Modern Programming for Generative Design
Extended Abstract

José António Branquinho de Oliveira Lopes

Dissertation for the degree of Master of Science in
Information Systems and Computer Engineering

Jury

President:	Prof. Dr. Mário Rui Fonseca dos Santos Gomes
Adviser:	Prof. Dr. António Paulo Teles de Menezes Correia Leitão
Member:	Prof. Dr. João António Madeiras Pereira

June 2012

1 Modern Programming for Generative Design

1.1 Generative Design

Throughout architecture history, coding has been a means of expressing rules, constraints, and systems, that are relevant for the architectural design process. Among other meanings (e.g., statutory, representation, and production codes), coding in architectural design can be understood as the representation of algorithmic processes that express architectural concepts or solve architectural problems. Even before the invention of digital computers, algorithms were applied and incorporated in the design process, as documented in the De re aedificatoria (Krüger, Duarte, & Coutinho, 2011).

Computers popularized and extended the notion of coding in architecture (Rocker, 2006) by simplifying the implementation and computation of algorithmic processes. As a result, increasingly more architects and designers are aware of digital applications and programming techniques, and are adopting these methods as generative tools for the derivation of form (Kolarevic, 2000). Even though the improvements of direct manipulation in CAD applications led many to believe that programming was unnecessary, the work of Maeda shows the exact opposite (Maeda, 1996).

Computational design methods allow automation of the design process and extension of the standard features of CAD applications (Killian, 2006), thus transcending their limitations (Terzidis, 2003). As a result, CAD software shifts from a representation tool to a medium for algorithmic computation, from which architecture can emerge.

The application of computational methods to design architectural structures or objects is called Generative Design (Krause, 2003). In other words, in Generative Design (GD), designers write programs that when executed produce geometric models.

However, in some cases, GD programs have little correlation between inputs and outputs. As a result, it is difficult, and in some cases impossible, to predict which inputs produce the desired outputs. This can be overcome by introducing constraints and parameters in GD programs, resulting in a constrained form of GD called parametric design (Shea, Aish, & Gourtovaia, 2005). Having stronger correlation between program inputs and outputs means that designers can search for a particular output simply by adjusting the input parameters, without modifying the program.

To apply computational methods, one must first translate the thought process into a computer program by means of a Programming Language. A Programming Language (PL) is composed of *(1)* prim-

itives, *(2)* combination mechanisms, and *(3)* abstraction mechanisms. The most used PLs, such as, C, Cpp, Java, and Cs, are general purpose languages: they provide few predefined abstractions that are not specific to any particular domain. On the other hand, domain-specific languages (Hudak, 1998; Deursen, Klint, & Visser, 2000) provide several primitives and abstractions tailored to a given domain, which together with adequate combination mechanisms can dramatically simplify the programming effort.

Unfortunately, several of the most used languages for GD (GD languages), such as, AutoLISP, RhinoScript, and GDL, provide little domain-specific features and make it difficult to define them. As a result, these PLs are difficult to use for GD. PLs that provide domain-specific features can more closely match the human thinking process and, therefore, are easier to use. Nevertheless, there are other contributing factors, such as, the learning curve and the required amount of background knowledge, that affect the success of a given PL within the GD community. And, ultimately, it is important to remember that designers do not have the same programming skills as software engineers, therefore, GD languages must be simple to learn and use.

Programming environments are equally important because they provide the tools necessary for developing programs, namely, editors, compilers, debuggers, and interpreters. Without the proper support from the programming environment, a good PL is still difficult to use.

1.2 Related Work

In order to design a successful programming environment for GD, it is first necessary to understand the features and limitations of current systems. To this end, a study was devised to analyze the most used PLs for GD, including Textual PLs, namely, AutoLISP, RhinoScript, GDL, MAXScript, PLaSM, Processing, Python, SDL, TikZ, and VisualScheme, and Visual PLs, namely, Grasshopper, GenerativeComponents, CGA, and Hypergraph.

