VISUAL MEETS TEXTUAL

A Hybrid Programming Environment for Algorithmic Design

RENATA CASTELO-BRANCO[1] and ANTÓNIO LEITÃO[2]
[1,2]*INESC-ID, Instituto Superior Técnico, Universidade de Lisboa*
[1,2]*{renata.castelo.branco|antonio.menezes.leitao}@tecnico.ulisboa.pt*

Abstract. Algorithmic approaches are currently being introduced in many areas of human activity and architecture is no exception. However, designing with algorithms is a foreign concept to many and the inadequacy of current programming environments creates a barrier to the generalized adoption of Algorithmic Design (AD). This research aims to provide architects with a programming tool they feel comfortable with, while allowing them to fully benefit from AD's advantages in the creation of complex architectural models. We present Khepri.gh, a hybrid solution that combines Grasshopper, a visual programming environment, with Khepri, a flexible and scalable textual programming tool. Khepri.gh establishes a bridge between the visual and the textual paradigm, offering its users the best of both worlds while providing an extra set of advantages, including portability among CAD, BIM, and analysis tools.

Keywords. Algorithmic Design; Hybrid Programming Environment; Textual Programming; Visual Programming.

1. Introduction

Algorithmic Design (AD) is a method increasingly present in the architectural practice, which entails the creation of designs through algorithmic descriptions, allowing the designer to delegate repetitive tasks to the computer, accelerating the production process, reducing human errors (Burry, 2011) and allowing considerable cost savings (Woodburry, 2010).

However, designing with algorithms is a foreign concept to many (Terzidis, 2006) and, moreover, it requires representation methods that radically differ from those used in architecture (Maleki & Woodbury, 2013), thus creating a mismatch that is further exacerbated by the inadequacy of current Integrated Development Environments (IDEs). This hinders the adoption of AD, demotivating architects from its use and, thus, limiting the potential benefits.

2. Programming Paradigms

Given the advantages AD brings to the practice, many IDEs have been developed. Despite multiple attempts by different communities to develop means for

RE: Anthropocene, Proceedings of the 25th International Conference of the Association for Computer-Aided Architectural Design Research in Asia (CAADRIA) 2020, Volume 1, 375-384.

describing computation, two main paradigms stand out: (1) Visual Programming Languages (VPLs), which simplify the learning process but lack scalability (Bentrad & Meslati, 2011), i.e., as programs grow in complexity, they become hard to understand and navigate (Leitão, et al., 2012); and (2) Textual Programming Languages (TPLs), which scale better with larger programming projects but usually entail a steeper learning curve and are less attractive for the innate visual nature of architects (Sammer, et al., 2019).

2.1. VISUAL PROGRAMMING

In Visual Programming (VP), users can specify programs in a two or more-dimensional fashion (Zhang, 2007). Elements like spatial relationships, time, or visual expressions such as diagrams and sketches can all be used to convey the semantics of the instructions (Burnett, 199). TPLs do not qualify as VP, since they are compiled or interpreted as long one-dimensional streams (Myers, 1990).

Strategies for VP include, for instance, concreteness (focus on particular instances) and explicitness (little inference required) - two features that make VP appealing at start by guaranteeing a smooth learning curve but which deter programmers from its use at large scales (Burnett, 199). This lack of abstraction mechanisms has long led to the conclusion that the benefit of using VP is inversely proportional to the size and complexity of the problem (Whitley, 1997).

Furthermore, the efficacy of VP systems is intimately dependent on the context and task to be performed. Despite largely reducing their ideal field of application (Whitley, 1997), this also makes them an important puzzle piece to end-user programming in specific domains, which is the case of architecture. In proof of this stand languages like Grasshopper (GH), Dynamo or Generative Components, who have had stunning success among this community.

2.2. TEXTUAL PROGRAMMING

For general-purpose programming by professional programmers, Textual Programming (TP) is the appropriate and most favored choice (Myers, 1990). While VP diagrams tend to rapidly overflow the bounds of the screen (Nardi, 1993), TP, on the other hand, supports higher information density with abstraction and filtering mechanisms. The problem of TP lies in diametrically opposed concepts to those which lower VP's utility: the steep learning curve and the difficulty architects face in understanding AD programs and relating them to the building's concept.

