Programming Languages for Generative design

Visual or Textual?

António Leitão[1], Luís Santos[2]
[1]Instituto Superior Técnico/INESC-ID, Portugal, [2]IHSIS - Institute for Humane Studies and Intelligent Sciences, Portugal
[1]antonio.menezes.leitao@ist.utl.pt, [2]luis.sds82@gmail.com

Abstract. *In this paper we compare visual and textual programming languages for generative design. We argue that, in the past, this comparison has been flawed and that it is now time to reconsider the potential of the textual programming paradigm but using modern programming languages and development environments specifically targeted to the generative design domain. We present VisualScheme as a prime example of such language and we compare it with the most used visual programming language in the generative design field.*
Keywords. *Generative design; Visual Programming Languages; Textual Programming Languages; Interactive Development Environments.*

INTRODUCTION

Nowadays, digital tools to architectural design are increasingly used as "generative tools for the derivation of form and its transformation" (Kolarevic, 2000) and it is common to hear about parametric, algorithmic, and generative design. Through parametric design one can manipulate a design or its parts with variable parameters while the algorithmic approach allows the designer to describe an architectural shape by the application of rules and constraints or a coherent combination of procedures. The generation of shapes through the application of any of these techniques is described as generative design.

It is true that some of those concepts are not new and were applied in architecture even before the invention of digital computers, as is visible, for example, in the architecture of Antoni Gaudí (Katz, 2010) but the use of computers popularized them by facilitating its description and application.

In order to implement a concept in a computer one must first translate the thought process into a formal language that the computer understands: a Programming Language (PL). The increasing use of programming in the design field had an impact on architecture praxis and theory in such a way that some announced the emergence of a new architectural style, the Parametricism, as representative of avant-garde tendencies with the digital generative approach as a common ground [1].

This paper discusses the use of PL in generative design, distinguishing between Textual PLs (TPLs) and Visual PLs (VPLs). TPLs and VPLs are defined, compared and analyzed in terms of their advantages, disadvantages, development environments and fitness to the specific domain of generative design.

Because of its increasing popularity, we will consider Grasshopper (version 0.8.0010) as the VPL of reference within the generative design domain. On the other hand, due to its wide availability in common CAD tools, we will focus on VBScript and its descendants as representative TPLs for generative design.

We will argue that comparing state-of-the-art VPLs with old general purpose TPLs is not appropriate. Instead, the comparison should be made with modern TPLs. We will use VisualScheme (Leitão et al, 2010) as a case study to show that TPLs can be more productive than VPLs, particularly, when the complexity of the design problem increases.

PROGRAMMING LANGUAGES: VISUAL AND TEXTUAL

A programming language is a formal medium for expressing ideas and not just a way to get a computer to perform operations. This means that programming languages should match the human thinking process, which includes the ability to combine simple ideas to form compound ones and the ability to abstract complex ideas so that they become more general (Locke, 1690).

Every programming language that follows these principles is made of three important concepts: (1) primitive elements, (2) combination mechanisms, and (3) abstraction mechanisms. General-purpose languages provide few pre-defined abstractions, while domain-specific languages provide abstractions tailored to a given domain.

A visual programming language (VPL) allows the description of programs in a bidimensional representation consisting of iconic elements that can be interactively manipulated by the user according to some spacial grammar (Myers 1990). Figure 1 shows a small visual program (designed in Grasshopper) that computes a sequence of points belonging to a conic spiral.

In a textual programming language (TPL), programs are described using a linear sequence of characters. The major difference between a VPL and a TPL is, thus, on the number of dimensions: TPLs are one-dimensional while VPLs are, at least, bi-dimensional.

