FROM VISUAL INPUT TO VISUAL OUTPUT IN TEXTUAL PROGRAMMING

MARIA SAMMER[1], ANTÓNIO LEITÃO[2] and INÊS CAETANO[3]
[1,2,3]*INESC-ID/IST, University of Lisbon*
[1,2,3]*{maria.joao.sammer|antonio.menezes.leitao|ines.caetano}@tecnico.ulisboa.pt*

Abstract. Algorithmic Design is an approach that uses algorithms to generate designs. These algorithms are built using either a Visual Programming Language (VPL) or a Textual Programming Language (TPL). In architecture, there is a clear propensity to the use of VPLs, e.g., Grasshopper or Dynamo, over the use of TPLs, e.g., Python or AutoLisp. In addition to all the user-friendly and interactive features that make VPLs more appealing to architects, most of them already integrate components for textual programming. In contrast, TPLs have not been as successful in incorporating visual features. Given the user-friendliness of VPLs and the relevance of TPLs for large-scale and complex designs, we discuss Visual Input Mechanisms (VIMs) in the context of TPLs. In this paper, we extend previous research in this area by exploring and implementing the most valuable VIMs in a TPL adapted for architectural design.

Keywords. Algorithmic Design; Metaprogramming; Textual Programming Languages; Visual Input Mechanisms.

1. Introduction

Algorithmic Design (AD) is no longer a novelty in architecture. While this paradigm shift is settling in the common practice of several architecture studios worldwide, more practical questions regarding its implementation arise. Multiple studies (Leitão et al. 2012, Janssen 2014, Zboinska 2015, Leitão and Santos 2011, Davis et al. 2011) on the use of textual and visual programming languages in AD have shown that each one has its own advantages and disadvantages. Visual Programming Languages (VPLs) are more intuitive to non-programmers due to user-friendly features that better adapt to the visual nature of architects. On the other hand, Textual Programming Languages (TPLs) are less intuitive but offer a clear advantage when dealing with more complex programs. Still, there is a clear preference among architects to use VPLs. Not only do they provide visual features that make them more appealing and easier to use, but they also support textual programming. In contrast, TPLs have a steeper learning curve and do not integrate the visual mechanisms that bring them closer to their users. In this paper, we address this issue by proposing Visual Input Mechanisms (VIMs) for TPLs and demonstrating their usefulness in the context of architectural design.

Intelligent & Informed, Proceedings of the 24th International Conference of the Association for Computer-Aided Architectural Design Research in Asia (CAADRIA) 2019, Volume 1, 645-654.

2. VPLs vs TPLs

One of the biggest disparities between VPLs and TPLs in AD is related to user interaction with both the AD program and the associated modeling tool. In architecture, VPLs are currently more popular than TPLs because of their friendlier programming environment and the available interactive features. The use of VPLs is based on drag-and-drop of (1) graphical components, representing geometry, functions, and other mathematical entities, and (2) the connections between these components. Using predefined components, architects can focus on devising the logic of the design exploration, instead of the intricacies of textual programming abstractions. The resulting process is, therefore, more intuitive and closer to the manipulation of objects in the real world (Clarisse and Chang 1986). Moreover, VPLs also provide visual interaction mechanisms to more intuitively change the parameters' values, e.g., *Grasshopper*'s `Number Sliders` and `Boolean Toggles`. Finally, some VPLs also integrate useful features for the development and debugging of AD programs: (1) *traceability*, to highlight the geometry generated by a selected component in the modeling tool; and (2) *immediate feedback*, to visualize in real time the effects of the changes being made to the program.

Unfortunately, VPLs tend to suffer from scalability problems. In fact, for complex programs, not only some of the components have performance problems, but also the legibility of the AD program tends to deteriorate due to the excess of components and tangled connections. In this regard, TPLs can improve upon VPLs as (1) compilation allows an efficient use of computational resources and (2) abstraction mechanisms aid the program understanding process.

