Teaching Computer Science for Architecture

A PROPOSAL

ANTÓNIO MENEZES LEITÃO
Instituto Superior Técnico, Technical University of Lisbon /INESC-ID
antonio.menezes.leitao@ist.utl.pt

ABSTRACT:
Computers have profoundly changed the way architects work. Computer science is nowadays recognized as one of the fundamental sciences that must be taught in architecture. Unfortunately, computer science is usually taught just like Physics or Probability Theory, without really preparing the students for the tremendous impact that it will have in architecture in the near future. In this paper we analyze that impact and we discuss some of the approaches that are currently being used for teaching computer science in architecture. Our main contribution is a proposal for teaching computer science in architecture using the principles of functional programming and, particularly, higher-order programming, while avoiding being dependent of specific CAD tools. We claim that this approach gives the student the ability to think, design, and explore designs more effectively than using previous approaches. We validate our claims using data from our own teaching experience during the last five years.

KEYWORDS:
Architecture, Generative Design; Computer Science; Learning

1. Introduction

Computer science is changing the way architects design. Instead of going directly from the idea to the design, computer science allows for an intermediate step, where an algorithmic description of a design is produced by the architect. This algorithmic description, which can be either a textual program or a visual program or both, is then executed by a computer, producing the design.

One might wonder if this intermediate step is not actually slowing down the design process and, in fact, for very straightforward designs, it does slow down the process. On the other hand, there are many other situations where this step provides considerable advantages, for example (a) when there are tasks of a repetitive nature, (b) when the designer wants geometric shapes that are not directly available in the CAD tool, (c) when the designer wants to inject some randomness in the design, (d) when the designer wants to quickly experiment different variations of the same design, and (e) when the design is also dependent of a performative process. In all these (and many other) cases, programming provides the designer with a very powerful tool for achieving better designs with smaller costs.

Unfortunately, programming is not an innate capability. It is, however, a learnable skill and, as long as computer science courses are properly designed, everybody can learn it. In this paper, we discuss this learning process in the context of architecture. In the next section, we describe some of the computer science courses that are already available for the architecture curricula. We will then explain our own proposal for teaching a computer science course for architecture.

2. Computer Science Courses for Architecture

Nowadays, there are several universities offering architecture courses that include computer science techniques. MIT, for example, offers several different courses in this area, including *Introduction to Computation in Architectural Design* and *Design Scripting*. The first one teaches building information modeling, generative methods, prototyping, shape calculation and simulation, focusing on the use of Revit. In the final part of the course, students learn shape grammars in the context of AutoCAD (using the NITRO plug-in). The second course teaches basic programming concepts and representation of formal design knowledge, including parameterized objects, procedural representation of form, typology and architectural grammar, and related topics. This course is divided into two parts, one where students learn RhinoScript in order to work with Rhinoceros 3D and the other where they learn Processing so that they can work with Arduino.

In most cases, these courses take advantage of the programming languages that are available in most CAD tools, namely RhinoScript and Grasshopper for Rhinoceros 3D, VBA and AutoLISP for AutoCAD, MEL for Maya, GDL for ArchiCAD, etc. This has the obvious advantage of providing the students with a working environment that is directly attached to a tool that, supposedly, they know how to use. On the other hand, this is also the cause of an important problem that affects all these courses: students learn how to program for a specific CAD tool but, in general, it is very difficult for them to program for a different CAD tool. In other words, they cannot easily migrate to different programming environments.

There are at least three aspects to this migration problem. The first aspect is that students tend to become addicted to the first language they learn. This means that students that learn, for example, RhinoScript, cannot easily program in, for example, AutoLISP. This problem becomes even bigger for programming languages that have different syntactic paradigms, as it happens between visual programming languages, such as Grasshopper, and textual programming languages, such as RhinoScript. In order to avoid this problem, some schools teach more than one programming language or CAD tool. The University of Campinas, for example, is currently experimenting teaching both VBA and Grasshopper (Celani, 2011). The University of East London teaches NetLogo 3D, VBA, RhinoScript, AutoLISP, and MEL. Unfortunately, this requires additional teaching time or, alternatively, less teaching material related to computer science.

