A High-Level Pedagogical 3D Modeling Language and Framework

Filipe André Cabecinhas

May 14, 2010

1 Introduction

In the past, traditional architecture avoided complex geometric objects due to the lack of tools and excessive cost of materials and parts which had to be made specifically to build these objects. Nowadays, with computer aided design (CAD) and computer aided manufacturing (CAM) tools, it is possible to mass-produce unique parts for roughly the same cost of producing the same quantity of identical parts [7], thus allowing a higher degree of freedom to the architect.

We believe architects should be able to express themselves and realize their vision, whatever that may be. And, to model shapes based on mathematical functions, architects must use programming tools to aid them in building the desired model. It is known that most vanguardist buildings such as the "Pála Siza Vieira", from Portugal's Expo ′98 pavilion or the Guggenheim museums' buildings in Bilbao and New York are usually based on well-defined mathematical functions that can be computed, meaning that those buildings can be generated programmatically.

Experimentation is another important requirement that justifies the programmatic generation of forms: when idealizing the form of a complex geometric object, the designer may want to experiment with different parameters for its components. However, with a programmatic approach, an architect may design a fully-parametrized object definition and then just change the parameters until the generated form satisfies the initial vision.

Nowadays architects need to hire a group of programmers if they want to model a complex, out of the ordinary shape. This solution is costly and may cause a mismatch between what the architect envisioned and the end result. To solve this problem, architects must be taught how to program.

There has been an increasing effort on providing architects with programming background, by teaching them how to program and how to use these modeling tools. However, the programming focus has been mostly on automating simple tedious tasks, such as long sequences of commands.

This thesis presents the VisualScheme programming language, which intends to be a 3D modeling language oriented to architects. This programming language is a domain specific language (DSL) for 3D modeling built on top of the Scheme programming language. VisualScheme provides mechanisms for a programmer to define a 3D model of an object and then export it to a visualization program (the *worker program*). This 3D model is not composed of the definition of all the object's vertexes or triangles, but as a tree of high-level primitives and operations which will then be passed to a *back-end* which will convert the tree of primitives and operations to the commands to be sent to the *worker program.*

This paper comprises 6 sections: section 1 introduces the work and presents its goals. The related work is presented in section 2 and the four main stages of this project in section **??**. Section 3 presents the language proposed here, called VisualScheme, describing the technical aspects related to the implementation of the language, and how a VisualScheme program is organized. Section 4 presents a case study (the Gare do Oriente model) with VisualScheme, and compares it to a solution in Grasshopper. Finally, section 5 concludes the thesis and suggests some guidelines for future work.

1.1 Technical Contributions

With this work we designed and implemented the VisualScheme programming language with facilities for modeling 3D objects, as well as a *backend*for AutoCAD. The language allows an architect with average programming skills and a familiarity with CAD tools the ability to produce complex objects. This language will be able to export its objects to several formats, maximizing the rewards of learning it by allowing the programmer to export to whatever format the client desires.

We also took into account pedagogical aspects like the students' needs, as this language is aimed at classroom and industry usage. The language was tested by students throughout its development stages, which helped understand where inexperienced users have more difficulty.

2 Related Work

In the following sections, several existing 3D modeling programming languages will be analyzed from the programmer's point of view.

A program that draws a set of Greek columns was written in every analyzed language so as to compare the languages and their pragmatics.

2.1 AutoLISP and Visual LISP

Autodesk™[1] is one of the leading vendors of architecture-related software in the world. Its AutoCAD®[2] line of products is one of the oldest and most widely used in the area of CAD.

AutoLISP®[3] [3] is a Lisp-based programming language with a small kernel and a large set of geometric primitives used in AutoCAD for script-based geometric modeling. AutoLISP was designed to be used in conjunction with CAD tools like AutoCAD to automate repetitive tasks or to be used when precision is of the utmost importance and the manual tools don't fit the job..