The study showed that the most popular TPLs for GD, such as, AutoLISP, RhinoScript, and GDL, are old, obsolete, provide little domain-specific features, and make it difficult to define them. As a result, they are also inadequate choices for GD. On the other hand, VPLs for GD, such as, Grasshopper and GenerativeComponents, enforce a very restricted programming paradigm and programs scale poorly with size and complexity, making them suitable mainly for small throwaway prototypes.

Table 1.1 summarizes the distinguishing features of the most used TPLs for GD. This table shows that most TPLs are imperative, statement based, lexically scoped, dynamically typed, and function based with higher-order functions. Iteration is performed mostly with loops and array indexes.

Table 1.3 and Table 1.4 summarize the geometric features supported by the most used CAD applications. Legend for these tables is in Table 1.2.

Language	AutoLISP	RhinoScript	GDL	MAXScript	SDL	PLaSM	Processing	Python	TikZ	VisualScheme
Paradigm										
Functional	✓					✓		✓		✓
Imperative	✓	✓	✓	✓	✓		✓	✓	✓	✓
Object based		✓								
Object oriented				✓			✓	✓		✓
Declarative					✓					
Syntax										
Expression	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Statement	✗	✓	✓	✓	✓	✗	✓	✓	✓	✗
Scope										
Lexical		✓	✓	✓		✓	✓	✓	✓	✓
Dynamic	✓				✓					✓
Type checking										
Static							✓			
Dynamic	✓	✓	✓	✓	✓	✓		✓	✓	✓
Eval	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓
Data structures										
Lists	✓	✗	✗	✗	✗	✓	✓	✓	✗	✓
Arrays	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓
Tuples	✗	✗	✗	✗	✗	✓	✗	✓	✗	✓
Control structures										
Case	✗	✓	✗	✓	✓	✗	✓	✗	✓	✓
Repeat	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
For	✗	✓	✓	✓	✓	✗	✓	✓	✓	✓
While	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗
Do while	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗
For-each	✓	✓	✗	✗	✗	✗	✗	✓	✓	✓
Return	✗	✓	✓	✓	✗	✗	✓	✓	✗	✗
Break	✗	✓	✗	✓	✗	✗	✓	✓	✗	✗
Continue	✗	✗	✗	✓	✗	✗	✓	✓	✗	✗
Iterators										
Internal	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓
External	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
Subroutine										
Macro	✗	✗	✗	✗	✓	✗	✗	✗	✓	✓
Function	✓	✓	✗	✓	✓	✓	✗	✓	✗	✓
Procedure	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗
Higher-order	✓	✗	✗	✓	✓	✓	✗	✓	✗	✓

Table 1.1: Distinguishing features of GD languages

Symbol	Description
blank	No match
✓	Match
≈	Partial match
∼	Possible match
≈	Partial match

Table 1.2: Survey legend

	AutoCAD	Blender [1]	3ds Max	Maya	Rhinoceros3D	TikZ	PLaSM	GDL
2D shapes								
Arc	✓			✓	✓	✓		✓
Elliptical arc	✓		✓			✓		
Circle	✓				✓	✓		✓
Donut	✓							
Ellipse	✓				✓	✓		
Helix 2D [6]	✓			✓				
Line segment	✓			∼	✓	✓		✓
NGon	✓	✓		∼	✓			✓
NURBS	✓	✓	✓	✓	✓			
Rectangle	✓	✓		✓	✓	✓		✓
Spline			✓	✓	∼			✓
Star								
Text	✓	✓		✓	✓			✓
3D shapes								
Box	✓	✓	✓	✓	✓			✓
Cone	✓		✓	✓	✓			✓
Cut cone	✓							✓
Cylinder	✓	✓	✓	✓	✓		✓	✓
Elliptical cone	✓							✓
Gengon [23]			✓					
Helix 3D			✓					
Mesh	✓	✓	✓	✓	✓			✓
Paraboloid [37]		≈	≈		✓			✓
Pipe [5]		✓		✓				
Platonic solids					≈			
Pyramid	✓		✓	✓	✓			
Sphere	✓			✓	✓			✓
Spindle [16]			✓					
Superellipsoid								
Torus	✓	✓	✓	✓	✓			
T. icosahedron				✓	≈			
Wedge	✓			✓				✓