The second aspect of the migration problem is related to the modeling primitives. Each CAD tool provides different modeling primitives with different expressive power and this explains why certain CAD tools are known for its good capabilities in, e.g., mesh modeling, while others are known for its bad capabilities in, e.g., boolean operations. In general, users quickly learn which modeling primitives are most effective and which modeling primitives should be avoided. This has the nasty consequence that users tend to adopt the modeling techniques that they know are a good bet for the particular CAD tool they are using, instead of focusing on the modeling techniques that better solve the design problem they have. Again, this makes it more difficult for designers (and their programs) to migrate to different environments. The obvious solution is, once more, to expose students to several different CAD tools and some schools follow this approach.

The third aspect of the migration problem is related to the use of libraries. It is not cost-effective to write every generative design program from scratch. Instead, as it happens in other areas, designer should use libraries that provide a significant part of the needed functionality. This is already visible, for example, in Grasshopper and Rhinoceros 3D, where many programs depends on plug-ins such as Karamba, for structural analysis, or Galapagos, for optimization. Unfortunately, these libraries are deeply attached to some particular CAD program and are very difficult to use in other CAD tools.

In this paper, we discuss the first two aspects of the migration problem in the context of introductory computer science courses. The third aspect is still highly relevant but it is not usually addressed in such introductory courses, becoming important only in more advanced courses. Nevertheless, we will address it in the conclusion.

In short, the problem we want to address is the following: by using a particular CAD tool and the programming language provided by that particular CAD tool, many computer science courses currently available for architecture become focused on teaching a specific tool, instead of the relevant computer science concepts. Moreover, this problem cannot be properly solved by teaching even more languages and tools, as this requires time that would be more useful for teaching computer science concepts.

3. Teaching Computer Science for Architecture - A Proposal

The previous section presented courses intended to teach computer science concepts to architects. It should be clear that teaching computer science without relating it to any task considered useful by the architects is condemned to be a failure (Duarte, 2005): students learn best when they see a connection between what they are learning and their current or future needs. Most of the courses we reviewed address this point by adapting the subject matter to the architecture field and by providing computer science examples related to architecture.

There is one problem, however, with almost all of those courses: they teach programming languages (either textual or visual) that are strongly dependent on a specific CAD application. As a result, students are capable of using that language and the modeling capabilities of that CAD application but it is usually very difficult for them to adapt to different languages and tools.

In order to solve this problem without introducing even more languages and tools, we have been developing a one-semester computer science course for architecture students that focuses on computer science concepts and avoids being dependent on CAD-specific capabilities. To this end, we have been using a carefully selected subset of the AutoLISP language, and a carefully selected subset of AutoCAD capabilities that are available (in similar forms) in other programming languages and CAD tools. The selected subset is, obviously, not as convenient as the entire set but it provides three important advantages: (1) we can spend more time on computer science concepts and less on specific language features, (2) students are motivated to extend the language with additional constructs definable in terms of the already existent ones, which is an important computer science skill, and (3) it reduces the inevitable mismatch students will face when migrating to a different programming language. In the next section we will document several examples of this approach.

It is important to note that the choice of AutoCAD was purely pragmatic: it was a request from the architecture department, as learning AutoCAD would be put to good use in subsequent courses. The choice of AutoLISP was less pragmatic and more pedagogical: of all programming languages available for AutoCAD, AutoLISP is the one where it is easier to explain fundamental computer science concepts such as abstraction, functions, recursion, and data structures. It is obviously possible to explain these concepts in other languages such as VBA but, in our opinion, it takes more time and certain concepts, such as higher-order functions are much more difficult to use.

We will now describe and illustrate the topics of the course we propose.

3.1 Fundamental Concepts

We begin by teaching the significance of algorithms for the rigorous description of processes and of programming languages as the medium for such descriptions. We then explain the syntax and semantics of programming languages and, particularly, of a restricted subset of AutoLISP that includes the parts that contribute for the recognized pedagogical capabilities of the Lisp family of languages (Chen, 1992; Berman, 1994; Felleisen, 2002). In fact, we present the AutoLISP language as a syntactical variation of the language of mathematics and we avoid all those AutoLISP constructs that do not contribute to this view, such as assignment.