Most architects only use AutoCAD with its point-and-click interface or, for the few (comparatively) of those that use AutoLISP, have some small AutoLISP scripts to automate some simple repetitive tasks. These tasks are usually very easy to accomplish either by hand or by programming.

With AutoLISP, programmers can write a script that uses AutoCAD's API to manipulate objects in a document. An AutoLISP script has access to a large number of AutoCAD's object creation functions, including functions to create primitive geometric objects like circles, lines, spheres, poly-lines and text. And operations on those objects, such as unions, intersections and subtractions. AutoCAD's objects are also available in AutoLISP, represented as entities.

2.1.1 Conclusions

Being the most used CAD program in the world, AutoCAD and its programmability features must be taken into account for this work. Despite showing its age, AutoLISP is still used by many groups of people with disparate programming skills, for automating all kinds of tasks in AutoCAD. Although widely used, its potential is very underutilised. Many architects either don't know it's available or only use it by emulating what they would do manually.

2.2 Generative Modeling Language

The Generative Modeling Language (GML)[4] is a simple stack-based language for describing 3D objects. Traditionally, lists of geometric primitives are used to model 3D objects. Instead, with a generative approach [11], geometric models are created by composing operations and the design becomes focused on rules for transforming primitive objects instead of focusing on combining these primitives.

GML is a PostScript based language which extends it with a set of operations, from vector algebra to the handling of polygons, including several conversions [6]. Being a postfix language, it's not very easy to learn to someone who is not familiarised with programming, in general, at least.

2.2.1 Conclusions

Despite being an unusual programming language (due to being stack-based), GML can accomplish

[1]`http://autodesk.com`

[2]`http://autodesk.com/autocad`

[3]`http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=1911627`

[4]`http://www.generative-modeling.org/`

what it's designed to: Being a terse postfix programming language, which enables programmers t describe shapes with relative ease and without having to model every used mesh. By modeling the objects using a high-level language, GML doesn't need as much information as a mesh-based program to store the objects.

2.3 Programming LAnguage for Solid Modeling

The Programming LAnguage for Solid Modeling (PLaSM) is a functional programming language based on IBM's®[5] FL [1] programming language. It is a functional design language for geometric and solid parametric design developed at the Roma Tre and La Sapienza universities [13, 12].

PLaSM takes on a programming approach to generative modeling, where geometric objects are generated by composing functions using the language's operators. It has a dimension-independent approach to geometric representation. PLaSM has support for higher-order functions in the FL style so, one way to look at it, is to see PLaSM as a geometric domain-specific language (DSL) for FL. No free nesting of scopes or pattern matching is allowed in PLaSM although an identifier can name any language object.

2.3.1 Conclusions

Despite the long learning curve for anyone not familiar with the FP or FL languages, PLaSM is a very expressive 3D modeling language where one can design big, complex parametrized objects with relative ease and make any change to the parameters, having the result just a render away. The completely functional approach, although better for parallelism and closer to the mathematical foundations of geometry, is harder to grasp for most people that learned to program in an imperative language (*e.g.* C, Java, *etc.*).

2.4 POV-Ray

The Persistence of Vision Raytracer (POV-Ray)™[6] is an open source ray-tracing program that grew out of David Buck's hobby ray-tracer, DKB-Trace [15]. With its large comprehensive library of objects, colors, textures, and many available special effects and rendering effects, POV-Ray is one of the most widely used ray-tracers around the world, and has a very large community [7] where users can find help.

2.4.1 Scene Description Language

POV-Ray's Scene Description Language (SDL) is a Turing-complete language that allows a programmer to describe the world in an efficient and readable way. The scene description language is a very basic language with support for abstraction mechanisms (*e.g.* macros and functions). It's mostly a declarative language, which means most of the objects are defined at one time in the program execution and aren't modified afterwards.