Nevertheless, most TPLs neither provide traceability nor immediate feedback, instead requiring a cycle where the user (1) writes the program, (2) runs it, and (3) visualizes the result. However, there are exceptions, such as *Processing* (Reas and Fry 2007), that only supports immediate feedback, and *Luna Moth* (Alfaiate et al. 2017), a web-based AD Integrated Development Environment (IDE) for TPLs, that provides (1) user-interaction mechanisms similar to *Grasshopper*'s sliders, (2) immediate feedback, and (3) traceability mechanisms that improve upon *Grasshopper*'s by being bi-directional. A more extensive but less performant approach was presented by Leitão et al. (2014), where the traceability mechanism also illustrates the control flow of the program.

Furthermore, VPLs also support visual inputs, namely, the ability to use geometric shapes, such as points, curves, and surfaces, as input of the program. Unfortunately, this useful feature is absent in most TPLs. In this research, we focus on the implementation of such mechanisms in the context of TPLs.

3. Visual input mechanisms

A VIM allows the use of manually created geometric shapes as inputs to the AD program. VIMs are particularly helpful when the intended geometry is easier to produce by hand or when it is necessary to use already existing geometry, e.g., a city plan. It is also possible to use, as input to an AD program, geometry that was itself produced as output of another AD program, as it happened with the

development of the Morpheus Hotel (Heijden 2017).

In *Grasshopper*, there are specific components to import different kinds of geometries, e.g., the `Point` component only stores points, which prevents input errors and makes the program more understandable. These components then give access to the information about the imported shapes, such as coordinates and center points, length of lines and vectors, area of surfaces, or volume of solids.

While a program is running concurrently with the document that contains the imported shapes, these can be changed by the user, who expects the program to immediately assimilate the changes and recompute the results. However, this might be a disadvantage, as the program becomes dependent on another Computer-Aided Design (CAD) document, a drawback that can be overcome by the `Internalize Data` option of *Grasshopper*. This mechanism processes the geometry that was imported into the component and, assuming it is fixed, it preserves its information directly in the AD program, thus freeing the latter from depending on the document that contains the imported geometry.

4. Visual input mechanisms in a textual programming environment

To explore the implementation of VIMs within the textual programming context, we use a novel textual AD tool, *Khepri*, that is based on the TPL *Julia* (Bezanson et al. 2017). *Khepri* is a descendant of *Rosetta* (Lopes and Leitão 2011), supporting multiple CAD and Building Information Modeling (BIM) backends, while exploring a different software architecture.

4.1. IMPLEMENTATION

Khepri provides a set of pre-defined operations to facilitate the portable generation and manipulation of geometry in a series of modeling tools, including *Rhinoceros*, *AutoCAD*, and *Revit*, among others. More interestingly, *Khepri* provides operations to treat shapes created in the modeling tool as an input to the designer's textual program, such as `select_position`, `select_point`, `select_curve`, and `select_surface`. When one of these operations is executed, *Khepri* asks the user to select an entity of the corresponding kind in the modeling tool, which will be provided as input to the AD program. This facilitates the creation of programs that do incremental and/or repeated selection of input shapes, while providing immediate feedback on the user choices. Differently from VPLs, such as *Grasshopper* or *Dynamo*, this approach keeps the program independent from the CAD documents containing the selected geometry. On the other hand, it does not establish a live connection between that geometry and the AD program, meaning that changes in the selected shapes are not taken into account unless they are re-selected. When that live connection is intended, the AD program only needs to inform *Khepri* that there is a dependency on a set of shapes, so that when the input is changed in the modeling tool, the AD program is automatically re-executed, reacting to those changes. An advantage of this approach is the ability to separate two different features: having a dependency on a visual input and making that dependency a live one.

Finally, for the cases where we want to establish a dependency that is preserved

across different design sessions, *Khepri* relies on metaprogramming.

4.2. METAPROGRAMMING

Metaprogramming entails the use of programs that generate other programs. In this research, metaprogramming is used to generate, in the AD program, fragments that represent existing visual inputs. To that end, *Khepri* analyzes the visual input and generates a textual program fragment that, when executed, reproduces that same input. Thereafter, this fragment can be inserted in the original AD program, replacing the fragment that asked the user to select the input. Additionally, this can be done by the programming environment, relieving the designer from the required textual changes.