3.2 Data Abstraction

We then move on to data structures and data abstraction. This is an important topic that paves the way for a clearer explanation of coordinates and coordinate systems. In spite of its basic representation as lists of numbers, we intentionally avoid using list operations to manipulate coordinates. Instead, we teach abstract data types and its realization as constructor and selector operations. Students learn how to describe positions in space and how to operate them in geometric terms, using rectangular, polar, cylindrical and spherical coordinate systems defined by the students themselves. It is also in this topic that we explain basic modeling operations, first in two dimensions (lines, circles, rectangles, etc.) and then in three dimensions (spheres, boxes, cylinders, etc.). Again, instead of focusing on the AutoCAD primitives, we immediately abstract these operations in functions, so that students do not need to remember the particular AutoCAD incantations needed for the implementation of the operations.

3.3 Functional Abstraction

This topic explains parameterized functions, using the Doric order as a motivating example. Here, students learn how to parameterize shapes and how to establish dependencies between parameters so that a particular architectural canon is achieved. As an example, Figure 1 shows several columns in the Doric style but where only one of them follows the proportions of the Doric order.

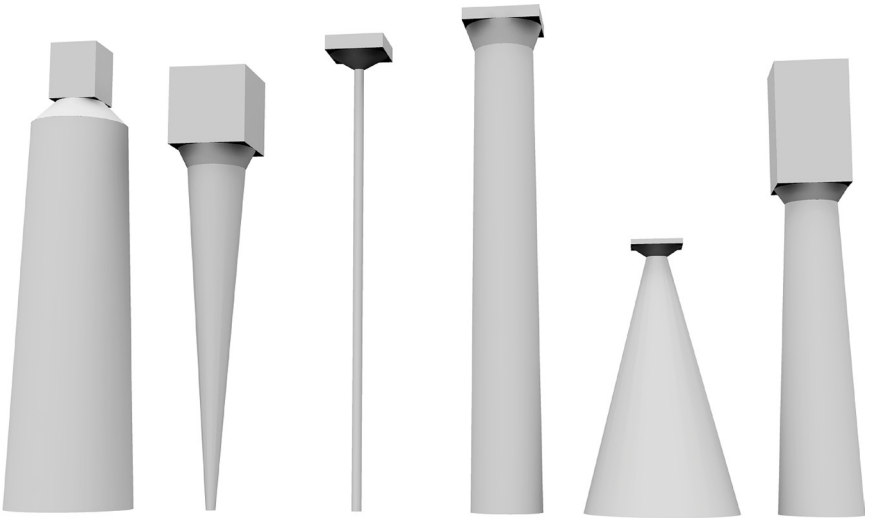

Figure 1 – Fully parameterized Doric columns. Only the fourth one follows the Doric order proportions.

In this part of the course, the only control structures that we teach are function calls, if expressions, and recursion. Note that we postpone teaching while-, repeat-, and for-loops, as these depend on assignment and assignment breaks the mathematical properties of the programs, making them harder to understand. In fact, recursion is strictly more powerful and, in many cases, easier to use that any specific looping construct, so that is what our students learn.

3.4 State and Randomness

Immediately after recursion, we teach computational state, in the sense of named values that affect a computation but that are also affected by that computation. This requires assignment but we restrict its use to functions that really benefit from it, such as random number generators. We also teach students to hide the assignment operations behind abstraction barriers so that they can forget that they are being used. As usual, students experiment these concepts in the context of some architectural problem. In this case, we ask them to model a (very simplified) city, exemplified in Figure 2.

Figure 2 –Recursive and randomized generation of cities.

In this modeling exercise, students combine several levels of recursion with the use of randomness, so that no two buildings are exactly identical. Besides learning how to use randomness, our students also learn how to control randomness. This is also visible in Figure 2: only a predefined fraction of the buildings are cylindrical towers and the height of each building, in spite of being a random value, is capped by a Gaussian distribution.

3.5 Recursive Data Structures

After acquiring some practice in the use of recursion, students learn recursive data structures, such as lists. We place a strong emphasis on the use of lists for separating the generation of geometrical coordinates from its use for some particular purpose. As examples, students are asked to implement sinusoidal curves and space frames, such as the one presented in Figure 3, where both concepts are used.

