Revisiting the Architecture Curriculum

The programming perspective

António Leitão[1], Filipe Cabecinhas[2], Susana Martins[3]
[1]Instituto Superior Técnico/INESC-ID, Portugal, [2,3]Instituto Superior Técnico, Portugal
[1]https://fenix.ist.utl.pt/homepage/ist13451
[1]antonio.menezes.leitao@ist.utl.pt, [2]filipe.cabecinhas@ist.utl.pt,
[3]susanabmartins@ist.utl.pt

Abstract. *Nowadays, programming is quickly becoming part of the tool chest of the modern architect. Unfortunately, the architecture curriculum does not yet recognize its importance and usefulness or uses inadequate languages or programming environments to teach it.*
In this paper we argue that it is necessary to include computer science courses in the architecture curriculum and that these courses should be tailored to the needs of the architects. To help achieve this goal, we propose VisualScheme, an interactive programming environment that accompanies the architect from the learning phases to the advanced uses and that can be explored in pedagogic, research, and industry settings.
Keywords. *Generative design; programming; teaching; computer-aided-design.*

Introduction

When we look back at the History of Architecture, we find that drawing was not the only mechanism used to describe shapes. In fact, at certain times, the textual representation of form was much more common. Treaties like Vitruvius', Alberti's and Palladio's are proof that architectural matter can be encoded in textual language.

The interpretation of textual language, however, is time-consuming and ambiguous so it was only natural that Architecture practice evolved to use the clearer representations allowed by drawings and visual models. This form of representation is, nowadays, extremely widespread, being used in all CAD and BIM tools.

One of today's architects' cares resides in being able to describe and generate any kind of geometry he/she can envisage. However, many find the current drawing/designing tools very constraining: some forms are too complex to generate by hand, even using the best drawing tools. To overcome this complexity, architects are finding it worthwhile to revisit the textual representation of form but, this time, exploring the programming language approaches proposed by computer science. Just like the old Architecture treaties, a program is a textual representation of form, albeit with fundamentally different characteristics: a program uses a non-ambiguous language that can be quickly interpreted by computers.

Using programming tools, one can "transcend the factory set limitations of current 3D software" (Terdizis, 2003), allowing architects to develop complex, rigorous, and innovative forms, using any of the many strategies at hand, e.g., pure scripting

(McCullough, 2006), L-systems (Erstad, 2002), shape grammars (Mitchell, 1990), and cellular automata (Wolfram, 2002). Other important features are the exploration of the infinite variability of solutions allowed by parameterized procedures and the optimization processes that compute the best form to satisfy a given set of constraints (thermal, structural or other).

Regrettably, the teaching curriculum in architecture does not prepare students for programming and most architects find it to be a difficult tool to master. It is interesting to note that even designers that took courses in programming think that it is an exceedingly difficult subject. This is not surprising because, in many cases, they were forced to learn one mainstream language, such as C, C++, Java or C#, but these are languages that are too complex for people that lack an adequate background in computer science. As a result, most programming tasks that are useful in architecture are passed on to computer science specialists who know very little about architecture, creating an impedance mismatch between the architect's ideas and its implementation in some low-level computer language. It is then very difficult for the architects to recognize their ideas in the program and, consequently, it is almost impossible for them to adapt the implementation.

This state of things must change: architects should take advantage of better programming tools to improve their creative processes and the architecture curriculum must be updated to include appropriate computer science courses.

This claim is not new. In fact, in order to prepare students to the new paradigms, many schools started to include computer science courses. Unfortunately, teaching computer science without relating it to tasks 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 and this implies that any computer science course in an architecture curriculum should be tailored to the needs of the architects.

Programming Languages

Programming requires programming languages. It is our claim that a programming language usable by architecture students should (1) be pedagogic, (2) have a good interactive development environment, (3) have immediate applicability to the architecture domain, in particular, by allowing the generation of visualizable 3D models.