2.4.2 Conclusions

POV-Ray is a very good ray-tracer with a very helpful community built around it. It is capable of generating very high-quality photo-realistic renderings with small scripts due to its big high-quality object and texture library. The language in itself is declarative, although some support for imperative programming is present, in the form of macros. It is relatively easy for someone with some programming experience to start scripting POV-Ray, and start modeling parametrized objects programmatically with ease.

2.5 X3D

X3D is a run-time architecture and file format to represent 3D scenes. It is a royalty-free open standard ratified by the ISO committee [8], with various standardized encodings (*e.g.* XML and Classic VRML [9]) and bindings [10] for several languages (*e.g.* ECMAScript and Java).

X3D has a rich set of general parametrizable primitives that can be used for a variety of purposes in engineering, scientific visualization, CAD and architecture, and more. X3D was designed to be compatible with the legacy VRML standard and very extensible. Developed taking into account the COLLADA [2] standard for 3D representation,

[5]`http://ibm.com`

[6]`http://povray.org`

[7]`http://www.povray.org/community`

X3D is designed for real-time communication of 3D data while maximizing interoperability among different tools.

Additionally to the primitives X3D makes available for creating different shapes, there are nodes to group sets of nodes together for use as arguments, or for optimizations and transformations. A translation, rotation or scale transformation is always executed to a set of nodes and in regard to a certain point. The translation is absolute, with a translation vector.

2.5.1 Conclusions

X3D is a simple, web oriented 3D file format, designed for embedding 3D content in web-pages and publishing content for use in any program that implements this standard. The XML encoding facilitates integration in existing tools while the legacy VRML encoding helps people who write X3D directly. Despite some criticisms, The X3D standard is getting widely adopted, from 3D modeling programs to web browsers to GIS[8] programs. There is a lack of CSG functions, which makes it better suited for modeling some kinds of objects with the help of higher-level tools.

2.6 Grasshopper 3D

Grasshopper 3D is a plug-in for visual programming built on top of the Rhinoceros 3D modeling tool (Rhino). Rhino is a 3D modeling program which widely used in many design fields, such as marine and jewelry design, CAD/CAM and rapid prototyping. It's primitives are based on NURBS and it has several rendering options. Rhino is also used to convert between modeling program's file formats, since it can import from and export to a wide variety of formats. [14]

Grasshopper's programs are designed using a "boxes and arrows" model. Boxes represent primitives or operations and the arrows connect a box's output to another box's input. The inputs may also be fixed, using several widgets, instead of deriving from another operation's results.

Grasshopper lacks any mechanism for block abstraction in its programs. Programmers work around this by drawing delimiters over sets of boxes and inserting text in the model definition, but these have no semantic meaning for the program, which makes it easy to get these annotations out of sync with the program. Besides not being able to abstract, there is no equivalent, in Grasshopper, to higher-order functions.

2.6.1 Conclusions

Grasshopper is a simple, visual programming language, which is designed with non-programmers in mind. Any person who can use 3D modeling programs with a point-and-click interface (and therefore knows the terms used and the modeling concepts involved) can create a simple parameterized 3D model with constraints between its properties (*e.g.* the model of a building where one of its sides is based on a mathematical equation and its height is a function of its width). Unfortunately, like other visual programming tools, when a model's complexity grows, the lack of user-defined abstractions makes it difficult for a programmer to build complex parameterized models (it is even impossible to use a function (implemented as blocks) to parameterize any part of the model).

2.7 Comparative Table

In this subsection we present a small comparative table (Table 1) of the high-level characteristics of each of the analyzed programming languages.

These are very high-level characteristics of these languages, viewed by an architect's point of view. It is assumed the architect is marginally familiar with AutoLISP for the definition of each language's learning curve.

