Visual Input Mechanisms in Textual Programming for Architecture

Maria João Sammer[1], António Leitão[2]
[1,2]INESC-ID/IST
[1,2]{maria.joao.sammer|antonio.menezes.leitao}@tecnico.ulisboa.pt

Algorithmic Design (AD) is no longer foreign to architecture and its methodology embraces one of the most recent technological revolutions in the field. This approach lays on Programming Languages (PLs) to define rules and constraints within an algorithm that, in return, generates geometry in modeling and analysis tools. PLs can either be visual (VPLs) or textual (TPLs). In architecture, there is a clear propensity to the use of VPLs over TPLs, due to all the visual features and mechanisms they provide that make programming more intuitive for architects. Nevertheless, and even though TPLs are less appealing to learn and use, they offer clear advantages when dealing with complex programs. Therefore, in order to bring TPLs closer to their users, we discuss, explore, and implement Visual Input Mechanisms (VIMs) in Khepri, a new textual programming tool for architecture.

Keywords: *Algorithmic Design, Visual Input Mechanisms, Visual Programming Languages, Textual Programming Languages, Metaprogramming, Khepri*

1. INTRODUCTION

Algorithmic Design (AD) is a recent technological revolution in the field of architecture, being an alternative approach to the manual generation and manipulation of geometry within modeling and analysis tools. Therefore, instead of creating geometry directly in these tools, architects create a program that, when executed, generates the correspondent model in a chosen tool. This program is a set of rules and constraints defined using Programming Languages (PLs), that can either be visual (VPLs) or textual (TPLs).

VPLs use icons and connections to feed information and instructions to the computer, structuring diagrams of blocks connected by wires (Schaefer 2011) (figure 1). According to Noone and Mooney (2018), a VPL is any PL that allows users to manipulate the underlying code graphically, thus specifying the execution of a program without textual scripting (Menzies 2002). In a VPL, programs consist of icons that can be manipulated interactively and according to some spatial grammar (Myers 1990), i.e. iconic elements of data - that can contain either values, functions, or geometry, for example, related between each other through output-input relations.

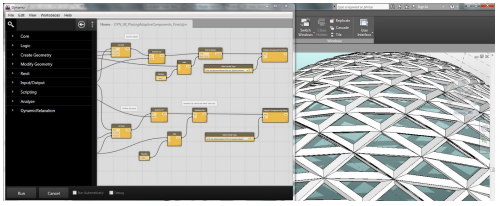

Figure 1
Program written in Dynamo, a VPL [3].

Contrarily to VPLs, textual programming languages, as the name suggests, belong to the category of PLs that use written text to build programs (figure 2), using a one-dimensional stream of characters (Brown and Kimura 1994). These programs structure a script of written instructions and descriptions that the computer understands and processes unequivocally. Just like any other language in the world, TPLs also obey specific syntax and semantics.

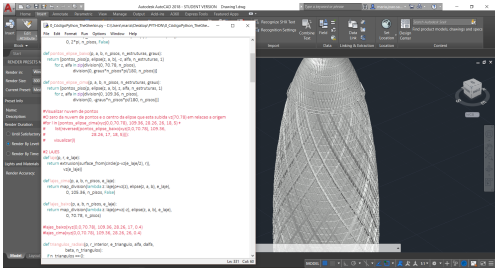

Figure 2
Program written in Python, a TPL.

The relative advantages and disadvantages between VPLs and TPLs have already been discussed in several studies (Leitão and Santos 2011, Davis et al. 2011, Leitão et al. 2012, Janssen 2014, Zboinska 2015). However, in the current practice of architecture, there is a clear propensity to the use of VPLs over TPLs, due to the user-friendly features and interactive mechanisms they provide that are absent in most textual approaches. Nevertheless, TPLs offer clear benefits when dealing with complex programs, which is a strong argument for the implementation of similar features, such as Visual Input Mechanisms (VIMs), in order to bring them closer to architects.

