CAD Scripting And Visual Programming Languages For Implementing Computational Design Concepts: A Comparison From A Pedagogical Point Of View

Gabriela Celani and Carlos Eduardo Verzola Vaz
Abstract

This paper compares the use of scripting languages and visual programming languages for teaching computational design concepts to novice and advanced architecture students. Both systems are described and discussed in terms of the representation methods they use. With novice students better results were obtained with the visual programming language. However, the generative strategies used were restricted to parametric variation and the use of randomness. Scripting, on the other hand, was used by advanced students to implement rule-based generative systems. It is possible to conclude that visual languages can be very useful for making architecture students understand general programming concepts, but scripting languages are fundamental for implementing generative design systems. The paper also discusses the importance of the ability to shift between different representation methods, from more concrete to more abstract, as part of the architectural education.
I. INTRODUCTION

The insertion of new technologies in architectural education has been the target of many discussions and publications, and the theme of many conferences, especially in the 1980's and 1990's, when computers became accessible for most architecture schools. The central issue in these discussions has progressively evolved from the introduction of software packages with simple representation purposes, to those with more complex analytical purposes, and finally to those which could be truly integrated in the design process, allowing automated form generation and exploration [1]. The interest in generative software has become more evident after the introduction of digital fabrication machines in the architecture field, which allow free-form exploration and mass-customization in a way that has never been possible before.

Generative design systems have been described by Mitchell in his seminal paper “The theoretical foundations of computer-aided architectural design” [2] as devices that are capable of generating potential solutions for a given problem. Most generative design systems use parametric variation and/or transformation rules as their main strategies. Although they do not necessarily need to be implemented in the computer, it is possible to automate otherwise time-consuming, repetitive tasks to increase the complexity of compositions in a way that would be impossible by hand.

Until the 1980's generative design systems were hard to implement, because they required not only expensive hardware, such as graphic displays and high memory capacity, but also very specialized programming skills. One of the first practical guides for implementing computer-based generative design systems was Mitchell, Ligget and Kvan's book The art of computer-graphics programming [3], which used Pascal programming. After this book, other guides for implementing generative design have followed, such as Schmitt's [4], Coates's [5], Celani's [6], and Terzidis's [7]. Although using different generative strategies, they had in common the fact that they all used scripting languages that were available within CAD packages, such as Mini Pascal, GDL, Lingo, AutoLisp, VBA and MEL [8].

However, with the recent introduction of parametric tools and visual programming in CAD software the interest in textual programming for teaching generative design is decreasing. A search for papers containing the keywords “education” and “programming” in Cumincad’s database, for example, returns 19 entries between 1977 and 1993 and only one in 2003. A search in the same database for papers containing the keyword “visual programming” returns 4 papers between 2006 and 2011, and a search for keywords “education” and “parametric” returns 8 papers between 2003 and 2011. This shows a tendency to change the focus from textual programming languages to visual programming and parametric design in CAAD education.

The reason for that is probably related to the fact that it takes much more time to learn to use textual than visual programming languages.
Learning syntax rules and understanding a program’s object model can be very frustrating at the beginning. However, textual languages allow the implementation of much more complex generative strategies that include transformation rules, while the use of visual programming is usually limited to parametric exploration.

The present paper categorizes textual and visual programming languages in terms of representation method, and describes them, with examples of applications. Next, it compares the use of textual and visual programming languages for introducing computational design concepts for novice and advanced architecture students.

2. MODELING METHODS IN ARCHITECTURE

Different methods can be used for modeling architectural form. Mitchell [2] categorized them in three types: iconic, analogue and symbolic. Iconic models are very literal. Typical examples of their use in architecture are plans, elevations and scale models. These models involve scale (enlargements and reductions) and projection (3D to 2D) transformations. Mitchell [2] emphasizes the role of this type of model in the generative process; according to him, in iconic models “a particular state of the system actually ‘looks like’ the potential solution which it represents” (p.130). By looking at an iconic model we are able to foresee how a building will look like when ready.