There are two different use cases for this feature. The first one emulates *Grasshopper*'s `Set One...` and `Set Multiple...` operations, where *Khepri* generates expressions that designate shapes in some Computer-Aided Design (CAD) document, making those shapes a permanent part of the AD program. To that end, the operation `capture_shape` can be used, possibly in combination with just one selection operation. The result is an expression that univocally identifies the selected shape. As *Khepri* supports multiple CAD tools, this expression specifies, not only the identification of the captured shape, but also the tool being used. For example, when using *AutoCAD*, the function generates an expression similar to $\texttt{captured\_shape}(\texttt{autocad}, 49675)$. Given that the identification of the shapes is preserved between different sessions of *AutoCAD*, this ensures that the program references the exact same shape as long as it is executed in the context of the same *AutoCAD* document. Similar identifications are used for other CAD tools.

Finally, when the goal is to isolate the AD program from the document containing the shapes upon which it depends, we can internalize the shapes by generating expressions that reconstruct them. As an example, consider a circle that was previously imported from a document, which the designer wants to internalize in his AD program. By using *Khepri*'s `internalize_shape` operation, presumably after the `select_curve` or `capture_shape` operations, the selected circle is inspected and then used to generate a program fragment, e.g., $\texttt{circle}(\texttt{center} = \texttt{xyz}(1, 2, 3), \texttt{radius} = 4)$. This functionality enables internalization of imported geometry much alike the `Internalize Data` feature of *Grasshopper*. Therefore, not only can we use the geometry in the modeling tool directly in the AD program, but we can also incorporate its corresponding algorithmic definition in other parts of the AD program, allowing the latter to become independent of the CAD model.

Table 1 illustrates the different dimensions of visual inputs for TPLs and the corresponding *Khepri* operations. Given that the dimensions are completely orthogonal, the different operations can be used in arbitrary combinations, to suit whatever user interaction is desired.

Table 1. Visual input operations in Khepri: there are single and multiple input versions of these functions.

Operation	Function in Khepri	Dependency on the AD program
Select	`select_position, select_point, select_curve, select_surface`	✗
Live Dependence	`with_shape_dependency`	✓
Capture	`capture_shape`	✓
Internalize	`internalize_shape`	✗

5. Evaluation

To evaluate the use of VIMs within TPLs, we developed a case study using both *Grasshopper* and *Khepri*, while considering the same set of parameters and visual inputs. We exemplify two different moments of the AD process: (1) the use of VIMs within the program's definition and (2) the internalization of visual input data.

The case study is inspired by the Quality Hotel Friends, whose façade design results from the use of *attractors* to create a wave-like effect with windows of different sizes. In this experiment, we consider three visual inputs previously created in the CAD tool: (1) the surface of the façade, (2) a curve representing the façade's thickness, and (3) an attractor point.

5.1. VPL: GRASSHOPPER

Using *Grasshopper*, the `Surface` component is used as input to other components to (1) compute a uniform grid of locations on the surface, (2) filter the outer locations, (3) measure the distances between the locations and the attractor point, (4) map the sinusoidal function over those distances, and (5) remap the results to determine the windows' radii. During this computation, `number sliders` can be used to define the number of circles, the sinusoid's frequency and amplitude, and the windows' minimum and maximum radii. Then, the circles are (6) extruded along the input curve to create a set of cylinders, which (7) are subtracted from a parallelepiped, also generated from (8) the extrusion of the imported surface along the same curve.

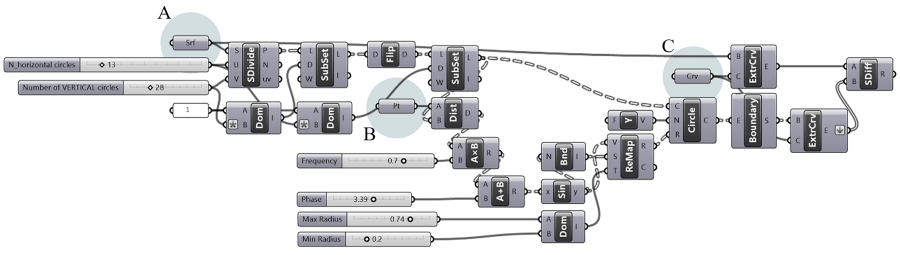

Figure 1. Grasshopper program using a visual input mechanism. The imported geometries are: (A) a surface, (B) a point, and (C) a curve.