For comparison, the following textual fragment (written in RhinoScript) computes the same sequence of points of a conic spiral:

```
Function ConicSpiral(Length,N)
  Dim points()
  ReDim points(N-1)
  Dim t, i
  For i=0 To N-1
    t=i*Length/N
    points(i)=Pt(t*Cos(5*t),
                 t*Sin(5*t),
                 t)
  Next
  ConicSpiral=points
End Function
```

In spite of the large number of studies comparing VPLs with TPLs, there is no conclusive evidence regarding their relative advantages (Menzies, 2002). It is generally admitted, though, that VPLs are more motivating for beginners, allowing them to become productive sooner. On the other hand, TPLs are considerably more productive for dealing with complex problems and, in fact, most of the existent programming languages are TPLs.

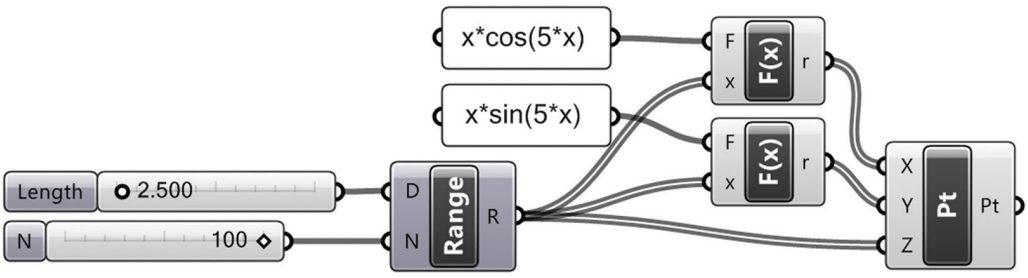

Figure 1
*A Grasshopper program for computing points of a conic spiral: values ranging from 0 to some length and their mappings using f(x)=x*cos(5*x) and g(x)=x*sin(5*x) become the point coordinates.*

One of the drawbacks of a traditional TPL is that it requires the user to master a relatively large set of concepts that, in many cases, are related not to the problem the user wants to solve but, instead, to implementation details of the language. For example, to understand just the first three lines of the previous RhinoScript example, the reader has to know (1) the syntax of functions, (2) the concept of array, (3) that arrays require declaration, (4) that non-statically sized arrays require a redimension operation, and (5) that arrays indexes start from zero. Additional knowledge is necessary to understand the full fragment. This shows the first significant advantage of a modern VPL over an old TPL: the amount of background knowledge required is smaller.

The second advantage is that, as a result of their iconic nature, the development environment of a VPL can present the user with all the language elements that can be used. TPLs, when used without a good development environment, usually require the programmer to either remember the functionality or to read extensive documentation.

The third one is the instant feedback provided by some VPLs: the effect of each element can be immediately visualized. This facilitates the detection of mistakes and the adjustment of the input parameters. It also permits an incremental development process, where each element added to the program can be immediately tested. In spite of a few exceptions, such as Fluxus (Griffiths, 2007), most TPLs do not support this type of development. Instead, the user is forced to go through a write-compile-execute iterative cycle that does not promote incremental development.

Unfortunately, VPLs also have some drawbacks: they do not scale well with the complexity of the design task and, in many cases, the majority of users rely on extensive copy and paste, creating future maintenance problems. Moreover, as programs grow larger, it becomes increasingly difficult to understand what they do. This might explain the size and throwaway nature of the majority of visual programs when compared with the multi-million lines of code and longevity of the large textual programs.

In spite of its disadvantages, there is a general perception among generative designers that VPLs are more productive than the textual alternatives. This perception is caused by two factors: the textual alternatives lack domain-specific concepts and they make it difficult to define them. Moreover, we will show that modern TPLs provide additional advantages over VPLs and, thus, should be considered as better alternatives for solving large scale problems.

MODERN PROGRAMMING LANGUAGES

A modern PL is designed to be unobtrusive to the programmer. To this end, it usually provides syntactic features that drastically simplify the development of programs.