In this investigation, we assess Grasshopper as a representative example of a VPL and Julia within Khepri representing the textual approach.

2. AD APPROACH: VPLS VS TPLS

The overall tendency for the use of VPLs is explained both by the interactivity of building a visual program and the intuitive and appealing visual features and mechanisms provided by them. To build a program in a VPL, architects simply need to drag-and-drop and then connect boxes that represent programming abstractions, such as geometry and functions, a process that is more intuitive and closer to the way we manipulate objects in the real world (Clarisse and Chang 1986). This dismisses architects from concerning about the intricacies of the implementation of these abstractions, only focusing on devising the logic of their design intentions.

Furthermore, VPLs provide a set of mechanisms that facilitate the understanding of what is being programmed, besides allowing a deeper interaction with the modeling tool. Some examples are (1) immediate feedback, to visualize in the modeling tool and in real time the geometry that is being generated and modified in the program, (2) traceability (figure 3), to highlight in the modeling tool the geometry generated by a selected part of the program, (3) number sliders and Boolean toggles, to easily change the value of a numeric or Boolean value, (4) gradients and visual graphs that can be used as inputs to the program, and (5) Visual Input Mechanisms (VIMs), to allow the use of geometric shapes as inputs to the program. Moreover, VPLs also support the use of textual code within specific abstraction boxes, extending some of the predefined features.

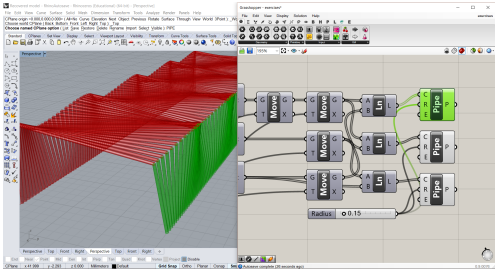

Figure 3
Grasshopper's unidirectional traceability.

Despite these advantages, VPLs present a serious handicap whenever the program becomes more complex. In fact, as the program complexity increases, not only the appealing features described tend to become less responsive, and even stop working for more extreme cases, but the visual program itself also becomes difficult to understand, manage, and maintain.

Contrastingly, TPLs are more demanding to learn and use, being less appealing and more difficult to master for less experienced users. However, they offer clear advantages when managing large-scale programs. Therefore, one of the greatest benefits of TPLs, when compared with VPLs, is the fact that the performance and legibility of a textual implementation are not as compromised with the increasing complexity of the programs created, partially due to the abstraction mechanisms that hide the program's complexity. This advantage of TPLs is a strong argument to encourage the implementation of visual mechanisms within the textual paradigm so that both advanced and less experienced users can benefit. The main goal is, thus, to make TPLs more intuitive and appealing for architects.

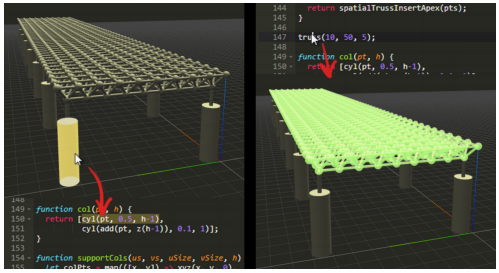

Figure 4
Luna Moth's bidirectional traceability (Alfaiate et al. 2017).

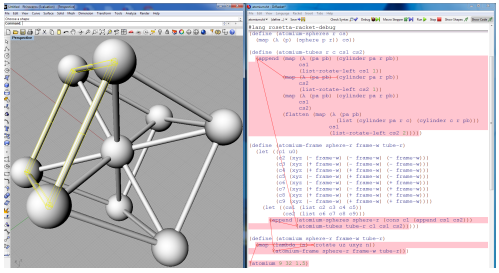

Figure 5
Rosetta's control flow visualization mechanism (Leitão et al. 2014).