Table 1.3: Survey of shapes

	AutoCAD	Blender [1]	3ds Max	Maya	Rhinoceros3D	TikZ	PLaSM	GDL
Operations								
Bend	✓	✓	✓	✓	✓			
Bevel		✓	✓	✓				
Cross section	✓	≈	✓					
Extrusion	✓	✓	✓	✓	✓		✓	✓
Guided loft	✓			✓	✓			
Intersection	✓	✓	✓	✓	✓		✓	✓
Lattice [19]			✓					
Loft	✓			✓	✓			
Mirror	✓	✓	✓		✓			
Move	✓	✓	✓	✓	✓	✓	✓	✓
Offset	✓		✓	✓	✓			
Path loft	✓		✓					
Revolution	✓	✓		✓	✓			✓
Rotation	✓	✓	✓	✓	✓	✓	✓	✓
Scale	✓	✓	✓	✓	✓	✓	✓	✓
Skew			✓		✓			
Slice	✓		✓					✓
Subtraction	✓	✓	✓	✓	✓		✓	✓
Sweep	✓	✓	✓	✓	✓			
Thicken	✓		✓	∼	≈			
Union	✓	✓	✓	✓	✓		✓	✓
Other Objects								
Particle systems		✓	✓	✓				
Doors			✓		≈			✓
Windows			✓		≈			✓
Stairs			✓		≈			
Railing			✓		≈			
Wall			✓		≈			✓
Foliage			✓		≈			

Table 1.4: Survey of operations and other GD objects

1.3 Design Principles

To overcome the problems of current GD systems, this thesis proposes to design and implement a programming environment for GD that is pedagogic, with domain-specific features for GD, capable of interacting with the most used CAD applications, and capable of supporting multiple PLs. This programming environment should be simple to use in the sense that advanced programming concepts, such as memory management, should be handled automatically by the environment so that designers can focus on the design task and do not become distracted with implementation details. Moreover, a successful programming environment for GD must meet a number of proposed design principles, including *(1)* portability, *(2)* parametric elements, *(3)* functional operations, *(4)* dimension independent operations, *(5)* algebra of sets, *(6)* algebraic equivalences, *(7)* traceability, and *(8)* immediate feedback. The rest of this section details each design principle.

Programs written in the PLs provided by CAD applications are not portable because they execute only in the family of CAD applications for which they were originally written. As a result, users are locked-in to one family of CAD applications and they cannot reuse programs written for other families. Reusability is also important because it allows software to adapt to new environments. Finally, unless there is a portable programming environment tailored for GD that enables collaborative development and allows the large number of existing GD programs to be reused, the GD community might never grow.

GD languages support few parametric elements but they work mainly with geometric shapes. The difference is that geometric shapes have a visual representation, whereas parametric elements can be described mathematically by functions. Because CAD applications cannot handle functions directly, designers must first convert the parametric elements used in their programs to geometric shapes. In some cases, this requires implementing sophisticated interpolation algorithms, thus distracting them from the essence of the design task and making the problem unnecessarily complex.

Another problem of current CAD applications is that, in some cases, objects are consumed by geometric operations to create other objects. Even though this behavior might make sense in the interaction between a human and a CAD application, it makes GD algorithms produce wrong results on even abort with an error. To overcome this problem, all geometric operations must be functional, in the sense that they should not consume their arguments.

Moreover, in most GD languages, there is no uniform treatment for one-, two-, and three-dimensional shapes. As a result, user-defined operators must make a large case-based analysis or comprehend several different definitions, thus aggravating the non-uniform treatment of shapes. In either case, the operator implementation might be incomplete. Similar to this problem is the insufficient support for shapes that can parametrically morph between different space dimensions. For example, a cylinder can morph into a circle, line, or a point, if the radius or height are either or both zero. For mathematical (and

computational) correctness, it is important that all operations accept and properly handle morph cases.