To illustrate this point, we will consider one important abstraction applicable to our previous example: list comprehensions, a specialized syntax heavily influenced by the set-builder notation used in mathematics. For example, in the Haskell language (Hudak et all, 2007), the conic spiral becomes:

```
conicSpiral length n =
  [(t*(cos 5*t), t*(sin 5*t), t) |
    t <- [i*length/n | i <- [0..n]]]
```

The comparison between Haskell and RhinoScript shows the amount of knowledge that is required in each case and provides anecdotal evidence that modern TPLs can be significantly easier to use. Haskell is a prime example of a modern language but many other recent languages (e.g., Python) could be used with identical results.

In this paper, we will argue that it is possible and, in fact, advantageous, to use TPLs in place of VPLs as long as we restrict ourselves to modern implementations targeting the generative design domain.

VISUAL SCHEME

VisualScheme (Leitão et al, 2010) is a research project based on PLT Scheme (Findler et al, 2002) that combines a modern TPL with a pedagogical interactive development environment (IDE) focused on

generative design programs. VisualScheme aims at exploring the advantages of TPLs, in particular, to handle the complexity of large programs, with some of the advantages of VPLs, including domain-specific constructs and IDE that provides auto completion, immediate feedback, visual widgets for input, and visualization of the results in the most used CAD applications.

As a first example, the following definition computes the set of points of the conic spiral that was computed in our previous examples:

```
(define (conic-spiral length n φ)
  (for/list ([i (in-range 0 n)])
    (let ([t (/ (* i length) n)]))
      (cyl t (* φ t) t))))
```

There are two noteworthy differences between this example and the Haskell example. The first one is the use of the fully parenthesized prefix notation that is typical of Lisp dialects. This notation might seem strange at the beginning but our teaching experience shows that it can be quickly mastered. The second one is the use of coordinate systems: in order to better match the generative design domain, VisualScheme implements the traditional Cartesian, polar, cylindrical, and spherical coordinate systems. As can be seen in this example, where the cylindrical coordinate system is being used, an adequate choice of coordinate system reduces the need for trigonometric expressions.

The definition we have just presented replicates the corresponding Grasshopper example that we provided in the beginning of the paper but it is not a typical VisualScheme definition. The typical implementation is, usually, more parametric, as can be seen in the following example:

```
(define (conic-spiral r φ z dr dφ dz n)
  (if (= n 0)
    (list)
    (cons (cyl r φ z)
          (conic-spiral (+ z dz)
                        (+ r dr)
                        (+ φ dφ)
                        dr dφ dz
                        (- n 1)))))
```

This second example illustrates a recursive definition, where the function is defined in terms of itself. In this case, the conic spiral is parameterized by the starting radius, rotation, and height; by their successive increments, and, finally, by the number of incremental steps. Note that this example cannot be directly encoded in Grasshopper (without using special scripting components) because it currently lacks the ability to do recursion. In this particular case, this is not as bad as it looks because it is possible to find different encodings that produce the same results. In the general case, however, this can be a serious limitation.

If, instead of the linear behaviour allowed by the recursive definition, one prefers different kinds of spirals, where, for example, the radius changes non linearly with the height of the spiral, then it might be preferable to use an approach where that variation is explicitly described. One hypothesis is to define each different variation in the range [a,b] as a function over the domain [0,1][1]:

```
(define (linear a b)
  (λ (t)
    (+ a (* t (- b a)))))
```

Note that each call to the linear function computes another function as a result, making it one example of a higher-order function. This capability is

1 For reasons of space, the following function definitions will use short names for parameters. It should be noted that normal VisualScheme programs use longer and clearer names

rare in older PLs but common in modern ones.