Previous research already followed this path and proposed selected visual mechanisms within a textual environment. Some examples are Processing (Reas and Fry 2007), that already supports immediate feedback; Luna Moth (Alfaiate et al. 2017) that implemented both immediate feedback and bi-directional traceability (figure 4), besides providing user interaction mechanisms similar to number sliders; and Rosetta (Lopes and Leitão 2011) that extended traceability by also implementing control flow visualization mechanisms (figure 5) to visualize the sequence of instructions and decisions made by the computer while executing a program (Leitão et al. 2014). The research here presented goes even further along that path by also combining the most valuable VIMs within a TPL for AD.

3. VIMS WITHIN A VPL

VIMs are inherent to VPLs like Grasshopper and they allow the user to select previously modeled geometry from a modeling tool and use it as an input to the program. The main advantages of these mechanisms relate to (1) the use of geometry that is easier to produce manually and directly in the modeling tool, (2) the use of already existing geometry, e.g., city plans, or even (3) the use, as an input to an AD program, of geometry that was generated as an output of another AD program, for which the Morpheus Hotel [1] is a successful example.

Within the implementation of VIMs in Grasshopper, the user selects the *Set One* or *Set Multiple* options of specific abstraction boxes, the storing components representing the input geometry, and then clicks on the geometry to import directly from the modeling tool in use. Some of these components import specific types of geometry, e.g., the *Point* component solely allows to import points, while others are able to import all kinds of geometry, e.g., the *Geometry* component.

The intrinsic implementation of VIMs within VPLs imply a dependency between the geometry imported and the program that contains it, meaning that whenever that geometry is modified in the modeling tool, those changes are automatically communicated to the program that generates again the results accordingly. However, this situation may not be ideal in many cases, since the AD program becomes dependent on a specific document. This requires that both the program and the document containing the shapes imported are reopened simulta-

neously whenever the user intends to continue to work and maintain the same previous geometric results. When that independence is intended, the *Internalize Data* option becomes useful, as it enables to fasten the geometry within the component that imported it, loosing, although, its ability to be altered.

4. VIMS WITHIN A TPL

In order to implement VIMs within a textual programming context, we resorted to a TPL, Julia (Bezanson et al. 2017), integrated within Khepri, a textual programming tool for architecture. Khepri is a new AD tool inspired by Rosetta (Lopes and Leitão 2011), revising the way architects interact with it, that also allows a single textual program to generate models in different Computer-Aided Design (CAD), Building Information Modeling (BIM), and analysis tools.

A program created using Khepri represents a set of written instructions and operations, rules and constraints, that, when executed, generates models in a correspondent modeling or analysis tool. Khepri is being developed to also incorporate a set of predefined functions that, alike languages like Grasshopper, enables architects to focus on higher-level aspects of the design, instead of concerning about low-level implementation details.

Therefore, VIMs were also implemented within predefined abstractions in Khepri that, similarly to Grasshopper, when executed, ask the user to select from the modeling tool the geometry to import. Some of these predefined functions are `select_position`, `select_point`, `select_curve`, and `select_surface`, and they differ in terms of the type of geometry they import. This approach offers clear advantages not only when the programs created require an incremental or repeated selection of geometry as inputs, but also by providing immediate feedback on the choices being made.

However, and unlike Grasshopper, it is independence between program and input geometry that is assured unless told the contrary. In other words, this approach does not establish a live connection between the geometry and the program, meaning that changes in the selected shapes are not taken into account unless they are re-selected. Nevertheless, there are two possible ways to accomplish that dependence between program and imported geometry: (1) either by informing Khepri that there is a dependency on a set of shapes using the `with_shape_dependency` function, that automatically re-executes the program when that input geometry is changed, or (2) by using metaprogramming.

Metaprogramming, in the context of this investigation, encompasses the use of programs that generate other programs. In our approach, it is used to generate fragments of an AD program that represent existing visual inputs, inverting the flow of information: instead of creating a fragment within the AD program that generates geometry in a modeling tool, the architect incorporates an existing geometry from the modeling tool within a fragment of a textual program. Whenever the fragment generated through metaprogramming is executed, the same imported shape is generated in the modeling tool in use.