Another case in which mathematical correctness is important is in the calculation of shapes. Shapes are mathematically described as sets of points in space. Therefore, Boolean operations, namely, intersection, subtraction, and union, are merely operations on sets. These operations have well defined identity and absorbing elements, such as, the universal and empty sets. Unfortunately, CAD applications do not implement them. As a result, even the most basic operations are not properly defined.

While sets allow understanding shapes from their mathematical point of view, algebraic equivalences are important to understand operations. These equivalences give the designer freedom of choice and are fundamental when writing a program because *(1)* the actual performed combination of Boolean operations can be difficult to predict; *(2)* a program that only runs in the CAD application that supports one particular combination is not portable; and *(3)* adapting a program to use only the combinations of Boolean operations supported by a CAD application might require extensive changes.

As mentioned before, in GD designers interact with a program that creates a model. Because they do not interact directly with the model, it becomes difficult to understand the relationship between the parts of the program and those of the model. Traceability overcomes this problem by allowing designers to *(1)* point to a program element and immediately identify the corresponding elements of the model; and *(2)* point to an element of the model and immediately identify the corresponding elements of the program.

Traceability allows a designer to understand the correlation between his GD program and the generated model. However, it does not allow the designer to easily understand the correlation between the program inputs and that model (output). To this end, the program must be re-executed when the input changes and the model re-visualized, a slow-pace process that will tire even the most patient designer. Immediate feedback attempts to solve this problem, by allowing the designer to continuously adjust the program inputs and immediately visualize the generated model until it reflects his intentions.

1.4 Rosetta

Because currently there are no GD systems that implement these principles with the proper support for GD, a new programming environment, called Rosetta, was created. This section explains the features of Rosetta.

Rosetta overcomes the portability problem by providing *(1)* multiple PLs as frontends, from which users can choose to write their GD programs; and *(2)* multiple CAD applications as backends, which are used to display the geometric models. With Rosetta, users can explore different frontends and backends in order to find a combination that is most suitable for the design task. Moreover, users have access to different PLs which can be used interchangeably to write portable GD programs. Furthermore, a single program creates identical geometry in different CAD applications. This approach promotes the

development of programs that are portable across the most used CAD applications, thus facilitating the dissemination of those programs and of the underlying ideas. Finally, providing multiple PLs not only overcomes the portability problem but also creates an easy migration path for users of other PLs, such as, AutoLISP and JavaScript, who can find these languages available in Rosetta.

Currently, Rosetta implements three backends, namely, *(1)* AutoCAD and Rhinoceros3D are two of the most used CAD applications and they provide enough functionality to design a portable platform; and *(2)* OpenGL does not provide as much functionality as the previous backends, but rendering is considerably faster; Moreover, Rosetta overcomes several limitations of these backends. For example, Rhinoceros3D cannot model hollow shapes or calculate Boolean operations of non-intersecting shapes. But Rosetta explores algebraic rules for circumvent these limitations.

Currently, Rosetta implements three frontends, namely, AutoLISP, JavaScript, and RosettaRacket. Racket, the PL of DrRacket, provides several tools that simplify the design and implementation of new PLs. For example, macros simplify the implementation of compilers because they allow the definition of syntactic forms that expand to Racket code. As a result, different PLs can interoperate because they are part of the same ecosystem. Similarly to the backends, Rosetta overcomes the limitations of the frontends. For example, the AutoLISP frontend has a syntax checker that is capable of detecting undeclared names at compiled time, even though the original AutoLISP cannot.

Rosetta implements most geometric shapes and operations provided by the most used CAD applications. However, Rosetta overcomes the limitations of these CAD applications to provide shapes and operations with mathematical and geometric correctness. Moreover, it implements also the empty and universal sets as special geometric shapes. Boolean operation automatically recognize and handle these shapes, implementing the identity and absorbing elements of the intersection, subtraction, and union, operations.