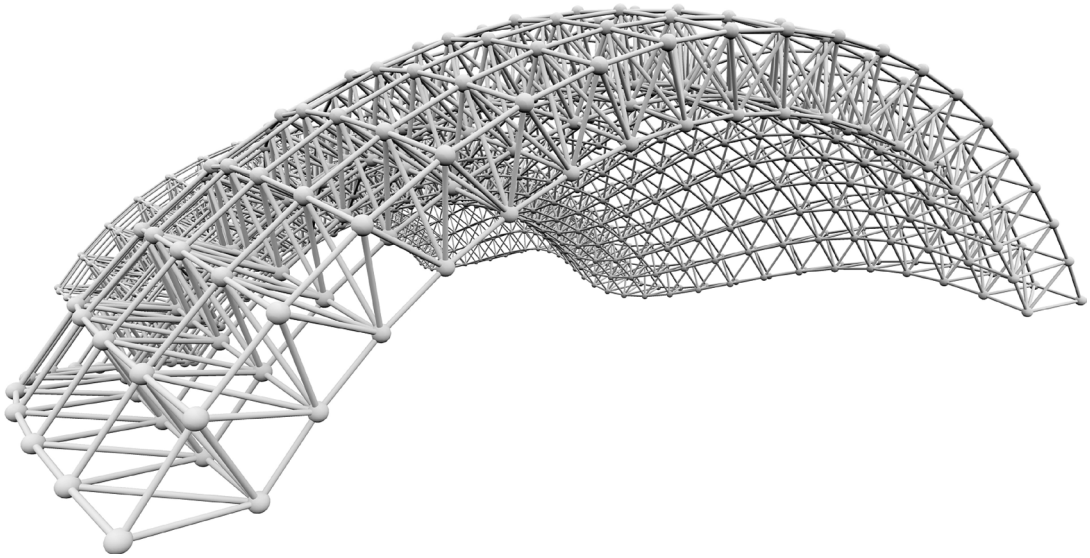

Figure 3 –A space frame with a circular arc whose radius follows a sinusoidal function.

3.6 Constructive Solid Geometry

The next topic in the course is Constructive Solid Geometry (CSG). Instead of explaining the specific operations provided by the CAD tool being used, we concentrate our efforts in describing a solid as an (infinite) set of points in space, so that modeling operations can be explained in terms of set operations such as union, intersection and subtraction. In order to allow a mathematical treatment of the subject matter, we also introduce the concepts of empty shape and universal shape, as identity elements of the set operations. We also demonstrate that without these special elements, that do not have correspondence in any CAD tool, it becomes more difficult to define CSG operations over sequences of shapes. This approach makes it clear to the students that algorithmic descriptions become easier to develop when we follow a mathematically correct approach instead of just using what is provided by the scripting languages of the CAD tools being used. Figure 4 shows an example of the use of the CSG operations for modeling a shelter.

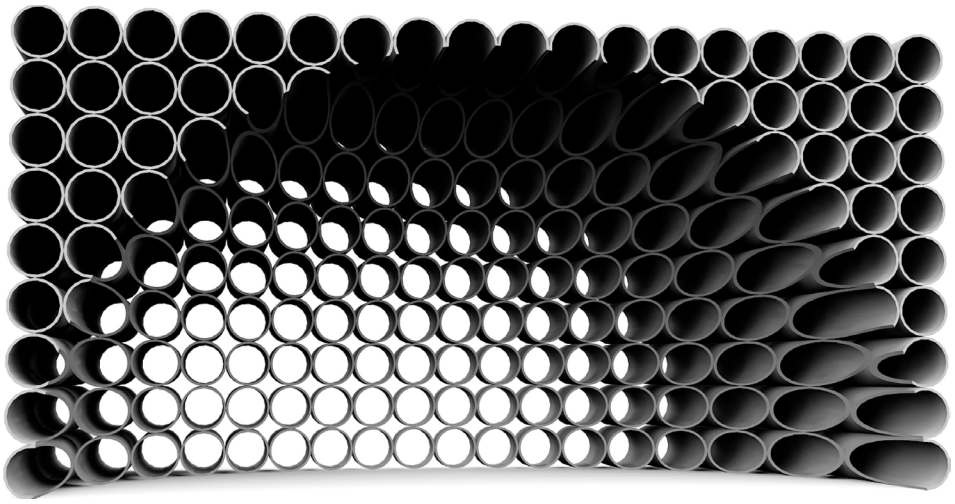

Figure 4 –A shelter made using cylinders, a sphere, and CSG operations.

Besides the basic CSG operations, we also introduce shape forming operations such as revolving, extrusion, sweeping, and lofting, as always, by teaching functional abstractions of the actual operations provided by the CAD tool and by providing actual architectural examples, such as the columns idealized by Gaudi for the Sagrada Familia cathedral.