The relevance of metaprogramming for the implementation of VIMs is in the ability to surpass the independence inherent to functions, such as `select_position`. Therefore, another function was implemented within Khepri to enable the extraction of a fragment that represents the geometry selected within the document where it was created, the `capture_shape` function. When executed, this function also asks the user to select a geometry from the modeling tool and returns a fragment that specifies both the modeling tool in use and the ID number of that geometry, e.g., `captured_shape(autocad, 36729)`. Thus, this approach mimics the behavior of the *Set One* and *Set Multiple* operations in Grasshopper, having the same limitation regarding the mandatory simultaneous operation between the program and, in this case, the Rhino document from where the visual inputs were selected.

Besides being an alternative process to achieve the intended live connection, metaprogramming also provides a different approach to independence. By using the predefined function `internalize_`-

`shape`, it is generated, in the AD program, a fragment that represents the way the language would generate the same imported geometry, i.e., a specific function with specific parameters, e.g., $\texttt{sphere}(\texttt{center} = \texttt{xy}(3, 6), \texttt{radius} = 9)$. This alternative approach surpasses the limitations of depending on a specific CAD document, while also allowing the integration of the imported geometry, that not only becomes visually available in the program, but also becomes editable. This is more advantageous than Grasshopper's *Internalize Data* option that, on the contrary, fastens the geometry within the storing component in a way that it can no longer be changed within its implementation.

In table 1, we summarize the implementation of VIMs within Khepri, describing the four operations that establish different relations between the program, the CAD document, and the geometry imported.

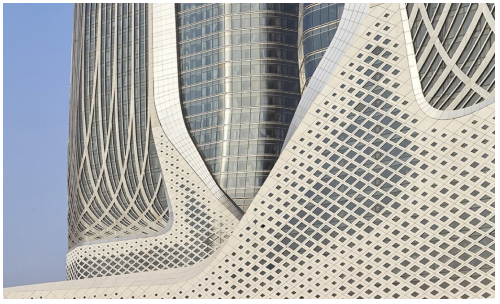

Figure 6
Nanjing International Youth Cultural Center by Zaha Hadid Architects, Nanjing, China, 2018 [2].

5. EVALUATION

In order to evaluate the implementation of VIMs within the textual programming context, we developed an architectural design challenge within the two programming approaches, i.e., using both a VPL, Grasshopper, and a TPL within Khepri. Both programs generate the same geometry and require the use of identical, previously modeled geometries as inputs.

As a case study, we programmed an attractor mechanism to simulate a similar effect to the façade of the Nanjing International Youth Cultural Center by Zaha Hadid Architects (Nanjing, China, 2018) (figure 6). The generated design inspired by this façade includes a grid of rectangular openings, whose dimension vary according to their distance to an attractor point and a sine function, in order to mimic the wave-like effect.

5.1. Grasshopper

The visual program created in Grasshopper imports four different types of visual inputs, namely, a surface for the façade, a curve for the façade thickness, a geometry, to describe the window shape, and a point to locate an attractor (figure 7).

Therefore, the program first analyzes the imported surface to compute a matrix of positions to then iterate the imported geometry that represents the openings. Hence, it is extracted a list of values containing the distances between each geometry and the attractor point that is used to scale each corresponding geometry, creating a pattern. This pattern is later extruded and subtracted from a wall, that was also generated from the extrusion of the surface along the value of the imported line.

Furthermore, in order to simplify and explore different natures of attractors, we created a second version of this program that, instead of generating a pattern with one single attractor point, explores Grasshopper's ability to select multiple inputs within a single storing component, where we selected multiple points to create an attractor curve.

The inherent dependence between the geometry imported and the program allows generating different versions of this façade by either changing the position of the attractor point or editing the points of the attractor curve (figure 8).