The third criterion suggests that we should teach using the scripting language of a given CAD system. Given the variety of CAD systems available, many different languages can be used for this purpose, e.g., GDL for ArchiCAD, AutoLISP and VBA for AutoCAD, RhinoScript for Rhino, MEL for Maya, etc. Unfortunately, none of these languages possess all the above mentioned qualities. GDL directly inherits from BASIC, a language that promotes bad programming habits (Gries, 1974) (Dijkstra, 1982). VBA, VBScript, and RhinoScript are different versions of VisualBasic, a language designed for rapid development of event-driven applications but, unfortunately, at the cost of hiding many important computer science concepts (Dingle, 2000). MEL is similar to Perl, a language with a messy syntax that is hard to learn and use by novices (Warren, 2001). AutoLISP is an old member of the Lisp family of languages (Steele, 1993) that is still very popular simply because it is attached to the most used CAD systems. Its huge success is, unfortunately, also its Achilles' heel: for compatibility reasons, AutoLISP did not evolve and continues to this day to present the user with the same set of obsolete linguistic features.

It is a fact that the scripting languages available for CAD systems were designed for users that just want to mechanize some tasks and that do not have a lot of time available to learn programming concepts or even the will to do so. Moreover, many of those scripting languages were developed a long time ago, at a time when programming language design was still in its infancy. The consequence is that current CAD scripting languages do not have the necessary pedagogical qualities.

On the other end of the spectrum we find Scheme (Kelsey, 1998), a programming language that is well known for being an excellent vehicle for teaching computer science concepts (Chen, 1992; Berman, 1994; Felleisen 2002). Scheme is used in many introductory and advanced courses in hundreds of high schools and universities and several companies also use it for in-house training. Unfortunately, most of the teaching material available for Scheme is not targeted at architecture students because it does not emphasize the relation between computation and generative design. This relation, to be properly felt by the student, requires a direct connection between the execution of programs and the visualization of the generated forms.

Scheme for Architecture

There were several attempts in the past to provide such a connection. We will now describe them.

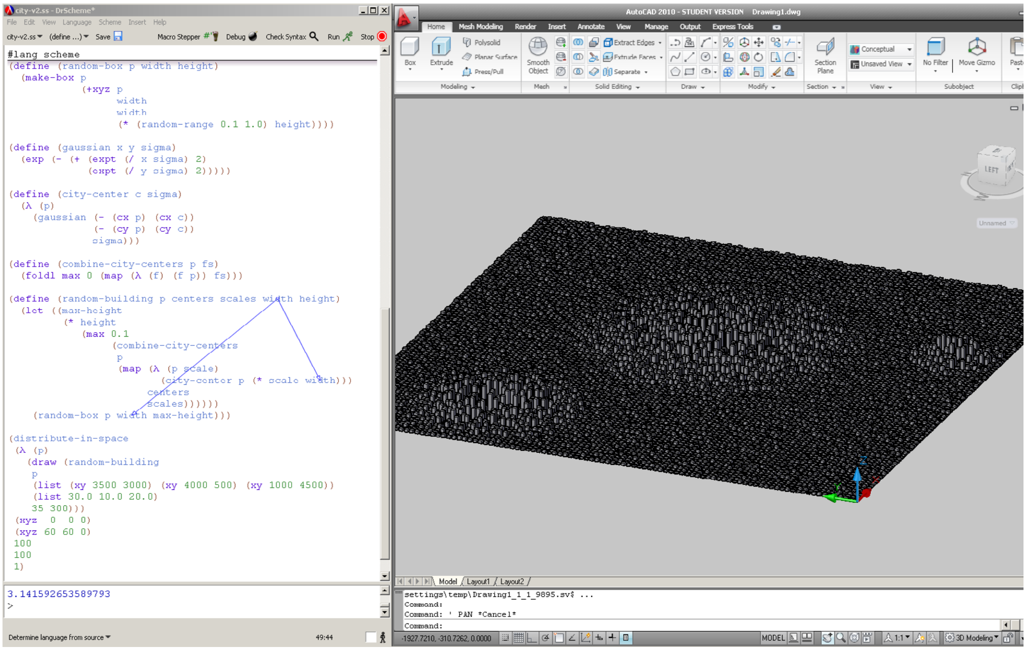
3DScheme (Martin, 1995b) integrates EdScheme with the ACIS 3D geometric modeling kernel and rendering is done in a dedicated window. Unfortunately 3DScheme, whose development stopped in 1997, lacks a sophisticated editor and an easy to use debugging environment. Another attempt was made with Scheme AIDE (Scheme ACIS Interface Driver Extension): a simple command-line interface to the ACIS 3D kernel that is based on Elk Scheme, an embeddable extension language for applications written in C or C++. Although used in some geometry courses, Scheme AIDE is better characterized as a simple testing tool for the ACIS kernel than as a pedagogic or professional environment. AL (Animation Language) (May, 1996) is also an extension of Elk Scheme that implements a variety of procedural modeling and animation techniques. AL uses a plug-in architecture that allows it to render the procedural models in an arbitrary number of rendering devices although, in practice, only RenderMan-compliant renderers are used. Fluxus (Griffiths, 2007) is a rapid prototyping, livecoding and playing/learning environment for 3D graphics, sound and games that runs on top of PLT Scheme. It can be used from its own livecoding environment or from PLT Scheme's IDE. All rendering is done using an OpenGL window. Fluxus is focused on live artistic performances and was not designed for CAD. Finally, SGDL (Rotgé, 2000) is an IDE for 3D solid modeling, also developed on top of PLT Scheme, which employs a particular modeling approach based on the composition of primitive recursive functions. This restricts its usefulness as a teaching tool as it makes it more difficult to experiment with other competing approaches to the modeling task. As in Fluxus, rendering is done in a dedicated window, unrelated to any CAD tool. A final obstacle is that this environment is proprietary and very little information about it is available.