The human brain is inherently visual and multi-dimensional (Zhang, 2007) and, despite the proven advantages of this approach, graphics are, in fact, closer to the user's mental representation of problems. Especially for non-expert or novice programmers, graphical representations make the programming task easier to understand (Myers, 1990). Specifically, in the architectural context, users must translate mental pictures into textual representations, which is a considerable challenge even for the most gifted (Boshernitsan & Downes, 2004). This suggests that textual programming alone also fails to satisfy our needs for all possible scenarios.

2.3. HYBRID PROGRAMMING

Hybrid programming solutions have also been developed, in an attempt to join the best of both worlds. Different classifications of hybrid programming consider both (1) programs that are created visually and then translated into an underlying textual language (Boshernitsan & Downes, 2004; Nardi, 1993), and (2) the use of VP environments to write textual code (Burnett, 2001). Within the specific domain of architecture, there are several examples of hybrid systems.

For the first case, there is Programming In the Model (PIM) (Maleki & Woodbury, 2013), whose interface offers three interactive live windows showing the model, the dependency graph (visual), and the script (textual). Users can write a program both in a VPL and a TPL and the tool converts between these paradigms: a twist on option number one. Unfortunately, this solution requires substantial computational power in order to scale.

Möbius (Janssen, et al., 2016) is a parametric modeler for the web and, although the authors consider this to be a VPL system, the interface is definitely hybrid. It presents a flowchart area where users develop networks of nodes and wires in an associative/data flow programming style. Each of these nodes corresponds to a set of procedures defined in imperative/code-block style. When run, the entirety of the program is converted into JavaScript for execution.

For the second case, and once more despite being considered a VPL, Dynamo provides users with the possibility to code in miniature text-scripting interfaces within the visual IDE as well, using DesignScript, the associative programming language at the heart of Dynamo and, more recently, Python.

3. Hybrid Programming Environment

Neither the visual nor the textual paradigm seem to suffice for the complex task of developing AD programs. Furthermore, current hybrid systems do not always succeed in gathering the best of both nor at providing the learning opportunity lying underneath. This research aims to provide architects with a programming tool they feel comfortable with, while allowing them to fully benefit from AD's advantages in the creation of complex architectural models. However, instead of developing new VPLs or TPLs, we rely on already established ones, facilitating the learning process. Thus, we present a hybrid solution that combines GH, a popular VP environment, with Khepri, a flexible and scalable textual programming tool - Khepri.gh.

3.1. TOOLS

Khepri (Sammer, et al., 2019) is a TP tool based on the idea that a single algorithmic description can be used to generate equivalent models in different backends, such as Computer-Aided Design (CAD), Building Information Modelling (BIM), analysis, and gaming applications. Khepri is currently implemented in Julia (Bezanson, et al., 2017), a modern programming language with a smooth learning curve, fast execution, and support for large-scale development.

GH is a well-known VP tool tightly integrated with Rhinoceros. As a VPL, GH

requires no prior knowledge of programming, which is a very attractive quality for architects with little scripting experience. However, as most VPLs, this one also lacks scalability, as the abstraction mechanisms that help manage complexity are at fault in this paradigm. GH offers several solutions to bypass this issue, which resort to textual programming, namely in C#, Visual Basic (VB) or Python.

3.2. IMPLEMENTATION

Khepri.gh's implementation is in many ways similar to GH's currently available textual scripting editors: users drag a blank Khepri component onto the canvas (Figure 1A), open up the editor box by double-clicking a component and develop Julia code within (Figure 1B); they can then save the new components onto a dedicated tab (Figure 1C).