In analogue models “one set of properties [...] is used to represent another analogous set of properties of the item being designed” (p. 130). Analog representations allow easy manipulation:

“Analogue generative systems often represent potential designs by settings of wheels, dials, sliding columns, etc. The operations performed to change the state of the system (that is to describe a new potential design) are thus mechanical, for example, the spinning of wheels, setting dials, sliding columns alongside each other.” (p. 131)

The representation of the Sagrada Familia produced by Gaudi with wires and sand bags is an example of an analogue model. In this representation the tension vectors are represented by the wires, whose shape and direction are analogue to the physical phenomenon they represent. As a result, the architect was able to find the (inverted) ideal shape for the vaulted structures.

Symbolic models use symbols, such as words, numbers and mathematical operators. In architecture, symbolic models are used mainly for simulations and evaluations of structural, acoustical, lighting and thermal performance. Symbols are typically displayed as mathematical formulae, tables, arrays and algorithms.
The three representation methods described by Mitchell have different abstraction levels: iconic representations are closer to reality while symbolic representations are very abstract. Analogue representations are in between.

Computational design concepts can be represented in the three ways, with different levels of abstraction (Figure 1). At a more concrete level, parametric relations can be specified directly on iconic representations. This is possible, for example, in certain CAD software that allow to visually specify parametric relations between drawing entities directly on the graphic screen, such as Micro Station and the latest versions of AutoCAD. Visual programming languages use analogue representation, in which icons are used to indirectly represent and manipulate drawing entities. Two examples of this type of representation are Generative Component’s Symbolic Diagram and Grasshopper. These environments allow to visually describe relations between entities, without the need to write code. Textual programming languages use symbolic representations, such as text and numbers, to describe and perform operations on drawing entities. Examples of this type of representation are all CAD scripting languages, such as Rhino Script, AutoLisp and VBA.

3. CAD SCRIPTING LANGUAGES

Scripting languages are programming languages that allow control within a program. Differently from most programming languages, they are interpreted by the software, and do not need to be compiled. Scripting languages can vary a lot in terms of syntax and structure, depending on the software for which they were developed. Some examples of scripting languages for CAD are Rhinoceros’s Rhino Script, Maya’s MEL, and 3DMax’s MaxScript. In the
teaching experiment described in this paper AutoCAD’s VBA, an object-based scripting language, was used. VBA scripts (also called macros) can be developed in the Visual Basic for Application Interactive Development Environment (VBAIDE), which is embedded in Microsoft Office applications (such as Word and Excel), as well as in AutoCAD (VBAIDE used to be installed by default with AutoCAD, but since the 2011 version it needs to be downloaded from Autodesk’s website and installed as a plugin).

Although it is not a compiled language and it does not allow creating new proper classes of objects or stand alone applications, VBA scripting language is very powerful for automating procedures within AutoCAD. Its development environment allows designing user interfaces in an easy and intuitive way, which represents a great advantage over AutoLisp. The language has typical conditional (if, then, else) and looping (for each, for next, do while, select case, go to) structures, like any other languages. Procedures can be grouped under functions or subroutines. The syntax is similar to other object-based languages, known as “dot syntax”, in which a dot after the name of an object is used to access its properties (e.g. line.color = red) and trigger its methods, followed by parameters (e.g. line.move frompoint, topoint). In order to use this syntax, it is important to know AutoCAD’s object model, a hierarchic structure that shows how collections and classes of objects relate to each other. For example, Model Space is considered a collection in which certain objects can be found, such as lines, polylines, 3dsolids, etc. It is recommended (but not required) that VBA scripts start with the declaration of variables, including the specification of their type (such as numbers, words or AutoCAD entities) and the definition of their accessibility (public or private). VBA uses arrays for storing data, such as the x, y and z coordinates of a point. Because VBA is a very structured language, teaching it requires the introduction of many programming concepts, even for developing very simple programs. Alternatively, it is also possible to use the SendCommand function to simply deliver strings of text to the AutoCAD editor’s command prompt. The strings can be mixed with variables, and although this type of scripting is a bit limiting, it is very easy for students with no experience in programming to understand.