In spite of demonstrating the feasibility of using Scheme for solid modeling, all the described systems lack one feature that is critical for our purposes: they are not integrated with any of the usual CAD tools used in the Architecture field, thus making it much more difficult for the architecture student to realize their usefulness. Moreover, most of those systems either are moribund projects, or have a significant lack of maintenance, or have a very limited number of users.

VisualScheme

Given that Scheme has good pedagogic qualities and that there were already successful experiments regarding its use for teaching solid modeling (Ferguson, 1990; Martin, 1995a), we decided to revisit the theme but, this time, avoiding being caught in the same trap. To this end, we needed a Scheme system that was being actively used and maintained and we needed to connect it to the most used CAD or BIM systems.

Fortunately, there was a system that satisfied some of our requirements: thanks to the efforts of the TeachScheme! Project, the DrScheme pedagogic programming environment (Findler, 2002) is


city-v2.ss - DrScheme*
File Edit View Language Scheme Insert Help
city-v2.ss (define ...) Save Macro Stepper Debug Check Syntax Run Stop
AutoCAD 2010 - STUDENT VERSION Drawing1.dwg
Home Mesh Modeling Render Insert Annotate View Manage Output Express Tools
Model Layout1 Layout2
Determine language from source
49:44
MODEL
3D Modeling

```
#lang scheme
(define (random-box p width height)
  (make-box p
            (+xyz p
                  width
                  width
                  (* (random-range 0.1 1.0) height))))

(define (gaussian x y sigma)
  (exp (- (+ (expt (/ x sigma) 2)
             (expt (/ y sigma) 2)))))

(define (city-center c sigma)
  (λ (p)
    (gaussian (- (cx p) (cx c))
              (- (cy p) (cy c))
              sigma)))

(define (combine-city-centers p fs)
  (foldl max 0 (map (λ (f) (f p)) fs)))

(define (random-building p centers scales width height)
  (let ((max-height
         (* height
            (max 0.1
                 (combine-city-centers
                  p
                  (map (λ (p scale)
                         (city-center p (* scale width)))
                       centers
                       scales))))))
    (random-box p width max-height)))

(distribute-in-space
 (λ (p)
   (draw (random-building
    p
    (list (xy 3500 3000) (xy 4000 500) (xy 1000 4500))
    (list 30.0 10.0 20.0)
    35 300)))
 (xyz  0  0 0)
 (xyz 60 60 0)
 100
 100
 1)
```

```
3.141592653589793
>
```

```
settings\temp\Drawing1_1_1_9895.sv$ ...
Command:
Command: ' PAN *Cancel*
Command:
```

```
-1927.7210, -310.7262, 0.0000
```

