11**(3), 527–541 (2022). https://doi.org/10.1016/j.foar.2021.12.008
8. Castelo-Branco, R., Caetano, I., Pereira, I., Leitão, A.: Sketching algorithmic design. Journal of Architectural Engineering **28**(2), 04022010 (2022). https://doi.org/10.1061/(ASCE)AE.1943-5568.0000539
9. Castelo-Branco, R., Leitão, A.: Comprehending algorithmic design. In: Computer-Aided Architectural Design. Design Imperatives: The Future is Now, pp. 15–35. CAAD Futures 2021, Singapore (2022). https://doi.org/10.1007/978-981-19-1280-1_2
10. Davidson, S.: Grasshopper: Algorithmic modeling for rhino (2023), https://www.grasshopper3d.com, last accessed 2023/02/14
11. Diehl, S.: Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software. Springer Berlin Heidelberg (2007)
12. Goldschmidt, G.: Design representation: Private process, public image. In: Goldschmidt, G., Porter, W.L. (eds.) Design Representation, pp. 203–217. Springer London, London (2004). https://doi.org/10.1007/978-1-85233-863-3_9
13. Goldstine, H.H., Von Neumann, J.: Planning and coding of problems for an electronic computing instrument: Report on the mathematical and logical aspects of an electronic computing instrument (1947)
14. Jackson, M.A.: Principles of Program Design. Academic Press, Inc., USA (1975)
15. Kelly, N., Gero, J.S.: Design thinking and computational thinking: A dual process model for addressing design problems. Design Science **7**(May), 1–15 (2021). https://doi.org/10.1017/dsj.2021.7
16. Kery, M.B., Myers, B.: Exploring exploratory programming. In: Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 25–29. IEEE (2017). https://doi.org/10.1109/VLHCC.2017.8103446
17. Knuth, D.E.: Literate programming. The Computer Journal **27**(2), 97–111 (May 1984). https://doi.org/10.1093/comjnl/27.2.97

18. Leitão, A., Lopes, J., Santos, L.: Illustrated programming. In: Proceedings of the 34th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA). pp. 291–300. Los Angeles, California, USA (2014)
19. Nassi, I., Shneiderman, B.: Flowchart techniques for structured programming. SIGPLAN Notices **8**(8), 12–26 (Aug 1973). https://doi.org/10.1145/953349.953350
20. Nguyen, A.T., Reiter, S., Rigo, P.: A review on simulation-based optimization methods applied to building performance analysis. Applied Energy **113**, 1043–1058 (2014). https://doi.org/https://doi.org/10.1016/j.apenergy.2013.08.061
21. Oppenheimer, N.: An enthusiastic sceptic. Architectural Design **79**(2), 100–105 (2009). https://doi.org/https://doi.org/10.1002/ad.862
22. Picon, A.: Architecture and the virtual: towards a new materiality? Praxis Journal of Philosophy pp. 114–121 (2004)
23. Picon, A.: Beyond digital avant-gardes: The materiality of architecture and its impact. Architectural Design **90**(5), 118–125 (2020). https://doi.org/https://doi.org/10.1002/ad.2618
24. van der Plas, F., Bochenski, M.: Pluto.jl (2021), https://github.com/fonsp/Pluto.jl, last accessed 2023/02/14
25. Price, B.A., Baecker, R.M., Small, I.S.: A principled taxonomy of software visualization. Journal of Visual Languages & Computing **4**(3), 211–266 (1993). https://doi.org/10.1006/jvlc.1993.1015
26. Rein, P., Ramson, S., Lincke, J., Hirschfeld, R., Pape, T.: Exploratory and live, programming and coding: A literature study comparing perspectives on liveness. Programming Journal **3**(1), 1:1–1:33 (2018). https://doi.org/10.22152/programming-journal.org/2019/3/1
27. Roman, G.C., Cox, K.C.: Program visualization: the art of mapping programs to pictures. In: International Conference on Software Engineering. pp. 412–420. IEEE (1992). https://doi.org/10.1145/143062.143157
28. Ruck, A.: Abacus and sketch. In: Kara, H., Bosia, D. (eds.) Design Engineering Refocused, chap. 5, pp. 76–87. AD Smart 03, John Wiley & Sons Ltd (2017)
29. Rule, A., Tabard, A., Hollan, J.D.: Exploration and explanation in computational notebooks. In: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems. pp. 1–12. CHI '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3173574.3173606
30. Sammer, M.J., Leitão, A., Caetano, I.: From visual input to visual output in textual programming. In: Proceedings of the 24th International Conference of the Association for Computer-Aided Architectural Design Research in Asia (CAADRIA). vol. 1, pp. 645–654. Wellington, New Zealand (2019)
31. Terzidis, K.: Expressive Form: A conceptual approach to computational design. Spon Press, London and New York (2003)
32. Terzidis, K.: Algorithmic Architecture. Architectural Press, New York (2006)
33. Turrin, M., von Buelow, P., Stouffs, R.: Design explorations of performance driven geometry in architectural design using parametric modeling and genetic algorithms. Advanced Engineering Informatics **25**(4), 656–675 (2011). https://doi.org/10.1016/j.aei.2011.07.009
34. Ventura, R.: Geometric Constraints in Algorithmic Design. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal (2021)
35. Wang, A.Y., Mittal, A., Brooks, C., Oney, S.: How data scientists use computational notebooks for real-time collaboration. Proceedings of the ACM on Human-Computer Interaction **3** (2019). https://doi.org/10.1145/3359141