2.7.1 Available 3D Primitives

The primitives available from the analyzed programming languages were separated in three different groups: 2D primitives, 3D primitives and operations on objects. The languages were then compared and a minimized set of primitives was constructed. While, for example, AutoCAD has several kinds of line objects, we joined, in VisualScheme, these primitives into a single `line` object which can have multiple points of passage and exist in a 2D or 3D space. These primitives, in AutoCAD, are kept mostly for legacy reasons, as it evolved from 2D to 3D modeling and previous

[8]Geographic Information System

Feature	AutoLISP	GML	PLaSM	POV-Ray	X3D	Grasshopper
Paradigm	Multi-paradigm[a]	Stack-based	Function-level	Declarative	Declarative	Visual
Syntax	Parenthesized Prefix	Postfix	Prefix	Mixed	XML	N/A
Usage	Widespread	Limited	Limited	Widespread	Widespread	Limited
Learning curve	Short	Long	Long	Short	Short	Short
Expressiveness	High	High	High	Moderate	Moderate	Moderate[b]
Abstraction	HoF,[c] Function creation	HoF	HoF	Textual macros Functions	Node creation	C#, VB.NET Blocks, lines

[a]Functional and imperative paradigms, although some data structures are purely functional.

[b]Grasshopper may be more expressive if one takes into account C# and VB.NET blocks, which parts from the visual programming paradigm.

[c]Higher-order functions

Table 1: High-level comparative table between AutoLISP, GML, PLaSM and POV-Ray

tools and plug-ins must be kept working. This hinders architects in their daily work as they struggle to understand why a certain operation on a certain kind of line is not working as expected. The VisualScheme manual defines the available primitives and contains references to some of the languages where a form of the presented primitives can be found.

3 VisualScheme

The objective of this work is to create a domain specific language (DSL) for 3D modeling to enable architects to design buildings in a programmatic way, thereby reducing the amount of work needed to experiment with different and elaborated shapes and allowing the programmatic modeling of 3D objects.

VisualScheme programs are created by composing primitives and operations, creating a tree structure with primitives on its leaves and operations as the branches. Several library functions are available for the programmer to use, *e.g.* functions to iterate over sequences of points, generating them using interpolation, and coordinate transformation.

In the following subsections, we present the architecture of VisualScheme and its AutoCAD *backend*, as well as some examples of possible extensions to VisualScheme which can be provided by library writers.

3.1 Why Scheme?

The Scheme programming language was chosen due to several factors:

A Familiar Language

Because AutoLISP's users are already familiar with Lisp-like programming languages, another Lisp-like programming language such as Scheme can attract them, since it has the syntax, semantics and many of the features they already know how to use. Additionally, programs can be built on top of a modern programming language with many new facilities and powerful constructs for writing readable, compact code and protecting against certain kinds of bugs which plague AutoLISP programs. A Lisp-like language's learning curve will be much shorter than another language with a much more evolved syntax, at least for AutoLISP programmers. Taking all this into account, Scheme was chosen as a modern Lisp-like language, as our language of choice for parametric geometric modeling in the XXI century.

A Simple and Functional Language

Since Visual LISP is already built into AutoCAD and used around the world, it was decided to use a Lisp-like language for the 3D modeling language that will be designed, so as to gather the largest audience possible. As a Lisp dialect, the Scheme language shares much of its simple syntax: The program is written in S-Expressions, which are then evaluated by the interpreter or compiled and executed. Scheme is also an actively developed programming language with a continually revised specification [16].

Worker Program *Independent*

The high-level language must be independent of the *worker program*. It should not be important for the programmer to know if the *worker program* is a

CAD program, a 3D plotter, a simple visualizer, a flat file or a web-browser. Scheme does not depend on the *worker program*(unlike AutoLISP, which can only be used with AutoCAD). VisualScheme is divided in a *back-end* and a *front-end*, and the *back-end* is further divided in a *lower back-end* and a *upper back-end*. This *lower back-end* provides some Scheme functions to make the bridge between the Scheme process and the *worker program*, and the *upper back-end* maps the primitives of the language defined in this work into the *worker program*, using the functions defined in the *lower back-end*.