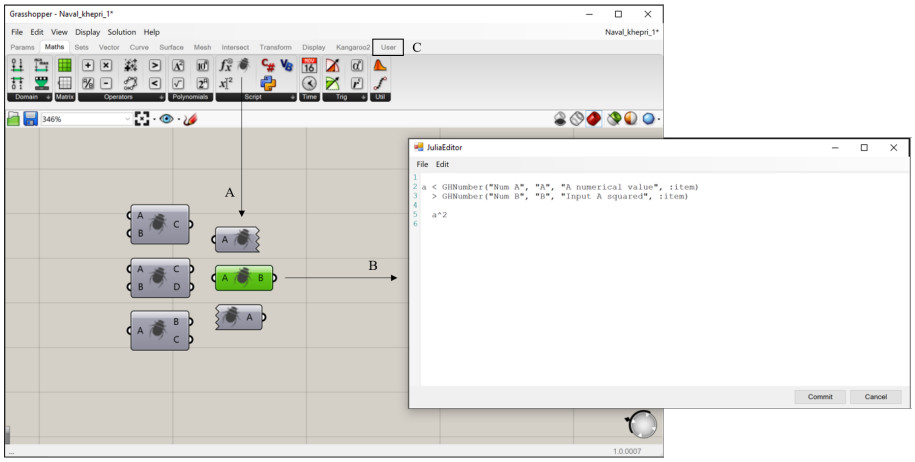

Figure 1. A – Khepri components on canvas with various inputs and/or outputs; B – Julia editor showing a component's code; C – User tab, where new components are saved.

However, while the C#, VB, and Python scripting editors only provide the features of each programming language unless other libraries are loaded, our implementation not only integrates the Julia language but also the Khepri programming tool from the start. This means, that besides the language primitives, users have access to the modeling primitives available in Khepri, which are transversal to the multiple backends.

3.3. PROGRAMMING STAGES

In order to better illustrate this workflow, we defined three main stages of program development where the differences between the typical GH approach and the Khepri.gh approach become visible. Phase one (1) includes *simple calculations*, the definition of mathematical functions, mapping locations, etc., all of which are typically achieved using native GH components. In the Khepri.gh workflow, these can also be done using Khepri components. Phase two (2) concerns the construction of more *abstract computations*, such as recursive and iterative

procedures, or high-order functions, which are typically implemented with the aid of textual scripts. Phase three (3) involves *geometry generation* and any other sort of operation realization in the connected applications. In GH's case, this is usually done with native components, which provide immediate visual feedback in Rhinoceros and that can be baked in the end. In Khepri.gh, this stage would have to be done with Khepri components forcefully if the geometry is to be portable amongst backends.

Figure 2 presents a graphical representation of these three phases organized temporally according to the most common strategy of program development. The last phase is transversal to all others, as it regards the programming environment itself. In GH's workflow, we are naturally using the IDE's canvas to program through the entire process. With Khepri.gh's, we are also given the possibility to visualize the entirety of the textual part of the program in a dedicated textual environment. This feature will be further discussed later on.