Let’s take as an example a macro that inserts parametric lines with a simple interface with slider bars. Figure 1 shows the user interface and the corresponding code in VBAIDE. The user loads the macro from AutoCAD’s editor (Tools/Macro/Macros) and the interface pops up. The user then slides the bars to change the start and end points’ x and y coordinates, presses the button and returns to AutoCAD’s graphic area to see the resulting line. To try another result, the line must be erased and the program must be reloaded. The interface could contain an extra button to erase the line, but the line always needs to be deleted and redrawn. The code is very simple. It attributes the numerical values from the slider bar to the x and y coordinates of each point (point1 and point2) and then instantiates a line object from point1 to point2 in AutoCAD’s Model Space.
The biggest disadvantage of using a scripting language is the need to follow its syntax very strictly. Although most scripting environments, such as VBAIDE, highlight mistakes and have debugging tools, such as flags and variable watches, certain syntax mistakes are not automatically detected and can take too long to be found. Most students can be discouraged by this.

4. VISUAL PROGRAMMING LANGUAGES

Visual programming languages or VPL's, also called “diagrammatic programming”, allow users to create programs by moving and putting together program elements graphically rather than by typing code. They use an analogue, instead of a symbolic representation of entities (in Mitchell’s sense). Figure 3 shows an example of a VPL’s user interface. Their main characteristics are the use of “box-and-wire” visual language, the hierarchical organization of elements, which can be grouped to form sub-units, and the possibility of typing code in certain special components.
Green and Petre [9] have studied the psychological aspects of VPL’s by comparing some VPL environments in terms of their cognitive dimensions. According to them, using a visual environment for developing programs is easier for many reasons:

“… there are fewer syntactic planning goals to be met, such as paired delimiters, discontinuous constructs, separators, or initialisations of variables; higher-level operators reduce the need for awkward combinations of primitives; and the order of activity is freer, so that programmers can proceed as seems best in putting the pieces of a program together” (p.40).

In this type of programming there is no need to declare variables. The user can simply grab a parameter element and attribute a value to it. There is also no need to plan the order of actions, and there are no syntactical rules. Even when a code window is used, it is constrained to a specific point, so bugs can be easily located.

In regards to code evaluation, Green and Petre [9] state that “The less experienced the programmer, the smaller the amount that is produced before it must be evaluated. Novices need ‘progressive evaluation’, an environment where it is easy to check a program fragment before adding to it.” (p.8)

Based on empirical studies about the use of VPL’s, authors such as Saariluoma and Sajaniemi, and Green and Navarro (apud [9]) have concluded that the spatial reasoning present in the manipulation of visual diagrams functions as a support for developing abstract ideas when developing code.

They can help organizing ideas when planning a computer program, without necessarily following a linear path.

5. VISUAL PROGRAMMING FOR CAD

More recently, some CAD packages have introduced built-in visual programming environments that allow generative capabilities without the need of writing code. Besides the regular geometric display, these packages have a diagrammatic – not symbolic, as in programming code – representation window in which the user can manipulate entities using an analogue representation.

The user has simultaneous feedback of the results, which can be seen in the geometric model window. Mistakes can be easily located because when the user makes a wrong connection between elements they immediately become a different color, while in scripting languages often the user only notices a mistake when the code crashes.

Visual programming environments within CAD packages can be very effective for shape exploration, through real-time generation of parametric variations. Most of these environments can be used without writing any code, although their capabilities can be extended with the use of scripting.
Two examples of this type of environment are Generative Components’s (GC) Symbolic Diagram and Grasshopper. GC is a module of Bentley’s Microstation CAD package, while Grasshopper is a plug-in for McNeil’s Rhinoceros. The Symbolic Diagram in GC is described as “a view of the geometric and non-geometric features you are placing, in graph form. The features are capsules with the feature type noted underneath the feature name. (…) The lines connecting the features show any dependencies between features. The arrows show the direction of the dependency. The Symbolic Diagram visually expresses dependencies that may not be as apparent in the Geometric view, but which influence other dependent features and so the behaviors of the whole model.” [10] (p.13)

In Grasshopper, the diagram area is called canvas, and is defined as “the actual editor where you define and edit the history network. The Canvas hosts the objects that make up the definition.” [11]

In both cases, since the geometry is developed in a diagrammatic way, it is not stored in a regular geometric model file, but in a special type of file, called transaction file in GC and definition file in Grasshopper. A transaction file contains the instructions that will generate geometry. When you open one in GenerativeComponents, you see the working environment. It is comprised of the GenerativeComponents dialog, the Symbolic Diagram and a Geometric view.” [10] (p.13).