Figure 1
A VisualScheme program driving the generation of a city model in AutoCAD.

available, is open-source, has a large and growing community of users and is actively maintained, having been recently updated to implement the most recent Scheme standard (Sperber, 2007). This makes DrScheme an excellent choice for computer science courses and, in fact, it is being used worldwide, both in universities and in the industry. Unfortunately, DrScheme is not directly and immediately usable for CAD programming, making it unsuitable for an architecture curriculum.

In order to solve this problem we joined the best of both worlds, combining the most pedagogic programming environment with the most used CAD systems, producing an excellent learning environment for architecture students. More specifically, we integrated DrScheme with AutoCAD and we plan to do the same for Revit, ArchiCAD, and other CAD systems. We are also developing the necessary libraries that allow the programming environment to be used in research and industry settings. Given the environment's ability to transform Scheme programs into visualizable architectural models, we named it "VisualScheme".

The fundamental contribution of this work is that, by integrating DrScheme with known CAD systems, we greatly reduce the learning effort of the architecture students and, at the same time, we provide a modern tool for the exploration of computer-generated forms. Moreover, given the syntactical similarities between Scheme and AutoLISP, we expect to attract the large AutoLISP community to this project.

In order to benefit from future updates to the DrScheme programming environment, VisualScheme was implemented as an add-on to DrScheme, preserving all its tools, including the dedicated text-editor, the interactive listener, the syntax checker, the algebraic stepper, and the static debugger, which are now operating tools in the context of an architectural modeling process. This includes the ability to help the student in the learning process by catching the typical syntactic mistakes of beginners and pinpointing the exact source location of runtime errors.

Currently, VisualScheme can already be used as a 'drop-in' replacement for AutoLISP, the scripting language of AutoCAD: everything that can be done in AutoLISP can also be done in VisualScheme and we even provide an emulation mode where old AutoLISP programs can run. [Figure 1] presents an image of the VisualScheme programming environment connected to AutoCAD 2010, testing a simple program that models the distribution of skyscrapers in a city. Our goal, however, is to teach Scheme combined with a 3D modeling abstraction that is applicable to many CAD programs.

In the next section we describe the architecture of VisualScheme.

Software Architecture

The guiding requirement for establishing the software architecture of VisualScheme is the need for integration with different CAD systems but without making the user aware of the details of that integration.

To satisfy this requirement we introduced an abstraction level that isolates the user from the specific CAD system he/she is using. This abstraction operates as a front-end that presents the user with a set of common data types (coordinates, curves, surfaces, solids, etc.) and modeling operations (Boolean operations, sweeping, lofting, etc) usable in the context of the Scheme language. Each back-end is then responsible for translating from the generic modeling operations to the specific procedures of the corresponding CAD system being used. Therefore, different CAD systems are represented by different back-ends, and a program that only uses the operations available in the front-end can generate the same model in all supported CAD systems. However, the architect can also request access to the operations and data types that are specific of a given CAD system, thus exploring further functionality at the cost of less portability. More advanced users can also take advantage of DrScheme's capability for changing the front-end language, e.g., switching from Scheme to Java while maintaining the same back-ends. This way, the designer can focus on the modeling task instead of wasting time learning additional programming languages.