Figure 1 shows the *Grasshopper* program used, highlighting the three visual inputs required. Figure 2 illustrates both the original design and several variations.

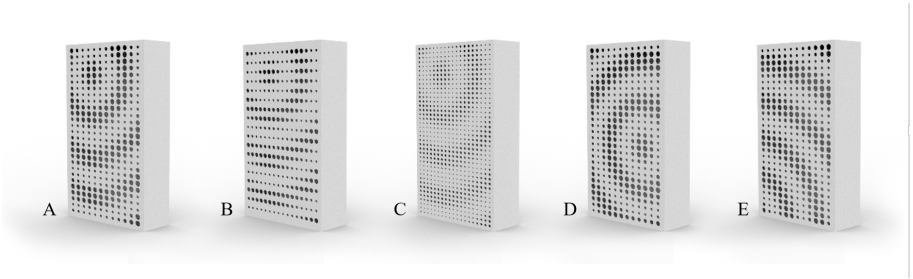

Figure 2. The original geometry generated from the program (A) and some variations: varying the number of circles (B, C) and moving the point attractor (D, E).

To illustrate *Grasshopper*'s ability to use multiple geometries as input, the AD program was adapted to receive six different points to define an attractor curve. Figure 3 illustrates the application of this version of the program.

Despite the usefulness of *Grasshopper*'s traceability and immediate feedback, the example also shows that the scalability of the program is already compromised at the early stages of its development: any change in the parameters, such as the attractor points, the number of windows, or their radii, blocks *Grasshopper* and *Rhino* until the computation is finished, making the interaction tiresome.

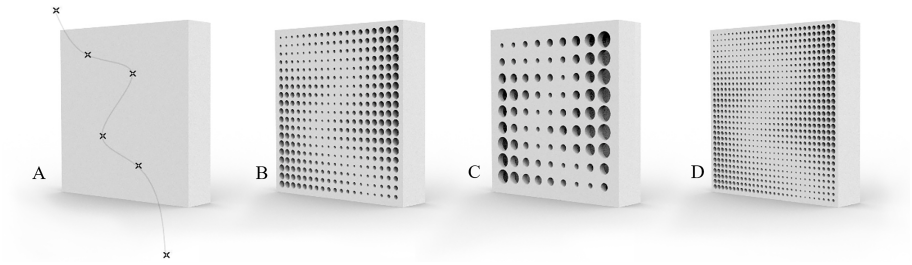

Figure 3. The new inputs of the adapted program: a list of points generating an attractor curve (A), the original generated geometry (B), and geometrical variations (C and D).

5.2. TPL: KHEPRI

Like *Grasshopper*, *Khepri* provides functionalities to (1) process shapes, including points, curves, and surfaces, and (2) create new geometric shapes, such as circles and extrusions. In this section, we reproduce the same case study to explain the options available in *Khepri* to process visual inputs. For this, we assume again that a surface was manually created in the CAD tool.

To generate a grid of positions on the surface, we use the higher-order operation `map_division` that takes as arguments a function, a surface, and the number of

values in the *u* and *v* dimensions and generates a matrix of positions similarly to the *Grasshopper*'s version. Then, we identify the location of the attractor and use the distance of each surface position to the attractor to influence the radius of each window.

One of the advantages of using a TPL is the finer control it provides regarding the computational process. In the case of a VPL based on the *dataflow* paradigm, such as *Grasshopper*, it is harder to exercise that control because the output of a component immediately affects the components to which it is connected. On the other hand, with a TPL, it is possible to create computational stages where the computation only moves to the next stage after some condition is achieved. In the case of the *Khepri* implementation, this allows us to postpone the computationally demanding operations that severely penalized *Grasshopper*'s performance, and freely experiment changing the attractor, while immediately seeing the future location and shape of the windows. When we commit to a final design, the AD program moves to the next stage, which now creates the actual windows (Figure 4). Given that this stage does not require immediate feedback, it can take as long as necessary. In the case of *Khepri*, this allows us to take advantage of its BIM capabilities so that the same AD program can generate an actual BIM model in, e.g., *Revit*. This is illustrated in Figure 5.