Figure 2 shows a diagram in Grasshopper for representing a parametric line, similar to the previous example developed in VBA. The representation is comprised of slider bars in which the user can set up values for the x and y coordinates of the two points, two components that receive these values and create the points, and one component that receives the points and “spits” the line. Components representing parameters or drawing objects can be simply dragged from the top menu and placed on the canvas, and then linked to each other with wires, in a very intuitive way. Unspecified parameters, such as the z coordinate, here, use a default value (zero, in this case). As the user changes the values in the slider bars, the line gets immediately updated in Rhino’s graphic display window. Although this is a very simple example, it shows how using a visual parametric modeling environment is more intuitive and easy to use than a scripting environment, because it does not require the introduction of theoretical knowledge about programming.

In Grasshopper it is possible to create conditional statements with the use of special components. Loops can be implemented by introducing a special component called Hoopsnake or by using VB scripting, but it is possible to subdivide objects – such as lines or surfaces – with special components without the need to write a loop. Finally, Grasshopper has ready-made components that automatically generate random numbers and numerical sequences typically used in generative design, such as Fibonacci series.
Visual programming environments for CAD can be very effective for design exploration, especially for the automatic generation of parametric variations of a shape. They can also be seen as a tool for representing design schemes in a diagrammatic way, which can be very useful in design education, considering that the diagram has become “the fundamental technique and procedure of architectural knowledge”, a “final tool … for architectural production and discourse” [12] (Figure 4).

### 6. CASE STUDY

An experiment with the use of a scripting and a visual programming language was carried out with second year architecture students at the University of Campinas (Unicamp) in a mandatory course, called “CAD in the creative process”. It consists of a two-hour session every week, for 15 weeks [13]. Classes have typically 30 students, with previous experience with 2D and 3D AutoCAD, but no experience with Rhinoceros or any programming language.

The course included the following topics:
Computer-aided architectural design: definitions and history.

Form generation strategies in architecture: symmetry, parameterization, randomness, recursion and substitution, fractals, rule-based design, shape grammars, performance-based design and evolutionary design.

Two practical generative design exercises. Usually only VBA for AutoCAD is used, but in this case both VBA and Grasshopper were used. In both cases only very simple controls, procedures and mathematical operators were introduced, to allow students developing experiments with parametric shapes. Conditional statements and loops were not introduced. Exercise 1 consisted of designing “a simple abstract composition” with the scripting language. Exercise 2 consisted of designing a “roof for the campus’ gate” with the visual programming language.

The course includes recitations about each computational design concept, followed by a short practical assignment and a brief research related to the concept. For example, for the recursion topic students develop a design assignment using an existing fractal-generation script. They were also required to find examples of applications of fractals in architecture.

The introduction of the scripting language and the visual programming language was done in one single session each. After the introduction of each technique, students were given about two weeks to develop a design exercise.

The analysis of the results was based on the two design exercises and students reports about the use of the two techniques in the creative process. Tables 2 and 3 show some of the designs developed by students.

During Exercise 1 students sent the instructor many questions about the scripting language and showed up many time during office hours to solve syntax problems. The number of student visits was much smaller during the second exercise, and students were able to solve problems on their own. Their reports confirm that they were much more enthusiastic about the use of the visual programming language, and that they did not like the scripting language at all.

For Exercise 1 students developed short scripts, most of them using parts of sample codes provided, and simply changing their parameters to produce design alternatives. For Exercise 2 the results were well the examples provided. Some students looked for tutorials on the Internet and developed very complex compositions.
In Exercise 1 most students used straight lines, except for one student, who developed a spiral based on Fibonacci series. In Exercise 2 most students used organic shapes, something that probably would have been more complex to define using the scripting language.