As an example of a non-linear variation, consider the following function that implements a sinusoidal variation in the interval [a,b], with amplitude d and frequency f:

```
(define (sinusoidal a b d f)
  (λ (t)
    (+ a
       (* t (- b a))
       (* d (sin (* f 2 pi t))))))
```

Using this approach, many different variations can be easily implemented. In order to to compute the actual variation, we need to generate the function domain as a sequence of values in the interval [0,1] . The image of the function over that domain is obtained by mapping it over the sequence of values:

```
(define (variation f n)
  (map f (range 1 n)))
```

The conic spiral is then just the mapping of cylindrical coordinates over three linear variations of the radius, angle, and height. A more interesting example is the spiral whose radius shows a sinusoidal variation along the height:

```
(define (spiral r0 r1 φ0 φ1 h d f n)
  (map cil
       (variation
         (sinusoidal r0 r1 d f) n)
       (variation
         (linear φ0 φ1) n)
       (variation
         (linear 0 h) n)))
```

Much more complex geometries can be defined just by combining functions. For example, we can create a tower by placing a (linear or otherwise) sequence of spirals around a circle. We can also simplify its use by including higher-level parameters, such as the number of spirals s and the number of turns t of each spiral:

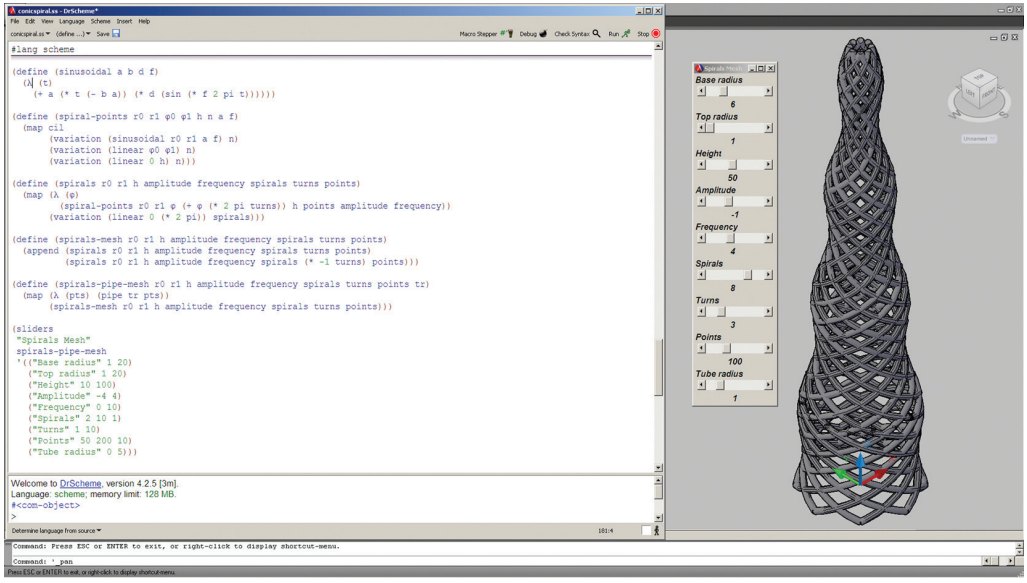

Figure 2
The VisualScheme IDE running on top of AutoCAD. The sliders window allows quick experimentation of the function parameters, by regenerating the corresponding geometry in real time. It was generated by the code fragment visible at the bottom of the editor.

```
(define (spirals r0 r1 h d f s t n)
  (map (λ (φ)
         (spiral r0 r1 φ
                 (+ φ (* 2 pi t))
                 h d f n))
       (variation
         (linear 0 (* 2 pi))
         s)))
```

In order to create a mesh of spirals that turn in opposite directions we combine two calls of the previous function, one of them with a symmetrical number of turns:

```
(define (spirals-mesh r0 r1 h d f s t n)
  (append
    (spirals r0 r1 h d f s t n)
    (spirals r0 r1 h d f s (- t) n)))
```

The points produced by the previous function can now be used to define a surface or a set of curves.

Figure 2 shows VisualScheme running alongside with AutoCAD which is used to visualize the resulting geometry. Also visible in the image is a small window with sliders that allows the user to quickly experiment different values for the function parameters just by moving the sliders. In order to generate the set of sliders, it is only necessary to write a small code fragment that references the function to test and the names and ranges of the parameters.