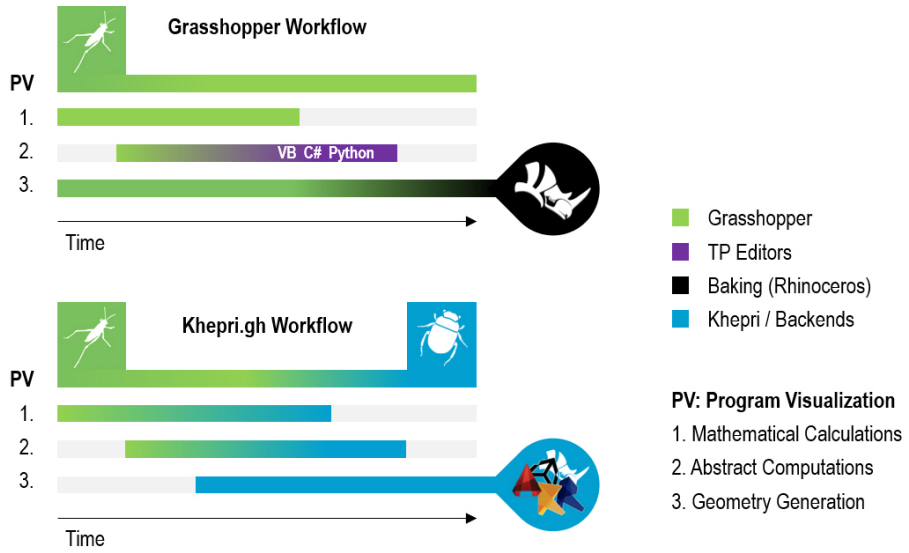

Figure 2. Workflow comparison, considering the three development stages, and the tools involved in the process (Grasshopper, Textual Programming (TP) editors, Rhinoceros, Khepri and its multiple backends).

4. User-friendly Features

Khepri.gh's implementation focuses on three main features designed to facilitate the coding task for non-expert programmers that, typically, find it difficult to understand complex programs and the impact of the changes applied to them: traceability, immediate feedback, and different code editing options.

To illustrate these features, in the ensuing subsections we present a project for a Nautical Center in Lisbon, a building projected onto the Tagus River. The Center is divided into three main blocks, anchored at the pier, which are then interconnected

by two other volumes lifted in the air above the boat launch ramps. Figure 3 shows rendered images of this project, along with a physical model. In the Khepri.gh model of the building, most of the geometrical constraints defining the shape of the three volumes are variable. Some of them can be seen in Figure 4.

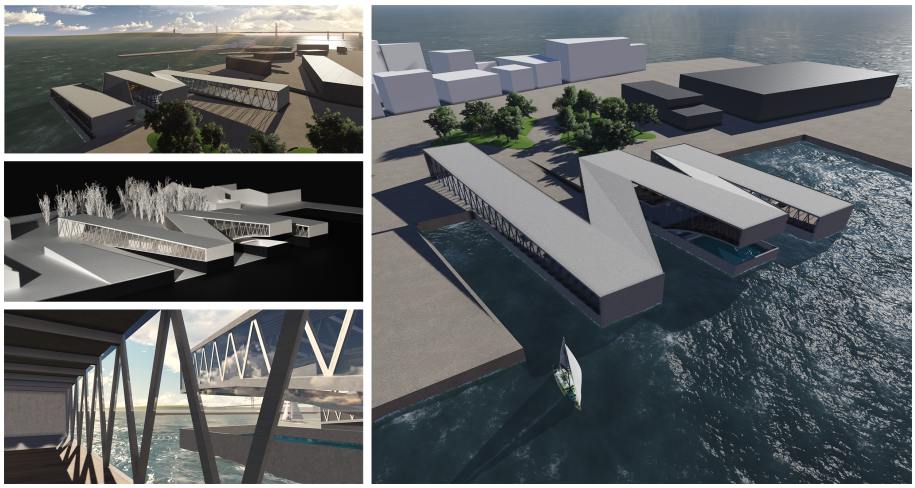

Figure 3. Case Study: Nautical Center renders and physical model.

4.1. TRACEABILITY

Traceability entails the identification of which parts of the model correspond to which parts of the program, and vice-versa (Leitão, et al., 2014), a crucial correlation to understand, maintain, and debug the program. While GH presents unidirectional traceability, only relating program components to model elements, Khepri.gh supports bi-directional traceability and in multiple backends. Figure 5 shows the traceability feature in the AutoCAD and Unity backends.

4.2. IMMEDIATE FEEDBACK

Immediate feedback is the ability to re-compute changes to the program's input or to the program itself in (near) real time (Rauch, et al., 2019), allowing designers to easily understand the program's behavior (Alfaiate, et al., 2017). GH immediately updates the geometry upon adding/changing components, Boolean toggles or sliders. Our solution extends this feature to the various backends but performance must be taken into consideration when dealing with complex models. Scale is already an issue for GH's optimized preview mode (i.e., non-baked geometry); even more so for Khepri.gh, which has to generate the geometry in the backends, a somewhat equivalent process to baking. This process is considerably faster in more performative backends, such as game engines, or using CAD tool's more performative view options, such as wireframe models. Figure 4 shows several variations of the Nautical Center generated by moving the project's sliders.