5.2. Khepri

Regarding Khepri, we were able to implement a similar program that benefits from other assets of textual programming. For instance, the program takes advantage of array comprehensions and higher-order functions such as the `map_division` function that applies a function to each element of a generated

	Operations	Functions	Dependence from the AD program	Original CAD document required
Without metaprogramming	Select	`select_position, select_point, select_curve, select_surface`	✗	✗
	Live Dependence	`with_shape_dependency`	✓	✗
With metaprogramming	Capture	`capture_shape`	✓	✓
	Internalize	`internalize_shape`	✗	✗

Table 1
Operations available for the implementation of VIMs within Khepri.

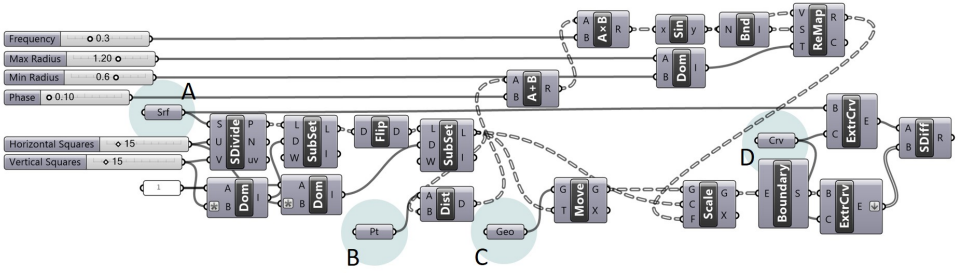

Figure 7
Grasshopper program importing four different types of visual inputs: (A) the surface of the façade, (B) the attractor point, (C) the shape of the windows, and (D) the line that gives the thickness of the façade.

matrix of values. In this case, it creates a matrix of positions that is then used to iterate the geometries of the windows.

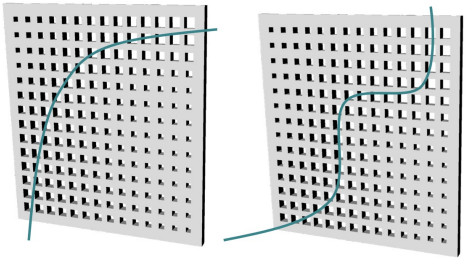

Figure 8
Alterations in the input points of another Grasshopper program that creates an attractor curve.

This also represents one of the greatest advantages of textual programming within Khepri: the ability to create flexible abstractions such as the `attractor_windows` function, that receives a function describing the shape of the window to be iterated along the area of the surface, as follows:

```
attractor_windows(f,attractor,length,
                  height,nx,ny) =
[f(p, attracted_radius(p,attractor))
 for row in map_division(
                  xy, 0, length, nx, 0,
                  height, ny)[2:end-1]
 for p in row[2:end-1]]
```

In this case, we chose a square, inspired by the Nanjing project, but we could have given as a parameter a circle, or a hexagon, for example (figure 9).

Moreover, the program itself can be flexible to integrate into a single script the possibility of selecting different types of visual inputs. While in Grasshopper we had to create two similar but independent programs to generate a façade with one attractor point or one attractor curve, the textual program can identify the type of the visual input by the selection performed by the user: he can select (1) one single attractor point, `p1`, as the first example, (2) multiple separated positions that work individually as attractor points, `pts`, or (3) multiple sequential positions that create an attractor curve, `c`, as the second exam-

ple in Grasshopper. Therefore, when the program is executed, it is able to identify the request of the user and choose which of those functions to perform, as follows:

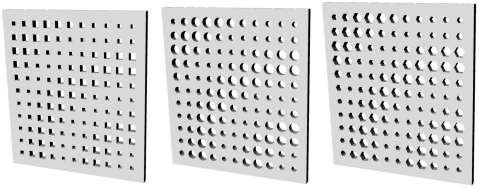

Figure 9
Different shapes for the openings of the façade.

```
attracted_radius(p, attractor) =
    min_radius+(max_radius-min_radius)
    *(sin(frequency*min_distance(
                      p, attractor)
          +phase)+1)/2

# minimum distance to a point
min_distance(p, p1::Loc) =
  distance(p,p1)

# minimum distance to a list of points
min_distance(p, pts::Locs) =
  minimum(map(p1->distance(p,p1),pts))

# minimum distance to a spline curve
min_distance(p, c::Spline) =
  min_distance(p, division(c, 20))
```

Another interesting advantage is the ability to stage operations: for a single attractor point, for instance, the user first selects successive positions until he finds one that pleases him. During this process, the interaction between the user, the tool, and the program being generated simulates Grasshopper's immediate feedback. Only after the architect chooses the final position of the attractor, the program is allowed to compute more demanding operations. This means that Khepri enables the creation of computational stages, which has a positive impact on performance, allowing to postpone more time-consuming operations to after other simpler decisions are made. This process is implemented as follows:

```
let p = nothing
  # 1st phase: loop until final choice
  while (new_p = select_position()) !=
        nothing
    p = new_p
    delete_shapes(output)
    with(current_layer, output) do

      # quick visualization
      create_windows(facade_surface,p)
    end
  end

  # 2nd phase: create the facade
  create_facade(facade_surface,
                thickness_curve, p)
end
```

In this case, the more demanding operations are the subtraction of the parallelepipeds from a wall. These functions are created to also be flexible and capable to receive as inputs the resulting data of the analysis of an imported surface and line, to obtain the size of the iteration and the thickness of the wall, as performed by the program in Grasshopper.

6. KHEPRI AND THE TEXTUAL APPROACH

Considering that our discussion is contextualized within a specific textual programming tool for architecture, Khepri, we extended the implementation of VIMs to integrate and illustrate some of its most relevant advantages.

Some of the main benefits of the textual approach regard to the flexibility of the abstractions and programs created. The sophisticated abstractions such as higher-order functions, the immediate interactivity with the modeling tool, and the interaction phases are some of the mechanisms enabled by textual programming.

Furthermore, and regarding Khepri in particular, one of its greatest advantages is related to portability, a feature that allows a single program to generate the same geometry in multiple back-ends, i.e., different modeling and analysis tools. Therefore, in addition to generating this façade in AutoCAD, the CAD

tool that was chosen for this evaluation, we also generated the same geometry in Revit, a BIM tool.

In Khepri, and unlike Grasshopper that requires an extension through complex plug-ins to connect with BIM tools, this portability is simple, as the user simply needs to inform Khepri which back-end to use.

Nevertheless, in order to generate geometry in different back-ends, the user should predict the type of information it will require, as the program will need to incorporate as much information as required by the most demanding tool it will connect to, e.g., BIM tools require more information from a program regarding the objects to use than CAD tools, that solely deal with pure geometry.

Therefore, in order to simulate a constructive solution for the generated façade, we imported a family of windows pre-modeled in Revit that had the particularity of having a parametric width and height. Thus, when incorporated into the program, this family scales to this extent, maintaining the same effect accomplished with the CAD tool (figure 10).

This connection with BIM tools may sometimes require compromises regarding the design outcomes when using the predefined families available. For instance, the majority of the existing window families did not allow parametrization due to the specific properties inherent to each family. If we had chosen one of those windows, they would not be able to scale, and the façade would lose the attractor effect. One possible solution is the manual generation of personalized families of objects within Revit, incorporating parametric variables in the dimensions of those objects.

This research within Khepri enabled the extension of VIMs to another paradigm, allowing a program that imports geometry from a CAD tool to also generate 3D models in a BIM tool. However, there are some considerations to acknowledge regarding the performance of the implementation of VIMs in a textual programming context.