EVALUATION

The previous section described VisualScheme, a research project whose main goal is the production of a pedagogical IDE for generative design. To compare VisualScheme with a VPL, we will use Grasshopper as reference and we will consider the three fundamental dimensions of a PL: primitives, combinations, and abstractions.

In what regards primitives, we can conclude that Grasshopper is currently in a very good position: it implements a huge set of primitive elements, some of them with a high degree of sophistication, allowing an important reduction in the implementation effort, a significant advantage over VisualScheme that currently does not implement as many primitives as Grasshopper.

In what regards the combination mechanisms, Grasshopper relies on an extremely simple metaphor: primitives can be combined by connecting the output of a primitive to the input of another. The connections allow data to flow from primitive to primitive, until it reaches the end of the graph, usually, in primitives that create geometric models. The problem with this metaphor is that it makes it difficult to express some control structures, such as iteration or recursion. In other dataflow languages, these control structures require either dedicated mechanisms or the ability to create cycles in the graph of primitives. In Grasshopper, most components implicitly map their operations over sequences of values, obviating the need for many cases of iteration and recursion but, in the general case, it might be difficult or impossible to describe a computational process without textually scripting a specialized component, thus contradicting the visual nature of the language.

In the case of VisualScheme, the available combination mechanisms are the ones provided by the Scheme language which includes the ability to form complex expressions from simpler ones and several control and data structuring mechanisms, allowing many different programming paradigms. In our opinion, this is strictly more powerful than the single metaphor provided by the VPL side of Grasshopper.

Finally, in what regards the abstraction mechanisms, Grasshopper implements a special component, the cluster, which allows the user to select a graph of components (including other clusters) and to treat it as a single component. This can make a huge difference in the clarity of programs and improves the reuse of its parts. Unfortunately, this capability has not been always available. In this regard, VisualScheme provides several different forms of abstraction. In this paper we only presented examples

of procedural abstraction but the language also provides data and control abstraction.

In order to better evaluate VisualScheme, we have been working on a small number of parametric design exercises that we implemented both in VisualScheme and in Grasshopper. In a previous paper (Leitão et al, 2010), we reported our findings for a moderately complex task. Although the experiment was not representative, it showed that VisualScheme could be more productive than Grasshopper for that particular task. In this paper we focus on a simpler task but we are more exhaustive by comparing the previous VisualScheme program with different solutions that were developed by four Grasshopper users that have different degrees of expertise, from beginner to advanced.

The task used in the experiment was a simplified version of the mesh of conic spirals discussed previously: the users only had to implement a cylindrical tower based on the same concept of intersecting spirals. This tower is visible in the top left part of Figure 2.