A Fast Language Implementation

Due to the sheer amount of data generated and operations executed when modeling a complex 3D object, the implemented language must not be a performance bottleneck. With a bottleneck in the interpreter or compiler, the resulting language would end up being unusable, because the ratio of the script execution time and the rendering would be high. If the language has a longer learning curve than simple tools like AutoCAD's basic drawing features, the programs would be used mostly for designing very complex objects, that would be (almost) impossible to design by using the non-automated AutoCAD drawing tools. This is especially significant because render times for drafts are practically instantaneous.

3.2 Architecture

The approach chosen to address the 3D modeling problem was to implement a compiler for a high-level 3D modeling language that is based on the Scheme programming language. This compiler will consist of two modules: The *front-end* and the *back-end*. The *back-end* will be dependent on the output the programmer wants to generate. Several *back-ends* may be implemented so the programmer just has to write one program and then export it to the desired format, as long as no *back-end*-dependent features were used.

3.2.1 Basic Architecture

The 3D programming language will be based on a Lisp-like language (the first implementation is based on PLT Scheme) and will communicate with AutoCAD (the first *back-end* to be implemented) via a foreign function bridge (the first *lower back-end* implementation uses Component Object Model (COM)[9] as the inter-process communication mechanism).

In the architecture, the *front-end* of the modeling language is implemented as a set of macros that transform programs written in the designed language into regular Scheme programs (that use the *back-end* and a run-time library that was written to support the language). The language's programs are compiled and run by a Scheme interpreter/compiler/virtual machine, outputting the designed objects to the *back-end* (AutoCAD, in the current implementation). The means of communication from the *front-end*to the *back-end*is a tree composed of VisualScheme's objects and primitives. This tree contains all the necessary information to build the 3D model. The *back-end*will, upon receiving the tree, convert the tree into a sequence of *worker program*commands which will be sent to it, creating the model in its working space.

3.2.2 *Front-end*

The *front-end* is responsible for most of the operations available in our language. Only after the objects are modeled in the *front-end*, with high-level primitives, will they be marshaled to the *back-end* and used according to what the programmer wants.

Every model starts in the *front-end*, by combining several primitives with operations which act on them. The core language *front-end* provides several primitives for creating and operating on points and vectors, as well as the core primitive objects. Several transforms and constructive solid geometry (CSG) operations are available for transforming and combining the primitive (or compound) objects.

Our Front-end

The *front-end* is responsible for defining the language that will be available to the programmer and also the main structures that will be used by both the *front-end* and the *back-end*. Our *front-end* defines several basic primitive objects and operations which work on them.

Transformations on objects, such as translations, rotations an scales are available, as well as opera-

[9] `http://microsoft.com/COM`

tions on sets of objects, such as unions, intersections and differences. These operations accept any created object as argument and, just like the primitives, must be implemented in every *back-end*.

3.2.3 *Back-end*

The language is made *back-end*-independent by having a distinction between the language core (which provides the primitives and operations that use them) and the *back-end* (which effectively serializes or draws the primitives).

The language core is the same regardless of what *back-end* is in use. A user won't have to change anything in his program in order to change the *back-end* if he only uses the language core.

The *back-end* may optimize the object and operations tree for itself (using more specialized *back-end* commands) and provide additional primitives for programs that don't need to be portable and require features that are only present in that *back-end*, as will be seen in subsection 3.3.

The AutoCAD Back-end

The AutoCAD *back-end* is divided in 3 components: A tiny DSL to facilitate writing code which uses several functions for COM operations from the MysterX library, bridges for the primitive AutoCAD COM methods which model several primitive objects on AutoCAD's side and the upper part of the *back-end*, that acts as a compiler which transforms our trees that are composed of VisuaScheme's primitives and operations to commands to be sent to AutoCAD via COM.

Adding support for additional features (that may be implemented in a further revision of the core language or core-dependent primitives) is simple: The new primitive must be added to the compiler driver so it can be recognized and modeled on the other side. The compiler driver will then call a specialized function to model that object whenever it is encountered.