Given that back-ends entirely abstract the output device of VisualScheme, it is also possible to use dedicated back-ends for more specialized task such as direct rendering using, e.g., LuxRender, or for exporting the generated 3D model to a file format accepted by some other CAD or render program, such as X3D or POV-Ray SDL. To this end, it is the back-end responsibility to implement the communication between VisualScheme and the current worker program being used. This communication can be done using inter-process communication (IPC), files, or any other means.

This architecture of VisualScheme is represented in [Figure 2].

At the moment, the most well-developed

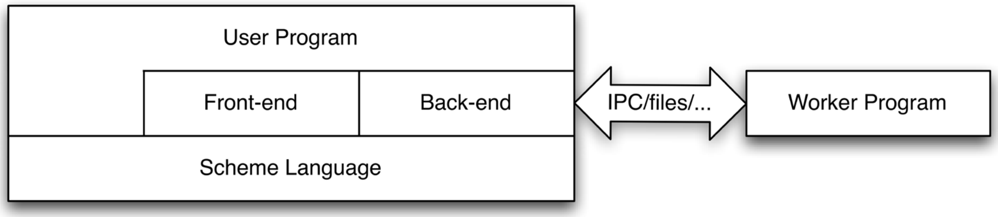


Figure 2
The software architecture of VisualScheme.

back-end deals with AutoCAD and implements a bi-directional direct connection that allows VisualScheme to send the generated models to AutoCAD and also retrieve drawing information from AutoCAD. The communication is done via a foreign function bridge using ActiveX and the Component Object Model (COM) as the inter-process communication mechanism.

Evaluation

Although this work is still in its infancy, we have already done a preliminary evaluation and the results allow us to be optimistic about the usefulness of VisualScheme.

The first result is the smoother learning curve of VisualScheme when compared to AutoLISP, especially in what regards the removal of bugs that are inevitable in any software development process.

Our experience regarding teaching AutoLISP tells us that one of the most frequent mistakes that students make is to accidentally misspell the name of some variable or function. Unfortunately, the AutoLISP environment will silently accept the mistake and will try to run the program. Obviously, something will go wrong but, in general, it will not be easy for the student to understand the cause of the error. In this regard, the syntax checker and the static debugger provided by VisualScheme will immediately point out to the student the cause of the error even before running the program. This capability extends to our emulation of AutoLISP and this has already proved to be very useful: VisualScheme warned us of a bug that survived many years undetected in one of our legacy AutoLISP programs.

Other similar problems that are automatically (and statically) detected by VisualScheme include syntax errors, wrong number of arguments to function calls, and a subset of type errors.

A second important result comes from a small number of experiments that we have been doing to compare VisualScheme with other programming environments. So far, we have only compared VisualScheme with Grasshopper, a graphical algorithm editor for Rhino intended for visual programming.

The comparison shows that, at this moment, for very basic tasks, Grasshopper is simpler to use and provides more immediate feedback than VisualScheme. This is explained by the fact that Grasshopper does not require previous programming skills and also because VisualScheme still lacks many of the operations that are already available in Grasshopper.

On the other hand, for more difficult tasks, we found that the abstraction capabilities of Scheme allow its user to easily manage the complexity of the program while the Grasshopper user becomes overwhelmed by the large number of components and connections required.

Another interesting result is that many of the problems where VisualScheme proves its value are also problems where Grasshopper is forced to use components that require textual programming, thus defeating the advantages of visual programming.

As a concrete example, [Figure 3] shows a particular instantiation of a parameterized model of Calatrava's Orient Station in Lisbon that was developed using VisualScheme and rendered using AutoCAD.

The parameterized model is described by a small set of Scheme functions that were developed by one of the authors in less than three hours. The entire program fits in two pages of text. For comparison, we asked a Grasshopper user to develop a similar model but it took her more than 3 days to complete.