A program developed in Grasshopper benefits from a deep integration within the CAD tool in which the language is implemented. This allows the visualization of geometry in the modeling tool that has not yet been computed, i.e., that was not generated as geometric data in Rhino. Thus, mechanisms such as immediate feedback become more performant, since heavy operations are not being executed. These operations, such as *Bake*, are only performed at the user's request, who can then choose when the program is ready to generate the final geometry in the modeling tool.

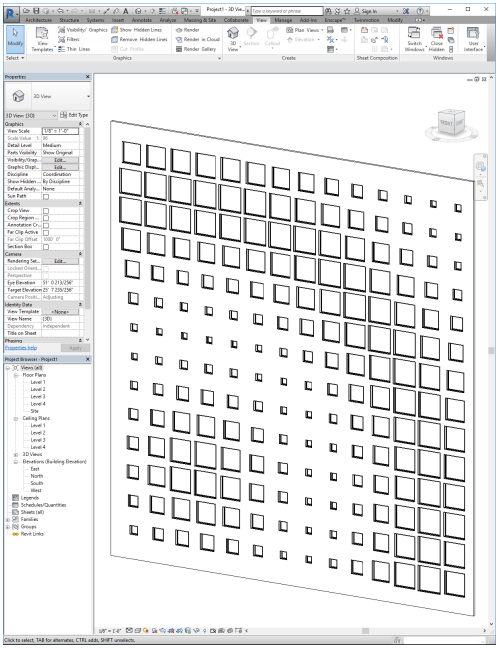

Figure 10
Portability of Khepri: generation of the same geometry in a BIM tool, Revit.

This process does not happen in Khepri, since, at each iteration, it is always generating geometry which, being a demanding computation, compromises the performance of the program. Even so, sequential interaction mechanisms allow to partially circumvent this limitation, enabling the user to decide when to establish a live connection with the tool or when to compute these more demanding operations.

7. CONCLUSIONS

In an architectural practice that is increasingly embedding new technologies, Algorithmic Design (AD) is an alternative to the conventional design and visualization methods that has been contributing with a number of advantages to the discipline. Within the AD practice, there is a noticeable tendency for the use of Visual Programming Languages (VPLs) over textual ones (TPLs) among architects. Unfortunately, despite providing intuitive features and mechanisms that make them more interactive and appealing for their users, VPLs struggle to scale with complex programs. On the other hand, TPLs are less encouraging to use due to their steeper learning curve and lack of interactivity. Nevertheless, their advantages for the development and maintenance of complex programs are strong arguments to work on their appeal to architecture by implementing visual features and mechanisms.

Visual Input Mechanisms (VIMs) in a textual programming context, besides allowing the use of geometry previously modeled as input, also proved to bring textual languages closer to the user by supporting a more dynamic interaction with the modeling tool.

In this paper, we assessed the current integration of VIMs within Grasshopper, a VPL, and within Khepri, a textual AD tool for architecture, by developing a hypothetical design problem in both paradigms. The comparison of both approaches promoted a more complete integration of VIMs in the textual approach, by either allowing the dependence or independence from the modeling tool in use. Those effects were accomplished through the use of metaprogramming, a textual programming mechanism that generates the program fragment that describes the indended input, either referencing it directly or by extracting its properties and producing an expression that generates an equivalent shape.

Nevertheless, some issues regarding performance should be considered by the user when choosing the type of interaction with the modeling tool. Considering that VIMs within Khepri require the continuous generation of geometry in the modeling tool, whereas Grasshopper enables a temporary visualization of what can be generated by the program, there should be a compromise between the interactivity and the time used to perform those mechanisms.

Furthermore, and even though the textual approach is less appealing and intuitive for less experienced users, it still brings further advantages regarding: (1) the support of a greater design complexity - without compromising the legibility and performance of the program created, (2) the improvement in the program's performance - by supporting sequential interaction phases, that allows to only perform computationally demanding operations when precedent decisions are made, (3) the increased range of combinations of the available operations, freeing the user from pre-defined abstractions, and (4), regarding VIMs in particular, the greater flexibility in manipulating and integrating the data of the imported geometry. Furthermore, in the particular case of using Khepri, these advantages are further extended by also (5) allowing the integration within other modeling tools, such as Revit, a Building Information Modeling (BIM) tool.