3.3 Optimizations

In VisualScheme we have mechanisms in place to allow *back-end* developers to optimize the object and operation trees and allow the objects to be modeled in a more efficient way.

One optimization will be described: Collapsing the creation of several objects into the creation of one master copy and its subsequent copy and transformations on these copy.

3.3.1 Caching

The caching optimization is a *back-end*-dependent optimization which has been implemented as an optimization test-case in the AutoCAD *back-end*. This optimization was implemented in order to reduce COM calls to a minimum.

Our cache implementation is limited to caching only parts of an object the user asked to model (using the `draw` method that is exported by each *back-end*), with the cache being zeroed before the start of the `draw` routine. Support may easily be added to have a "world cache" which would memoize parts from every object, even if they're shared between objects in different calls to `draw`, even by having a two-level caching approach (a cache for global objects and a cache for local objects), similar to modern CPU [5].

To implement this optimization, the `draw` function that is exported by the *back-end* becomes a trampoline function which sets up the local cache and then dispatches to the effective modeling function of the *back-end* (in this case, the `draw*` function).

The optimized version of the `draw` function will have an entry point (`draw-top-level`) where the cache for the current (top-level) object is created (if there is a whole-program cache, it is just maintained through calls to `draw-top-level`), the object is then passed to the function which verifies if we have a cache hit or a cache miss (the `memoized-draw` function). If we have a cache hit, we copy our template and return the copy. If it is a cache miss, we will draw the object (using the same `draw` function as before the optimization) and register a copy of it in the cache for further use. The `draw-top-level` function was exported from the *back-end* with the name `draw`, in order to comply with the *back-end-front-end* interface. We chose to keep the previous `draw` function in order to change the minimum amount of code. I we only want a global cache and not a per-object cache, we can simply delete the `draw-top-level` function and export the `memoized-draw` function as `draw`.

3.3.2 Measures and Conclusions

The caching optimization incurred in a significant speedup in stress tests. Our test consisted of a grid of 50×50 cylinders modeled by creating a matrix of cylinder objects. At first, a cylinder object is created, in the lower-left corner of the grid. The structure that describes that object is then copied several times and wrapped in translation structures which move the contained object to another point in space. After creating a whole row of 50 cylinders, that row is then acted upon in the same way, copying and translating it 50 times to finalize the grid.

As we expected, the caching optimization is significant. Many more optimizations can be implemented in the AutoCAD *back-end* that may yield a speedup in most or only a small part of programs.

3.4 Extensions

Extensions are a key feature of VisualScheme. The language is designed in order to facilitate the development and integration of extensions, which can range from simple additional object definitions to fully fledged model analysis and verification systems. In this subsection we will show how to extend the VisualScheme language with additional objects. All the relevant data structures are available to enable a developer to write sophisticated systems on top of our object model and extend it with additional objects and operations.

Extensions may be made to the common framework or only to a specific *back-end*. An extension made to a specific *back-end* may use any function that is not on the common framework but is available in that *back-end* to the end user. To create an additional object definition, we just need to create a function with the object's parameters as arguments and make the function return a description of that object obtained by combining language primitives and operations on them. Any additional object model available in an extension will be made available by a Scheme library, which must be enabled at run-time.

3.4.1 *Back-end* dependent extensions

When designing a *back-end*-neutral language, we have to deal with the lowest-common-denominator problem: will the language be fully *back-end*-neutral and stick to what can be achieved by every *back-end*? To address this problem, we allow *back-ends* to extend the language's functionality with additional primitives or operations that may not be available in every other *back-end*. If a program uses these features, it may not be portable to some or all other *back-ends*. But on the other hand, it may make available several features that the user may want for that project. Also, the needed feature may be implemented from the most basic primitives (*e.g.* sets of triangles) of the other *back-ends*, if the need arises.