The following program fragment illustrates this approach:

```
output_layer = create_layer("Output")

let p = nothing
  while (new_p = select_position()) != nothing
    p = new_p
    delete_shapes(output_layer)
    with(current_layer, output_layer) do
      create_circles(facade_surface, p)
    end
  end
  create_facade(facade_surface, thickness_curve, p)
end
```

The program uses two variables named `thickness_curve` and `facade_surface` that reference the visual input being used. The previous program also illustrates a technique to separate the visual input from the visual output based on the use of layers: the generated shapes are placed on a dedicated layer named `Output` whose contents are erased before each iteration of the attractor placement. Also visible is the separation between the phase that repeatedly asks the user for the attractor placement (the `while` loop) while giving immediate feedback of the attractor effects, and the phase where the final placement is used to create the wall and subtract the windows.

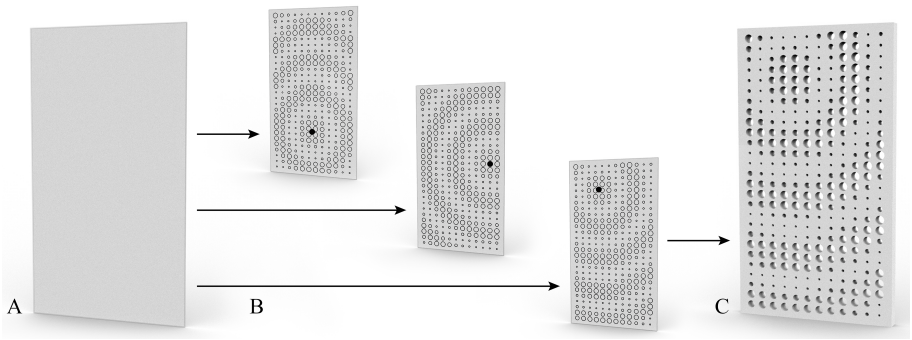

Figure 4. Computational design stages of the facade: (A) selection of the surface; (B) selection of different attractor positions and immediate visualization of the results; (C) final design choice and the generation of a 3D model.

In case we want to faithfully emulate *Grasshopper*'s behavior, instead of asking the user for the attractor location, we can ask him to select the point that represents the attractor. We can then create a dependency between this point and the creation of the façade, as illustrated in the next program fragment:

```
let p = select_point()
  with_shape_dependency(p) do
    delete_shapes(output_layer)
    with(current_layer, output_layer) do
      create_facade(facade_surface, thickness_curve, p.position)
    end
  end
end
```

In this case, each change in the location of the point will trigger the execution of the program, including the computationally demanding creation of the windows, evidencing the same performance problems that were visible in *Grasshopper*.

Finally, to use an attractor curve similar to the one generated in the *Grasshopper* program described previously (Figure 3), we can either use the `select_positions` operation, to ask the user to locate the different points that define the curve, or we can combine both `select_points` and `with_shape_dependency` operations to create a dependency on all selected points. In this last case, a change on any of these points will regenerate the façade.