3.7 Geometrical Transformations

We then briefly discuss geometric transformations, such as, translation, rotation, scaling, and mirroring. This is a relatively simple topic and we concentrate our teaching efforts in exemplifying its use in architecture. As an excellent case study, we explain the modeling process of the Sydney Opera House, presented in Figure 5.

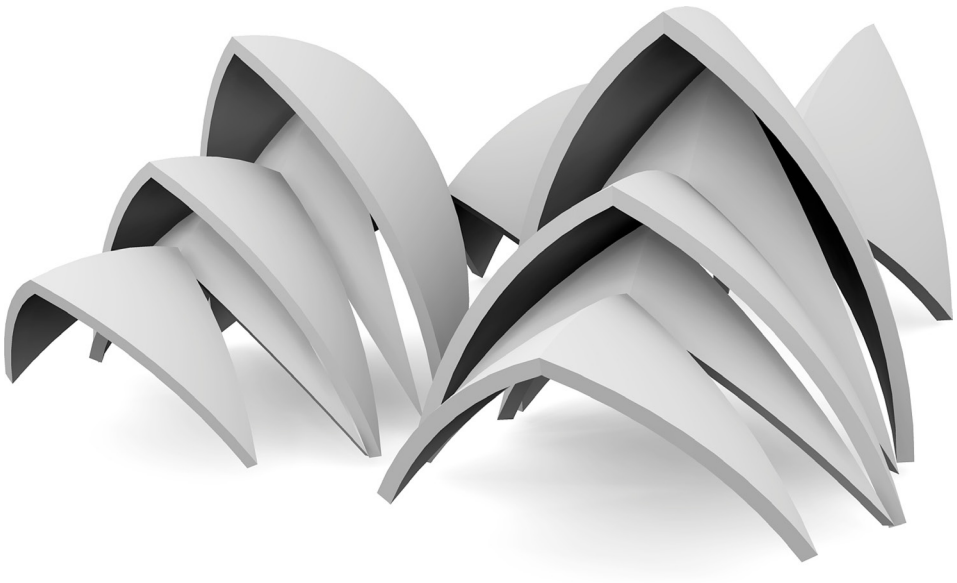

Figure5 –The Sydney Opera House as the result of geometric transformation of sliced sphere shells.

3.8 Higher-Order Functions

At this point of the course, students already acquired a strong set of modeling approaches and they have been practicing them in the laboratories. It is then time to teach more advanced programming techniques, particularly, those that depend on the use of higher-order functions. These are functions that accept other functions as inputs or return other functions as output. Figure 6 illustrates this powerful concept: all presented balconies are modeled by the same higher-order function that accepts as argument the function that defines the shape of the balcony.

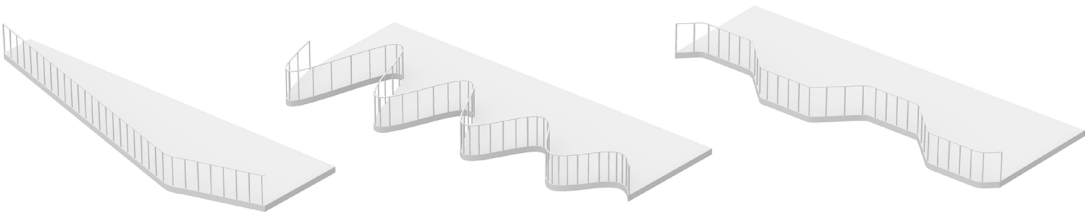

Figure 6 –Balconies modeled by a single higher-order function.

Instead of explaining the semantically incongruent predefined higher-order operations of AutoLISP, we explain their derivation from first principles, thus giving students the ability to define their own higher-order functions. Students also learn higher-order operations over collections, such as mappings, filterings, and reductions, a skill that becomes very useful to understand other languages, such as MEL or Grasshopper, that provide operators applicable both to scalars and collections.

As an application of higher-order operations, we explain a process for the automatic generation of three-dimensional models, a task that, previously, students had to painfully do by hand. One example of the outcome of this process is presented in Figure 7.