Conclusion

Nowadays, many architects consider programming as a new tool that they would like to have in their arsenal. Many of them also think that programming is too difficult to master and, unfortunately, that is precisely the case with many of the currently available programming languages or programming environments. This is the problem that we addressed in this paper and VisualScheme is our proposal for

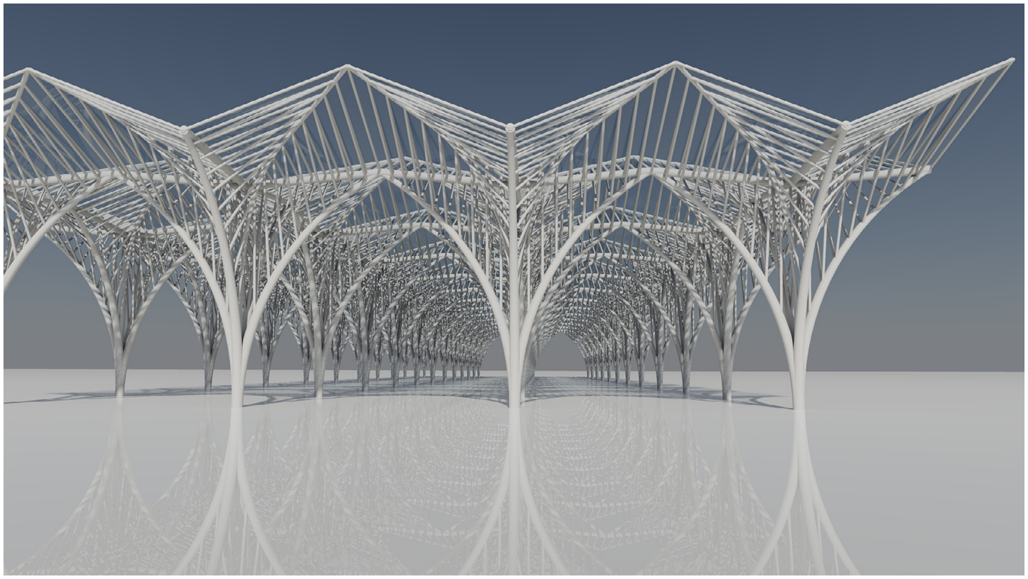


Figure 3
Orient Station in Lisbon, generated by VisualScheme.

solving that problem.

It is our belief that programming should be easy to learn, should be easy to use and should integrate well with the architect's tool chest. In order to achieve these goals, we claim that we need a pedagogical programming language, a good development environment, and a good integration with the common CAD tools used by the architect. The first two requirements are entirely satisfied by DrScheme, an interactive development environment for the Scheme language. The third requirement is the contribution presented in this paper.

VisualScheme extends DrScheme with a front-end/back-end software architecture that abstracts specific CAD programs and provides the user with a modern programming language tailored for the generation of models for several different back-ends.

Another important goal of the VisualScheme project is the prototypical implementation of a number of generative approaches to design, such as L-systems, shape grammars, and cellular automata, that will be used for teaching purposes and for giving the users a stepping stone for further research and experimentation. We are currently working on the implementation of some of these approaches and on the development of additional back-ends.

The end result of this project is an interactive development environment that accompanies the architect from the learning phases to the advanced uses and that can be explored in pedagogic, research, and industry settings. Instead of forcing architects to learn several inadequate or obsolete programming languages, we want them to learn a simple standardized modern language and then use it to program a variety of approaches in a variety of CAD systems.

References

Berman, AM 1994, 'Does Scheme enhance an introductory programming course? Some preliminary empirical results', *ACM SIGPLAN Notices*, vol. 29, no. 2, pp. 44-48.

Chen, MN 1992, 'High School Computing: The inside Story', *The Computing Teacher*, vol. 19, no. 8, pp. 51-52.

Dijkstra, EW 1982, 'How do we tell truths that might hurt?', *ACM SIGPLAN Notices*, vol. 17, no. 5, pp. 13-15.

Dingle, A and Zander, C 2000, 'Assessing the ripple effect of CS1 language choice', *Proceedings of the Second Annual CCSC on Computing in Small Colleges Northwestern Conference* (Oregon Graduate Institute, Beaverton, Oregon, United States), pp. 85-93.