Operations are also functional, meaning that designers do not have to worry whether or not shapes are consumed and shapes can be shared by all parts of a program and freely used as arguments to operations. In order to provide the correct mathematical semantics, Rosetta programs do not compute the geometric shapes and transformations described in the program. Instead, these elements are composed in a scene graph. When evaluated, the scene graph produces shapes and applies the transformations in the selected CAD tool.

Moreover, operations accept also parametric elements, meaning functions that describe shapes coupled with intervals that specify their domain. If necessary, these elements are automatically interpolated using an adaptive sampling strategy (Chandler, 1990) that minimizes the interpolation error.

Because several operations in Rosetta are dimension independent, user-defined operators are generic, meaning that they can be applied to shapes of different dimensions. Naturally, the internal implementation of each predefined operation might require several specific CAD procedures. But this is hidden

from the designer, who only has to know one generic operator.

Finally, Rosetta provides sliders which can be connected to program inputs. When designers change a slider, Rosetta automatically recomputes the model. This re-computation process operates in real time, for simple GD programs, being a form of immediate feedback. However, complex programs can take significant time to recompute and the interactive use of widgets can become annoying, a problem that affects both Grasshopper and Rosetta. Unfortunately, immediate feedback can never scale to arbitrarily large programs because each operation that is added to a program increases the total amount of time needed to compute it. Moreover, some operations have an intrinsic complexity, such as, linear, quadratic, or exponential, that cannot be avoided.

1.5 Evaluation

Rosetta and the implemented design principles were evaluated in three parts.

The first part evaluated the multitude of programming paradigms supported by Rosetta, showing several examples of GD programs that explore the advantages of the functional and imperative paradigms, as well as many programming techniques, such as, higher-order and anonymous, functions, monads, and non-deterministic programming.

The second part presented a discussion the portability and the advantages of Rosetta over other GD systems. Moreover, it showed how to introduce a new backend for TikZ and a new frontend for a VPL called RosettaFlow, which was inspired in Grasshopper and GenerativeComponents.

Finally, the third part completed this evaluation with practical experiments, namely, a comparison of TPLs and VPLs, as presented in Leitão et al. (Leitão, Santos, & Lopes, 2012), and conversion and analysis of AutoLISP programs.

1.6 Conclusion

Coding has always been present in the architectural design process. Computers popularized and extended the notion of coding in architecture, suggesting that coding could be used to represent algorithmic processes that express architectural concepts or solve architectural problems. As a result, increasingly more architects and designers are aware of digital applications and programming techniques, and are adopting these methods in a modern architectural form called Generative Design (GD). In GD, designers write programs that when executed produce geometric models. These programs are usually controlled by a large set of parameters, such that designers can experiment with different variations of a geometric model simply by changing those parameters and without modifying the program.

The choice of a good Programming Language (PL) reduces dramatically the effort in writing GD programs. However, the most used PLs, such as, C, Cpp, Java, and Cs, are inadequate because they

general purpose languages, providing few predefined abstractions for GD.

Moreover, the survey of the most used GD systems showed that the most popular TPLs for GD, such as, AutoLISP, RhinoScript, and GDL, are old, obsolete, provide little domain-specific features, and make it difficult to define them. As a result, they are also inadequate choices for GD. On the other hand, VPLs for GD such as, Grasshopper and GenerativeComponents, enforce a very restricted programming paradigm and programs scale poorly with size and complexity, making them suitable mainly for small throwaway prototypes. And CAD applications are based on interactive technology and impose their own programming environments and languages, making users locked-in to specific CAD families and introducing portability problems. This scenario clearly shows that even though there is a great need for a modern programming environment for GD, the current and most used tools are unsuitable choices.

In order to overcome this problem, this thesis argued that a modern programming environment should *(1)* be pedagogic; *(2)* provide domain-specific features; *(3)* provide multiple PLs; and *(4)* provide of multiple CAD applications. Moreover, this thesis proposed a set of design principles, which include, *(1)* portability, *(2)* parametric elements, *(3)* functional operations, *(4)* dimension independent operations, *(5)* algebra of sets, *(6)* algebraic equivalences, *(7)* traceability, and *(8)* immediate feedback. Finally, this thesis argued that a successful programming environment for GD must implement these principles. Because currently there are no GD systems that implement these principles with the proper support for GD, a new programming environment, called Rosetta, was created.