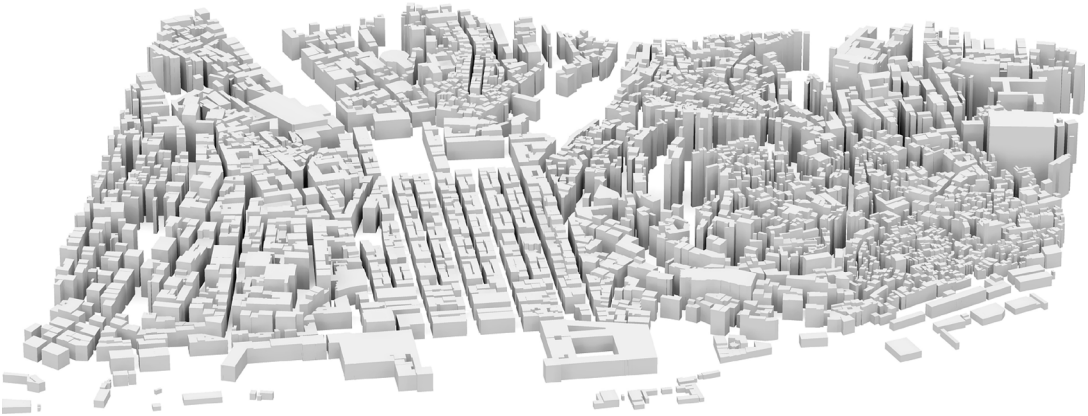

Figure 7 –Automatic generation of three-dimensional models from site data.

3.9 Parametric Curves and Surfaces

The final topics of the course are parametric curves and surfaces, which become trivial applications of higher-order functions: a function describing a mathematical curve or surface is repeatedly applied over a range of coordinates. Particular emphasis is placed in teaching students some useful mathematical curves and surfaces used in architecture that are not available in most CAD tools, such as the catenary, or the hyperbolic paraboloid. We also explain how to map functions over these curves and surfaces to produce more sophisticated shapes, such as the ones exemplified in Figure 8.

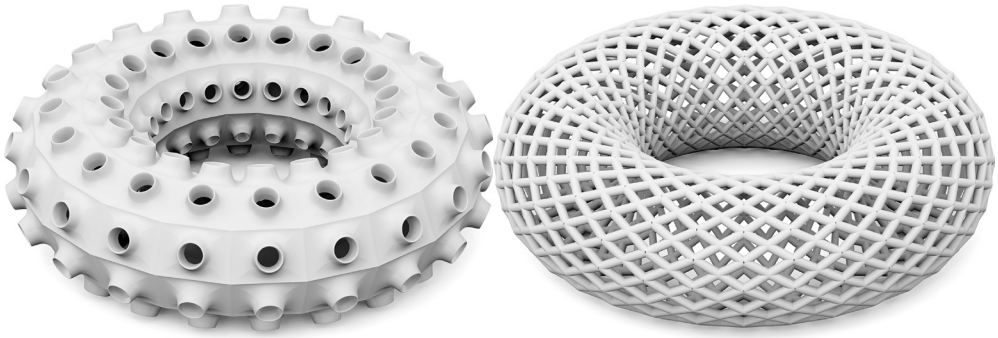

Figure 8 –Shapes that result from mapping simpler shapes over parametric surfaces.

4. Conclusion

In the last five years we have been teaching (and evolving) the course described in the previous section. The course structure was initially influenced by (Sussman, 1985) but we gave it an almost extreme bias towards applications in architecture. This is also the approach taken by (Woodbury, 2010) but where he prefers to embrace the modeling techniques promoted by the CAD tool being used (Generative Components), we opt for a more formal treatment of programming, with smaller emphasis on the CAD tool capabilities.

With time, the course we described has evolved to become ever more independent of specific CAD features, liberating students from becoming addicted to the CAD tool or the programming language being used. One might argue that this liberation is only apparent as, in fact, students were still working with AutoCAD and AutoLISP. This is partially true, of course, but there are two important points to be made here.

The first one is that we avoid AutoCAD-specific features by hiding them behind a functional abstraction barrier. This means, for example, that there is a single function to create a circle, with a specific set of parameters (center and radius), and students cannot use the many different ways that are available in AutoCAD (center and radius, two points, three points, etc.) but that might not be available in other CADs. We believe we have succeeded in this goal: some of our students are currently developing programs that work with Rhino without major problems in the use of the modeling primitives.