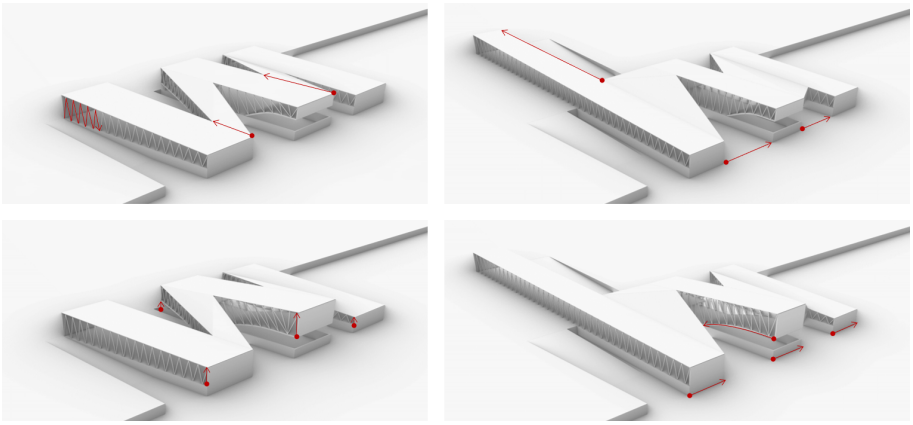

Figure 4. Screenshots from Rhino showing variations of the Nautical Center. The red arrows represent the variable parameters being changed in the model.

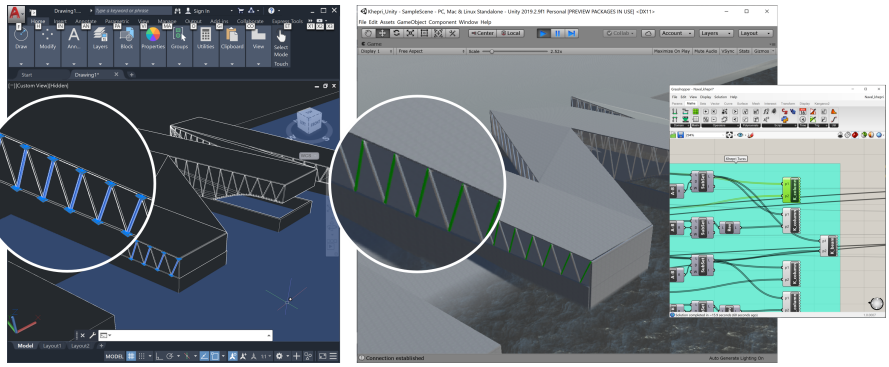

Figure 5. Traceability in different backends: GH program on the right showing the selected component, and respective representations automatically highlighted in AutoCAD, and Unity.

4.3. CODE EDITING OPTIONS

Khepri.gh supports different code editing options (Figure 6). In the visual IDE the user can (a) manipulate existing GH components and (b) create new Khepri components, each one representing a parcel of code, using the textual programming editor available within GH. However, for more experienced programmers, (c) the code can also be accessed in a textual programming editor, such as Atom.

5. The Power of Abstraction

Having toured around the visual benefits of the proposed solution, we now peer into the advantages brought about by the textual paradigm.