Duarte, JP 2005, 'Towards a New Curricula on New Technologies in Architecture', in Giaconia, P. (ed.), *Script: Spot on Schools*, Editrice Compositori, Sept. 2005, pp. 40-45.

Erstad, KA 2002, *L-systems, Twining Plants, Lisp*, Cand. Scient. Thesis, University of Bergen.

Felleisen, M, Findler, RB, Flatt, M and Krishnamurthi, S 2002, 'The Structure and Interpretation of the Computer Science Curriculum', *Functional and Declarative Programming in Education*, pp. 21-26.

Ferguson, I, Martin, E and Kaufman, B 1990, *The Schemer's Guide*, Fort Lauderdale, FL: Schemers Inc.

Findler, RB, Clements, J, Flanagan, C, Flatt, M, Krishnamurthi, S, Steckler, P and Felleisen, M 2002, 'DrScheme: A Programming Environment for Scheme', *Journal of Functional Programming*, vol. 12, no. 2, pp. 159-182.

Findler, RB, Flanagan, C, Flatt, M, Krishnamurthi, S and Felleisen, M 1997, 'DrScheme: A Pedagogic Programming Environment for Scheme', *International Symposium on Programming Languages: Implementations, Logics, and Programs*.

Flatt, M, Findler, RB, Krishnamurthi, S and Felleisen, M 1999, 'Programming Languages as Operating Systems (or Revenge of the Son of the Lisp Machine)', *International Conference on Functional Programming*.

Gries, D 1974, 'What should we teach in an introductory programming course?', *Proceedings of the Fourth SIGCSE Technical Symposium on Computer Science Education SIGCSE '74*. ACM, New York, NY, pp. 81-89.

Griffiths, D 2007, 'Game Pad Live Coding Performance.' Originally appeared in *Die Welt als virtuelles Environment*, Birringer, J and Dumke, T and Nicolai, K (eds), TMA Hellerau, Dresden.

Kelsey, R, Clinger, W and Rees, J 1998, 'Revised^5 Report on the Algorithmic Language Scheme', *Higher-Order and Symbolic Computation*, vol. 11, no. 1, pp. 7-105.

Martin, E 1995a, *Getting Started with ACIS 3D Toolkit Using Scheme*, Fort Lauderdale, FL: Schemers Inc.

Martin, E 1995b, '*Solid modeling with Scheme*', *ACM SIGCSE Bulletin*, vol. 27, no. 1, pp. 336-339.

May, SF, Carlson, W, Phillips, F and Scheepers, F 1996, *AL: A Language for Procedural Modelling and Animation*, Technical Report OSU-ACCAD-12/96-TR5, ACCAD, The Ohio State University.

McCullough, M 2006, "20 Years of Scripted Space" in Mike Silver (ed.), *Programming Cultures, Architectural Design*, July/August 2006, pp. 12-15.

Mitchell, W 1990, *The Logic of Architecture*, MIT Press, London.

Rotgé, JF 2000, 'SGDL-Scheme: A high-level algorithmic language for projective solid modeling programming', *Proceedings of the Scheme and Functional Programming 2000 Workshop*. Montréal, Canada, Sept. 2000, pp. 31–34.

Sperber, M, Dybvig, RK, Flatt, M and van Straaten, A 2007, 'Revised^6 Report on the Algorithmic Language Scheme', *Journal of Functional Programming*, vol. 19, pp. 1-301.

Steele, G and Gabriel, R 1993, 'The evolution of Lisp', *ACM SIGPLAN Notices* 1993, pp. 231-270.

Terdizis, K 2003, *Expressive Form: A Conceptual Approach to Computational Design*, Spon Press (London and New York), p.72.

Warren, P 2001, 'Teaching programming using scripting languages', *Journal of Computing Sciences in Colleges* vol. 17, no. 2, pp. 205-216.

Wolfram, S 2002, *A New Kind of Science*, Wolfram Media, Inc., May 14,7.