The second point is related to the use of AutoLISP. As we mentioned previously, we only use a subset of AutoLISP, avoiding all features that have unusual semantics, such as dynamic scope or operators with awkward evaluation rules. Forcing students to work with a restricted language might seem like a severe limitation but there are two important advantages: (1) by not teaching the intricacies of AutoLISP we have more time available to teach important computer science concepts, and (2) students become less dependent on specific details of AutoLISP and, thus, more apt to learn other programming languages. Although we do not have yet conclusive evidence regarding the second point, we already have some supporting data gathered from a survey involving 16 of our former students that were also enrolled in a RhinoScript workshop. At the end of the workshop, they were asked to evaluate not only the effort needed to migrate from AutoCAD to Rhino and from AutoLISP to RhinoScript but also their relative preferences regarding both the CAD tools and the programming languages. Table 1 presents the results.

Evaluation	Range	Average
Migrating from AutoCAD to Rhino is …	1-very hard, 3-neutral, 5-very easy	4.0
AutoCAD is preferable to Rhino	1-totally disagree, 3-neutral, 5-totally agree	2.4
Migrating from AutoLISP to RhinoScript is …	1-very hard, 3-neutral, 5-very easy	3.1
AutoCAD has a better IDE than Rhino	1-totally disagree, 3-neutral, 5-totally agree	2.1
AutoLISP is preferable to RhinoScript	1-totally disagree, 3-neutral, 5-totally agree	3.2

Table 1 – Evaluating the migration effort.

Although the number of enrolled students was not statistically significant it is still possible to draw some conclusions from the survey: (1) our students could easily migrate from AutoCAD to Rhino and they ended up preferring Rhino; (2) our students prefer Rhino's interactive development environment (IDE) but consider AutoLISP slightly preferable to RhinoScript; (3) for our students, migrating from AutoLISP to RhinoScript is not as easy as migrating from AutoCAD to Rhino but it is not difficult.

In spite of these results, we still find that most of our students become somewhat addicted to our AutoLISP subset. This seems to be unavoidable: learning the syntax of any programming language requires considerable effort and students tend to value that effort by always using similar languages. This is the reason why new programming languages, such as Java or C#, tend to adopt a syntax that is similar to those already established, such as C and C++. In the architecture field, however, programming languages are still very much attached to specific CAD tools and this creates a problem for architects that want to migrate to a different CAD tool but do not want to migrate to a different programming language.

We plan to address this problem by becoming even more independent of any CAD language and CAD tool. To this end, we plan to teach the same course topics but using Rosetta (Lopes, 2011), a multiple front-end, multiple back-end generative design tool that allows a choice of different programming languages, including Scheme, Racket, JavaScript and others, and a choice of different CAD tools, including AutoCAD and Rhino. By being independent of specific programming languages and CAD tools, Rosetta allows designers to use their preferred programming language to write programs for their preferred CAD application. Although still in its infancy, there are plans for providing Rosetta with a set of libraries usable by all front-end languages. This means that Rosetta will address all three aspects of the migration problem, namely, language independence, CAD tool independence and, finally, library independence.

Acknowledgements

We thank our students by the invaluable feedback they have given us since 2007. We also thank the anonymous reviewers for the insightful comments. The work reported in this article was supported by national funds through FCT under contract Pest-OE/EEI/LA0021/2011

References

Berman A., 1994: "Does Scheme enhance an introductory programming course?: some preliminary empirical results", ACM SIGPLAN Notices, 29(2), 44-48.
Celani G., Vaz C, 2012: "Cad Scripting and Visual Programming Languages for Implementing Computational Design Concepts: A Comparison From a Pedagogical Point of View", in International Journal of Architectural Computing, 1(10), 122-137.
Chen N., 1992: "High School Computing: The inside Story, in The Computing Teacher", 19(8), 51-52.
Duarte J., 2005: "Towards a New Curricula on New Technologies in Architecture", in Giaconia, P. (ed.), Script: Spot on Schools, Editrice Compositori, September 2005, 40-45.
Felleisen M., Findler C., Flatt M., Krishnamurthi S., 2002: "The Structure and Interpretation of the Computer Science Curriculum", in Functional and Declarative Programming in Education, 21-26.
Abelson H., Sussman G., 1985: *Structure and interpretation of computer programs*, MIT Press.
Woodbury R., 2010: *Elements of parametric design*. Routledge.
Lopes J., Leitão A., 2011: "Portable Generative Design for CAD Applications, in Integration through Computation": Proceedings of the 31st Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA), 196-203.