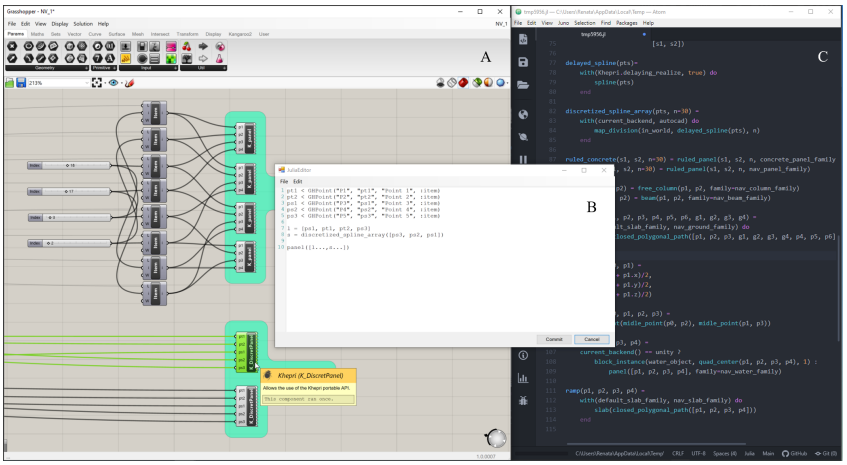

Figure 6. Code editing options: (a) visual, (b) textual within visual, and (c) full textual.

5.1. NO REPETITION

Following GH's native structure, VB, C# or Python scripting motivates the construction of standalone components, which both implement and execute operations. When reusing such components, the code is essentially duplicated and further changes to that structure must be performed in all similar components. Textual programming follows a different logic, where repetition is avoided at all costs for logic and organizational purposes. This makes the code structure more comprehensible and manageable, sparing the user the time-consuming and error-prone task of maintaining several versions of the same functionality.

Khepri.gh follows the textual logic in this issue, introducing a fundamental separation between function definitions and function calls. All components on canvas are processed on demand, with or without inputs, with or without specific modeling instructions. Users may gather function definitions, in one or more components, that can be used in all the others that call them, thus eliminating repeated definitions. When the user edits a definition component, the changes will have automatic repercussions in any other components dependent upon this one.

5.2. COMPLEXITY

The scalability issue inherent to VPLs affects many of the features that make these systems so appealing to users in earlier stages (Aish, 2003): both immediate feedback and program readability are compromised as complexity increases. For the first issue, GH cleverly offers a solution unsurprisingly inspired in the way TPLs deal with geometry generation: inputting numbers directly onto sliders instead of dragging them, or even disabling the solver as we edit the program, thus significantly reducing the number of times the model is recomputed.

For the second problem, the most obvious solution is abstraction: the possibility to encapsulate complex computations in much simpler program

structures, such as recursive and iterative first-order or higher-order functions. These mechanisms allow the development of more complex programs without resorting to the repetition of instructions. However, most VPLs, including GH, do not (directly) support these functionalities. Since Khepri.gh is built on top of the Julia language, any Julia functionality can be used from GH as well, meaning that the data that flows between components includes not only basic data structures, such as numbers and positions, but also functions, arrays, tuples, dictionaries, etc, and arbitrary combinations of these.

6. Discussing Transition

For all the advantages of abstraction, we get pulled further away from concreteness and explicitness, which helped smooth the learning curve of programming. The question then becomes: where should one draw the line between visual and textual? In this case, when should users shift from GH to Khepri components, relinquishing the ease of use in exchange for computational power? And should the answer somehow relate to one's programming capabilities?

To elaborate on this debate, we circle back to the scheme presented in Figure 2, where we defined three main phases of program development. After the presented experiments with the prototype, our own opinion reverts to the conclusion that phase (1) should be performed using mainly GH components by users with more experience using this IDE than any other. Provided they use Khepri components for phase (3) they will still benefit from the main advantages brought about by this approach. Users that find themselves limited by the existing GH components may also be motivated to learn how to create new Khepri components in phase (2). This solution is clearly the one where the least effort is required to program, and it represents a soft adaptation curve to Khepri's framework.

More experienced programmers may prefer to model Khepri components from the start. By developing phases (1), (2), and (3) on a Khepri base, users can access the entirety of their code in a textual programming environment as well, which provides a good incentive to the visual-to-textual transition. But, while developing Khepri components within GH's IDE, users can also benefit from the graphical properties provided by the environment, which facilitate the comprehension task.

In the end, we believe Khepri.gh can also serve as a smooth bridge for architects who which to transition from the visual to the textual paradigm. The dichotomy between the two has for long represented a gap in the learning curve, with no visible progression from one to the other. Our solution thus also hopes to provide the means for expert VPL users to transition to the textual paradigm without having to restart the learning process from scratch. Having all of Khepri's operations available for use in a platform that architects feel comfortable using may help them get acquainted with the tool before switching to a textual IDE entirely.