ACKNOWLEDGMENTS

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references UID/CEC/50021/2019 and Project Khepri, PTDC/ART-DAQ/31061/2017.

REFERENCES

Alfaiate, P, Caetano, I and Leitão, A 2017 'Luna Moth: Supporting Creativity in the Cloud', *ACADIA 2017: DISCIPLINES & DISRUPTION, Proceedings of the 37th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA)*, MIT, Massachusetts, USA, p. 72–81

Bezanson, J, Edelman, A, Karpinski, S and Shah, VB 2017, 'Julia: A Fresh Approach to Numerical Computing', *SIAM Review*, 59, p. 65–98

Brown, TB and Kimura, TD 1994, 'Completeness of a Visual Computation Model', *Software – Concepts and*

Tools, no. 15, p. 34–48
Clarisse, O and Chang, SK 1986, 'Vicon: A Visual Icon Manager', in Chang, SK, Ichikawa, T and Ligomenides, PA (eds) 1986, *Visual Languages. Management and Information Systems*, Springer„ Boston, MA, USA, pp. 151-190
Davis, D, Burry, J and Burry, M 2011, 'Understanding visual scripts: Improving collaboration through modular programming', *International Journal of Architectural Computing*, 9(4), p. 361–376
Janssen, P 2014 'Visual Dataflow Modelling: Some Thoughts on Complexity', *Fusion - Proceedings of the 32nd eCAADe Conference - Volume 2*, Department of Architecture and Built Environment, Faculty of Engineering and Environment, Newcastle upon Tyne, p. 305–314
Leitão, A, Lopes, J and Santos, L 2014 'Illustrated Programming', *ACADIA 2014: Design Agency, Proceedings of the 34th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA)*, Los Angeles, USA, p. 291–300
Leitão, A and Santos, L 2011 'Programming Languages For Generative Design: Visual or Textual?', *Respecting Fragile Places: 29th eCAADe Conference Proceedings*, University of Ljubljana, Slovenia, pp. 139-162
Leitão, A, Santos, L and Lopes, J 2012, 'Programming Languages For Generative Design: A Comparative Study', *International Journal of Architectural Computing*, 10(1), pp. 139-162
Lopes, J and Leitão, A 2011 'Portable generative design for CAD applications', *Integration Through Computation - Proceedings of the 31st Annual Conference of the Association for Computer Aided Design in Architecture, ACADIA 2011*, Alberta, Canada, p. 196–203
Menzies, T 2002, 'Evaluation Issues for Visual Programming Languages', in Chang, S (eds) 2002, *Handbook of Software Engineering and Knowledge Engineering, vol. 2: Emerging Technologies*, World Scientific Publishing Co. Pte. Ltd, London, pp. 93-101
Myers, BA 1990, 'Taxonomies of visual programming and program visualization', *Journal of Visual Languages & Computing*, 1(1), pp. 97-123
Noone, M and Mooney, A 2018, 'Visual and Textual Programming Languages: A Systematic Review of the Literature', *Journal of Computers in Education*, 5(2), pp. 149-174
Reas, C and Fry, B 2007, *Processing: a programming handbook for visual designers and artists*, The MIT Press, Cambridge, Massachusetts & London, England
Schaefer, R 2011, 'On the limits of visual programming languages', *SIGSOFT Software Engineering Notes*, 36(2), pp. 7-8
Zboinska, MA 2015, 'Hybrid CAD/E platform supporting exploratory architectural design', *CAD Computer-Aided Design journal*, 59, pp. 64-84
[1] https://vimeo.com/203509846
[2] https://www.zaha-hadid.com/architecture/nanjing-culture-conference-centre/
[3] http://www.theprovingground.org/2013/07/autodesk-edu-videos-computational.html