To add a *back-end*-specific feature to VisualScheme, we need to take care not to hinder other *back-end*. First we implement the primitive structure to represent lofted surfaces. This structure will contain a `start` argument, a function (`f`) which, given an argument, will produce a two-dimensional shape, a `step` function to step the argument and a `stop` function, which tests if we have arrived at the last shape of the loft. The dispatcher (the `draw` function) also has to support the loft primitive but that code was omitted for brevity (the dispatcher tests if the structure passed as an argument describes a lofted surface and, if it does, calls the `draw-loft` function, with that structure as its argument).

In the *back-end*, we unfold a list using the given high-level functions and create the corresponding shapes in AutoCAD. After creating the shapes that will guide the loft, a loft command is then issued to AutoCAD with the references to those shapes as arguments, in the order they were unfolded. The result is the creation of a lofted surface from a high-level function which receives a parametrization of the shapes and functions to generate the argument for the next function and a stop condition.

3.4.2 Conclusions

Our VisualScheme language is designed with extensibility in mind to allow for many future extensions such as adding objects, operations, and primitive hooks. The structure of the object and operations' tree is exposed to facilitate the development of analysis libraries on top of it, using the same model that will be marshaled to the *back-end*. We saw two examples of simple extensions, one to show how it is straightforward to add composite objects which can be used by any programmer and we saw how a *back-end* can add *back-end*-specific primitives with little additional effort.

Possible future extensions include support for "livecoding" [4], which comprises live performances involving music and visualizations, bi-directional coding, where the *back-end* would have hooks to recreate objects that were inserted in its workspace by means other than the VisualScheme language, or even computing deltas between the iterations of a program and only sending to the *worker program* the necessary commands to transform the objects in its workspace to those described in the new iteration of the program.

4 Evaluating the Work

The goal of this work is to design a 3D modeling language.

In order to evaluate VisualScheme and its suitability for 3D modeling and teaching, several approaches were taken: Several architecture students were asked to opine on the designed language and help steer the direction in which it is progressing. Early feedback from the part of a group of architects was positive and further developments are expected in order for VisualScheme to be usable in the architecture course mentioned before; Programs from that 3D modeling course for architecture students were converted into our language. We also created several simple test programs and a more complex one, a model of Lisbon's Gare do Oriente in VisualScheme to illustrate how we could easily create a model with high-level parameters which could be tweaked by the designer to create several different objects. Designers with experience in other modeling languages and programs were also asked to implement the same model and provide feedback on the obstacles encountered.

The model of the Gare do Oriente was created in the VisualScheme programming language, by programming in an incremental fashion. That way, the designer can easily experiment with several parameters and correct the model, if needed.

The model was divided in several basic shapes which were implemented using VisualScheme primitives. These shapes serve as abstractions over the VisualScheme primitives, to facilitate future changes, if needed.

After completing the Gare do Oriente model, the same test case was given to architecture students, with a background on Grasshopper. The students were supplied with some reference pictures, not the VisualScheme model.

The block structure is rigid and there are many connections which are difficult to track when the model's complexity grows. Even when the student tried to abstract parts of the model (by drawing a box around a set of blocks and labeling it (annotation)), the abstraction still forces the user to fully understand how it works.

Also, while in the VisualScheme version, the user only needed to change one function in order to change the way these plates were built, in the Grasshopper version, the user has to alter several connections and blocks, possibly losing track of some of them.

Eventually, the complexity forced the user to create some VB.NET blocks with custom functions for sweep-related problems which were encountered during the modeling. The final model also had some parameters which could be controlled by editing VB.NET blocks, which would behave as feature selectors.

If, during model building, the designer wants to add details to the model, the Grasshopper program must always be redesigned (except for very basic changes, such as changing the number of divisions of a line segment or controlling the size of basic shapes) and more connections must be added.