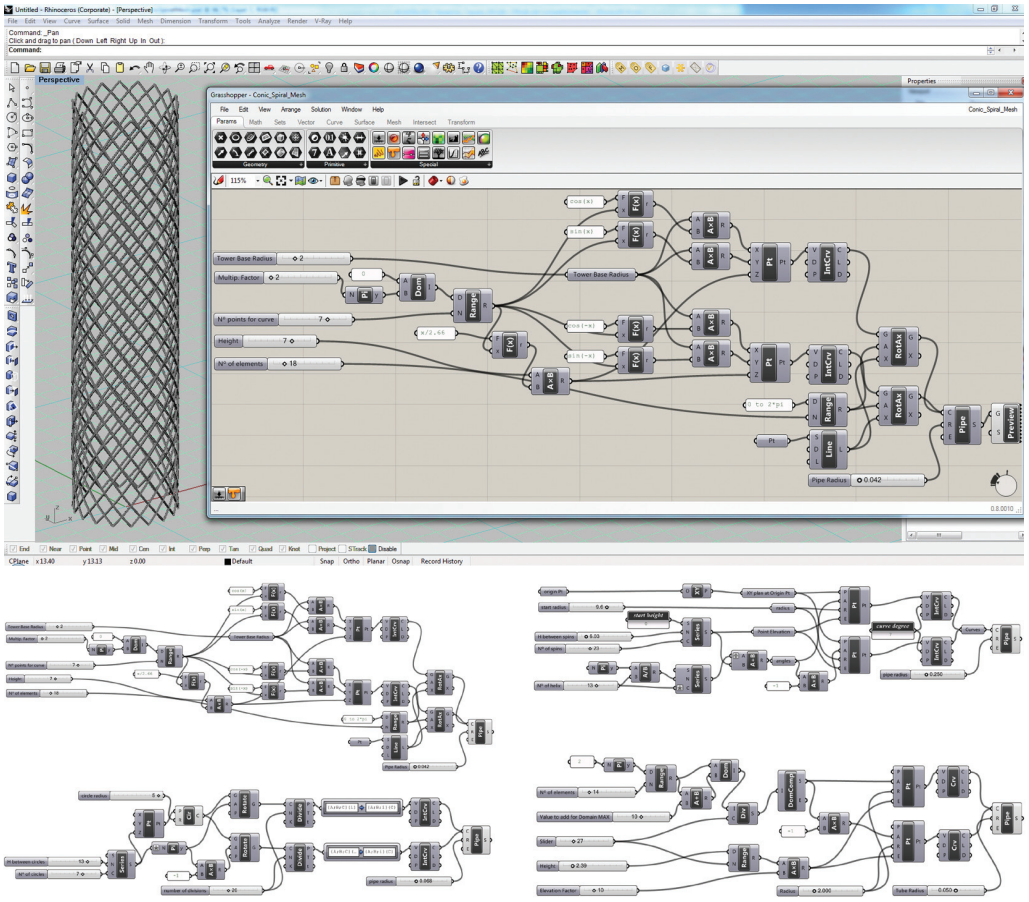

Figure 3
The Grasshopper programs encoded by four different designers for the mesh of conic spirals. Above: one of the Grasshopper definitions and its result in Rhinoceros. Below: the four Grasshopper programs produced (for reasons of space the images were reduced).

In what regards the Grasshopper implementations, it was observed that the design brief was easily accomplished, although, as expected, each designer followed a different approach and implemented a different program. Figure 2 shows the different solutions produced.

Comparing the VisualScheme program, showed previously, with the Grasshopper solutions, it is clear that, although not as aesthetically pleasing, the VisualScheme program is more synthetic, simpler, and easier to understand than the complex wire amalgam of the Grasshopper visual programs. Another clear advantage for VisualScheme is the freedom provided by the different data and control structures available, particularly, recursion. Without being able to use the general expressiveness of recursive definitions, Grasshopper users were frequently seen searching for a strategy that would allow them to achieve the same results.

The use of higher-order functions in VisualScheme made another important difference, as it allowed very quick experimentation of alternative designs. Grasshopper also allows the user to quickly changes the program inputs but to implement variations of a particular algorithm the user must design a different program, manually connecting and disconnecting the data flow wires from/to different components, a task that is much more time consuming and error prone.

Finally, the most relevant finding is the fact that the task was completed by the experienced VisualScheme user in a quarter of the time needed by the experienced Grasshopper user. As expected, the less experienced Grasshopper users needed even more time to complete the job. The analysis of the causes for the time gap shows that a significant fraction is due to the time consuming tasks of component placement, wire connection, data visualization, and commenting, that are required by the VPL. Modern TPLs, on the contrary, have very little requirements, allowing programmers to be much more productive.

CONCLUSION

This paper focused on the two different scripting approaches towards generative design: Textual, through a TPL, and Visual, through a VPL.