A comparison between students’ reports on computational design concepts in this class and previous ones showed that they developed a better understanding of the concepts of generative design and the use of computer tools for design exploration. In previous years students often mentioned that implementing generative design systems with a scripting language was interesting, but not something that they would do, whereas in this class most students acknowledged parametric modeling as a viable design strategy, and as something they would like to try in future architectural design situations.
In summary, the results show that the use of the visual programming language was more successful in terms of students’ enthusiasm, the complexity of the designs developed, and the understanding of computational design concepts.

7. COMPARISON WITH THE WORK OF ADVANCED STUDENTS

The comparison above showed that the use of visual programming can result in complex designs in a very short time. However, more advanced students who took a specific course on VBA CAD scripting for one semester were later able to combine visual and textual programming and develop programs that implemented more advanced generative concepts, such as the application of recursive transformation rules. Figures 5 to 7 show the implementation of three well-known shape grammars using Grasshopper’s visual language combined to VB scripting, by a graduate student at Unicamp. These definitions use conditional statements to recursively apply different rules to the compositions.

Figure 5: An implementation of the Hepplewhite chairback design shape grammar [14] in Grasshopper, by Carlos Vaz, a graduate student at Unicamp.
Figure 6: Implementation of the Mughul Gardens shape grammar [15] in Grasshopper, by Carlos Vaz.

Figure 7: Implementation of the Palladian shape grammar [16] in Grasshopper, by Carlos Vaz.
In the third example, the two first parts of the Palladian grammar were implemented. The grid definition is restricted to 3x5 units, but allows different proportions, controlled by slide bars. Next, the interior layout is defined, with either an I, a T or a + shape central hall.

Figure 8 shows another example of an implementation combining visual and textual programming. This example used an adaptation of a cellular-automata (CA) code by Terzidis [7], which was translated to Visual Basic and then implemented parametrically in Grasshopper. The use of slider bars allows to interactively controlling the number of iterations of the CA.

In all the examples above VB scripting was used mainly for controlling recursion or for implementing transformation rules that would require too many components if they were implemented simply with the visual programming language. On the other hand, slider bars and other interactive controls were easy to use in the the visual programming interface, and the visualization of the results could be seen in real time in Rhino’s graphic window. This would not have been possible in AutoCAD.

8. DISCUSSION AND FUTURE WORK

Based on Unicamp’s experience it was possible to infer the learning curve for the two programming techniques analyzed (Figure 9). The comparison between textual and visual programming languages showed that the later can lead to better results with novice architecture students. However, without any textual programming knowledge, applications are restricted to parametric explorations. Scripting languages, on the other hand, can be used to develop more complex implementations of generative design strategies that include the recursive application of rules, and can be combined to visual programming elements, which are more interactive and allow real time results. This conclusion is compatible with other authors’, such as Leitao and Santos [17], who have also pointed out the limitations of Grasshopper and suggested the use of textual programming to extend its capabilities (p.556).
Another interesting observation based on the pedagogical experiments herein described is the fact that it is important to introduce computational concepts along with programming techniques, so that students can use programming (either textual or visual) as a way to express their design ideas. But perhaps what is most interesting about scripting languages and visual programming languages is the fact that they use different types of representations of design ideas: symbolic an analogue. The ability to shift between them, mapping one into another, helps developing the capacity for understanding abstract concepts, which is extremely important in architectural design education.

ACKNOWLEDGEMENTS

The author acknowledges the students who have taken course AU303 since 2004 for their enthusiasm and interest in computational design concepts, and Carlos Vaz for his implementations in Grasshopper using VB script.

References


---

**Gabriela Celani**

School of Civil Engineering, Architecture and Urban Design
University of Campinas
951 Albert Einstein Ave.
Campinas SP 13083852
Brazil

celani@fec.unicamp.br

**Carlos Eduardo Verzola Vaz**

Department of Graphic Expression
University of Pernambuco
1235 Prof. Moraes Rego Ave.
Recife PE 50670-901
Brazil

ccevv00@gmail.com