5 Conclusions

Most complex buildings are practically impossible to design manually, even with CAD tools. Some means must be available in CAD programs to allow designers to express themselves without that limitation. One way to remove that limitation is to provide designers with a programming language with which they can programatically define and combine objects to form complex figures.

The used programming language should be easy to learn to someone experienced with CAD tools. AutoCAD is an unavoidable player in the CAD business so the language should, at the very least, be marginally familiar to someone who already uses AutoCAD's automation features.

With this work we've produced a high-level, teaching oriented (which is also adequate for a production environment) language for 3D modeling, called VisualScheme. The language is a DSL built on top of the Lisp-like language Scheme, inheriting its simple syntax and semantics. Besides retaining Scheme's characteristics, VisualScheme adds the CAD functionality on top of it, by defining the basic primitive objects and operations. This language has a common *front-end* and multiple *back-ends* available for the model output.

With this language, we open the doors to a generative architecture with easy testing of parameters.With VisualScheme, we open this field to many architects who are already familiar with AutoLISP, and we provide a simple, high-level, easy to learn language to architects, whom we can teach how to use these tools to their advantage for rapid prototyping and the creation of parametrizable objects which can be used to explore a whole range of possibilities for a given architecture project, facilitating and fostering experimentation and exploration.

References

[1] A. Aiken, J. H. Williams, and E. L. Wimmers. The FL project: The design of a functional language, 1991.

[2] R. Arnaud and T. Parisi. Developing web applications with collada and x3d. *COLLADA*, 2007. *See `http://www.khronos.org/collada/presentations/Developing_Web_Applications_with_COLLADA_and_X3D.pdf`.*

[3] Autodesk. *AutoLISP Developer's Guide.* Autodesk, San Rafael, CA, USA, 2009.

[4] N. Collins, A. Mclean, J. Rohrhuber, and A. Ward. Live coding in laptop performance. *Organised Sound*, 8(03):321–330, 2003.

[5] U. Drepper. What every programmer should know about memory. Technical report, Red Hat, Inc., 2007.

[6] S. Havemann. An Introduction to the Generative Modeling Language: GML Tutorial. `http://generative-modeling.org/GenerativeModeling/Documents/gml_tutorial.pdf`, June 2003.

[7] G. Henriques, J. Duarte, and M. C. Guedes. Sustainable housing cells: Mass customization of sustainable collective housing. In *Proceedings of the 2009 International conference on sustainable development in building and environment*, Chongqing, China, 2009.

[8] ISO 19775:2004. *Information technology – Computer graphics and image processing – Extensible 3D (X3D).* ISO, Geneva, Switzerland, 2004.

[9] ISO 19776:2005. *Information technology – Computer graphics and image processing – Extensible 3D (X3D) encodings.* ISO, Geneva, Switzerland, 2005.

[10] ISO 19777:2006. *Information technology - Computer graphics and image processing - Extensible 3D (X3D) language bindings.* ISO, Geneva, Switzerland, 2006.

[11] M. Leyton. *A Generative Theory of Shape*, volume 2145 of *Lecture Notes in Computer Science.* Springer Berlin/Heidelberg, Nov. 2001.

[12] A. Paoluzzi, V. Pascucci, and M. Vicentino. Geometric Programming: A Programming Approach to Geometric Design. *ACM Transactions on Graphics*, 14(3), 1995.

[13] A. Paoluzzi and C. Sansoni. Programming Language for Solid Variational Geometry. *Computer-Aided Design*, 24(7), 1992.

[14] A. Payne and R. Issa. *The Grasshopper Primer.* LIFT Architects, 2009.

[15] Persistence of Vision Raytracer Pty. Ltd. *POV-Ray Documentation: The early history of POV-Ray*, 2001.

[16] M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten, R. Kelsey, W. Clinger, J. Rees, R. B. Findler, and J. Matthews. Revised^6 Report on the Algorithmic Language Scheme. *See `http://www.r6rs.org/`.*, 2007.