Given the obsolescent state of the majority of TPLs available for generative design, it is clear that a modern VPL offers a more attractive approach. However, it is inadequate to compare state-of-the art, domain-specific VPLs, such as Grasshopper, with old and general-purpose TPLs, such as VBScript. Instead, the comparison should use modern, domain-specific, TPLs.

Our findings show that Grasshopper does not scale well with the complexity of the tasks, due, in part, to its shortcomings in abstraction mechanisms and generic control structures and, in other part, to the time-consuming programming construction based on the manipulation of wires and boxes. It also was noticed that more complex programs become considerably harder to understand, a problem that can only be mitigated with extensive annotations.

The fact that it is possible to extend the capabilities of Grasshopper with textually scripted components is an important and useful feature that allows Grasshopper to transcend its limitations. However, it also shows that advanced users will always need to learn a TPL.

In this paper, we argued that modern TPLs with user-friendly IDEs can be much easier to program and understand than the older ones, and they can surpass recent VPLs, especially in complex tasks. These modern TPLs do not dispense learning their syntax but what might seem as a steeper learning curve quickly provides a good return on the investment. Unfortunately, most of these TPLs lack domain-specific primitives, a fact that can significantly delay the scripting process because it forces the user to define all needed functionality.

VisualScheme (Leitão et al, 2010) is our proposal for applying the advantages of modern TPLs to the generative design domain. Based on DrScheme, a modern pedagogical IDE, it extends the Scheme programming language with predefined

domain-specific primitives for the generative design and provides a direct connection to AutoCAD.

This paper presented examples of VisualScheme programs and compared them to equivalent programs in Grasshopper. Based on the comparison results, we argued that algorithms for generative design can be easier to implement, understand, and modify in VisualScheme than in Grasshopper.

Programming languages for generative design are here to stay. However, in spite of their usefulness, there is an important feature that is still absent in most of them: portability. Currently, it is difficult to port a generative design program between different CAD tools and it is difficult to reuse generative design libraries in different programming languages. We plan to address this issue in the near future, by allowing the same program to be executed in the context of different CAD tools, such as AutoCAD and Rhinoceros, and by allowing the combination of programs that were written in different languages, such as Scheme and JavaScript.

ACKNOWLEDGMENTS

The authors are grateful to Brimet Silva and Susana Martins for their participation in the Grasshopper experience and for their helpful comments.

REFERENCES

Findler, C, Flanagan, F, Krishnamurthi, S, Felleisen 2002, 'DrScheme: A Programming Environment for Scheme', Journal of Functional Programming, 12(2), pp. 159-182.

Griffiths 2007, 'Game Pad Live Coding Performance', in Die Welt als virtuelles Environment: Birringer, Dumke and Nicolai. TMA Hellerau, Dresden.

Hudak, P, Hughes, J, Peyton, S, Wadler, P 2007, 'A history of Haskell: being lazy with class', Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III): pp. 12–1–12–55.

Katz, N 2010, 'Algorithmic Modeling; Parametric Thinking: Computational Solutions to Design Problems', 5th ASCAAD Conference Proceedings, Fez, Morocco, pp. 19-36.

Kolarevic, B 2000, 'Eternity, Infinity and Virtuality in Architecture', 22nd ACADIA Conference Proceedings, Washington D.C., USA, pp. 251-256.

Leitão, A, Cabecinhas, F and Martins, S 2010, 'Revisiting the Architecture Curriculum', in Future Cities, 28th eCAADe Conference Proceedings, ETH Zurich (Switzerland), pp. 81–88.

Locke, J 1690, 'An Essay Concerning Human Understanding', London.

Menzies, T 2002, 'Evaluation Issues for Visual Programming Languages', Handbook of Software Engineering and Knowledge Engineering, vol. 2, pp. 93-101.

Myers, BA 1990, 'Taxonomies of Visual Programming and Program Visualization', Journal of Visual Languages and Computing, 1(1), pp. 97–123.

[1] www.patrikschumacher.com/Texts/Parametricism as Style.htm