7. Conclusion

Khepri.gh is a hybrid programming environment that combines Grasshopper, a popular visual programming environment, with Khepri, a flexible and scalable textual programming tool for algorithmic design. Exploiting the potential

of hybrid environments to serve as the middle ground between visual and textual programming languages, Khepri.gh provides users with a comfortable user-friendly visual interface while guaranteeing the portability, scalability, and complexity-handling mechanisms offered by the textual paradigm.

While the evaluation here presented focused mainly on Khepri's functionalities, as future work, we plan on further exploring the proposed benefits of the abstraction mechanisms provided by the Julia programming language within the Grasshopper environment.

Acknowledgments

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references UIDB/50021/2020 and PTDC/ART-DAQ/31061/2017.

References

Aish, R.: 2013, DesignScript: Scalable Tools for Design Computation, *Proceedings of eCAADe 2013*, Delft, The Netherlands, 18-20.

Alfaiate, P., Caetano, I. and Leitão, A.: 2017, Luna Moth: Supporting Creativity in the Cloud, *Proceedings of ACADIA 2017*, Massachusetts, USA, 72-81.

Bentrad, S. and Meslati, D.: 2011, Visual Programming and Program Visualization: Towards an Ideal Visual Software Engineering System, *Journal on Information Technology*, **1**(3), 56-62.

Bezanson, J., Edelman, A., Karpinski, S. and Shah, V.B.: 2017, Julia: A Fresh Approach to Numerical Computing, *SIAM Review*, **59**(1), 65-98.

Boshernitsan, M. and Downes, M.S.: 2004, Visual programming languages: A survey, *EECS Department, University of California, Berkley*, 25.

Burnett, M.: 2001, Visual programming, *Wiley Encyclopedia of Electrical and Electronics Engineering*, 275-283.

Burry, M.: 2011, *Scripting Cultures*, John Wiley & Sons Ltd., United Kingdom.

Jassen, P., Li, R. and Mohanty, A.: 2016, Mobius: A Parametric Modeller for the Web, *Proceedings of CAADRIA 2016*, Melbourne, Australia, 157-166.

Leitão, A., Lopes, J. and Santos, L.: 2014, Illustrated Programming, *Preceedings of ACADIA 2014*, Los Angeles, USA, 291-300.

Leitão, A., Santos, L. and Lopes, J.: 2012, Programming Languages For Generative Design: A Comparative Study, *International Journal of Architectural Computing*, **10**(1), 139-162.

Maleki, M.M. and Woodbury, R.: 2013, Programming In The Model: A New Scripting Interface for Parametric CAD Systems, *Proceedings of ACADIA 2013*, Cambridge, Canada, 191-198.

Myers, B.: 1990, Taxonomies of Visual Programming and Program Visualization, *Journal of Visual Languages & Computing*, **1**(1), 977-123.

Nardi, B.A.: 1993, *A Small Matter of Programming: Perspectives on End User Computing*, MIT Press, Cambridge, USA.

Rauch, D., Rein, P., Ramson, S., Lincke, J. and Hirschfeld, R.: 2019, Babylonian-style Programming - Design and Implementation of an Integration of Live Examples Into General-purpose Source Code, *Programming Journal*, **3**, 9.

Sammer, M.J., Leitão, A. and Caetano, I.: 2019, From Visual Input to Visual Output in Textual Programming, *Proceedings of CAADRIA 2019*, Welligton, New Zealand, 645-654.

Terzidis, K.: 2006, *Algorithmic Architecture*, Architectural Press, Oxon and New York.

Whitley, K.N.: 1997, Visual Programming Languages and the Empirical Evidence For and Against, *Journal of Visual Languages & Computing*, **8**(1), 109-142.

Woodbury, R.: 2010, *Elements of Parametric Design*, Routledge, Oxon.

Zhang, K.: 2007, *Visual Languages and Applications*, Springer Science, New York.