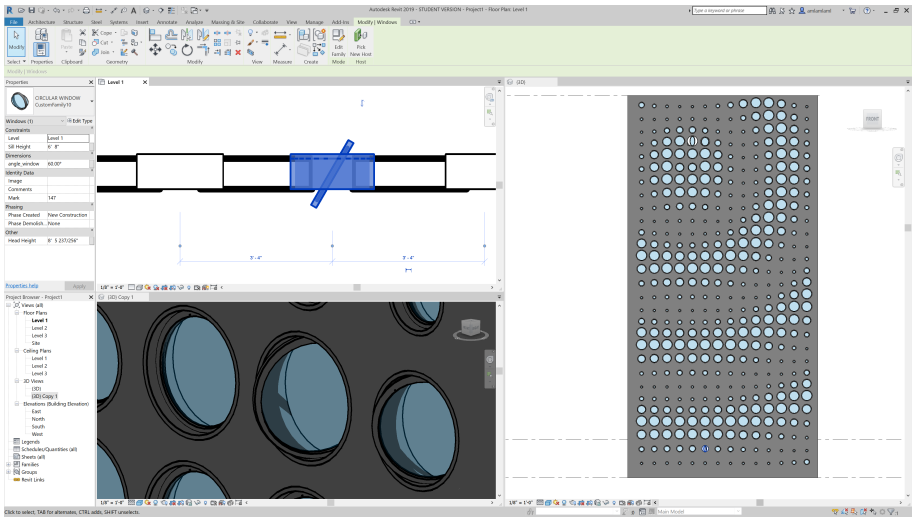

Figure 5. BIM model generated by Khepri in Revit after the user decided the final location of the attractor point.

6. Conclusions

Visual Programming Languages (VPLs) present several features that make them more appealing to non-experts, such as traceability, visual feedback, and Visual Input Mechanisms (VIMs). However, they suffer from a scalability problem, as some of these features stop working for more complex programs. On the other hand, Textual Programming Languages (TPLs) offer a better alternative to create and manage complex Algorithmic Design (AD) programs, even though they do not offer such appealing features. Nevertheless, their potential is a strong argument to motivate the implementation of visual features within the textual programming context.

This research demonstrates an implementation of VIMs within a textual programming environment based on an orthogonal organization of the relevant features. Besides implementing mechanisms similar to the ones offered by VPLs, it extends their advantages by (1) supporting a greater design complexity, (2) providing more flexibility in manipulating and integrating the data of the imported geometry, (3) increasing the range of combinations of the available operations, and (4) considerably improving the program's performance when dealing with computationally demanding operations by supporting sequential interaction phases.

In this paper, we evaluated the application of VIMs in the context of VPLs and TPLs by developing a case study using both scenarios. By comparing the approaches, we demonstrated that VIMs for TPLs allow the integration of visual inputs in two dimensions: in correlation with the modeling tool by selecting or capturing the geometry in the tool, or separated from the modeling tool by using metaprogramming to internalize the imported geometric data. We also

demonstrated the ability of the chosen textual programming environment - *Khepri* - to deliver the output of an AD program as a BIM model.

From the perspective of programming for architecture, we plan to extend the presented research to also support the use of visual inputs from BIM tools, such as *ArchiCAD* or *Revit*, making the AD programs capable of using the complex data that is usually associated with BIM objects. These should also include the ability to internalize this complex data.

Acknowledgements

This work was supported by national funds through *Fundação para a Ciência e a Tecnologia* (FCT) with references UID/CEC/50021/2019 and PTDC/ART-DAQ/31061/2017, and by the PhD grant under contract of FCT with reference SFRH/BD/128628/2017.

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)*, 72–81.

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

Clarisse, O. and Chang, S.K. 1986, Vicon: A Visual Icon Manager, *in* S.K. Chang, T. Ichikawa and P.A. Ligomenides (eds.), *Visual Languages. Management and Information Systems*, Springer, Boston, MA, 151-190.

Davis, D., Burry, J. and Burry, M.: 2011, Understanding visual scripts: Improving collaboration through modular programming, *International Journal of Architectural Computing*, **09**(04), 361–376.

Heijden, R.V.D.: 2017, "The Morpheus Hotel: From Design to Production: Live Webinar" . Available from <https://vimeo.com/203509846> (accessed Nov. 13, 2018).

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, 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, 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, Faculty of Architecture (Slovenia), 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), 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*, 196–203.

Reas, C. and Fry, B.: 2007, *Processing: a programming handbook for visual designers and artists*, The MIT Press, Cambridge, Massachusetts & London, England.

Zboinska, M.A.: 2015, Hybrid CAD/E platform supporting exploratory architectural design, *CAD Computer-Aided Design journal*, **59**, 64-84.