Rosetta is modern programming environment designed to overcome the problems of current GD systems. To this end, Rosetta provides *(1)* multiple frontends PLs, which can be used interchangeably to write portable GD programs and to reuse existing software; *(2)* multiple CAD applications, which generate identical geometry, thus freeing designers from vendor lock-in, allowing the Rosetta programs to be used as an alternative to the file formats of these applications, and allowing students to learn more than one CAD package; *(3)* multiple programming paradigms, which allow designers to more closely match the computational method with the design task, thus simplifying the implementation effort, as shown in the examples of coordinate generators and tessellators design pattern and automated rendering; *(4)* modern linguistic features and programming techniques, such as, higher-order and anonymous functions (balcony example), monads (Lego panda example), and non-deterministic programming (pipes inside a sphere example); *(5)* visual input widgets that make it simple to adjust GD programs interactively and in real-time. Moreover, Rosetta was designed to be a pedagogic programming environment, providing a choice of simple and advanced programming concepts, making it a suitable platform for designers to study programming in a controlled environment, with the additional advantage that geometric concepts are independent of the chosen PL.

Finally, Rosetta overcomes several limitations of current CAD technology, most of which arise from the fact that CAD applications were designed for the interaction between a human and a computer. These limitations are overcome by implementing well defined mathematical rules in the programming

environment. Moreover, Rosetta programs have a strong correlation between the program and the model, providing traceability mechanisms that establish a relationship between the elements of the program and those of the model and allow the designer to navigate in both directions, and immediate feedback via the OpenGL backend and visual widgets.

To evaluate these design principles, several GD programs were written that explore the multitude of programming paradigms and techniques, showing the advantages of Rosetta over GD systems that enforce a particular programming approach. Moreover, a discussion was presented that focused on the advantages of a portable programming environment that supported multiple frontends and backends. To prove this point, the TkZ backend and the RosettaFlow frontend were implemented. Finally, this evaluation was complete with practical experiments, namely, a comparison of TPLs and VPLs, and conversion and analysis of AutoLISP programs, which include large and complex projects, namely, the TurningTorso.

In summary, Rosetta is a modern programing environment for GD, designed to implement and evaluate several design principles that this thesis argues as fundamental for successful GD. In seek of this goal, current CAD technology has always been a difficult obstacle but Rosetta is the proof that these problems have been overcome in a number of ways. In the end, Rosetta is merely a computational manifestation of the principles and ideas argued in this thesis, and if the GD community and GD system developers follow these principles, it is possible for GD to evolve in a successful and effective fashion with good support from Computer Science and software.

Bibliography

Chandler, R. (1990). A recursive technique for rendering parametric curves. *Computers & Graphics*, *14*(3/4), 477-479.

Deursen, A. van, Klint, P., & Visser, J. (2000, June). Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, *35*(6), 26-36.

Hudak, P. (1998). Domain specific languages. In *Handbook of programming languages, Volume III: Little languages and tools* (p. 39-60). Indianapolis, IN, USA: MacMillan.

Killian, A. (2006). Design innovation through constraint modeling. *International Journal of Architectural Computing*, *4*(1), 87-105.

Kolarevic, B. (2000, October). Digital architectures. In M. Clayton & G. de Velasco (Eds.), *Eternity, infinity and virtuality in architecture [Proceedings of the 22nd annual conference of the Association for Computer Aided Design in Architecture (ACADIA)]* (p. 251-256). DE, USA: Association for Computer Aided Design in Architecture.

Krause, J. (2003, December). Reflections: The creative process of generative design in architecture. In C. Soddu (Ed.), *GA2003 Proceedings of the 6th international conference on generative art*.

Krüger, M., Duarte, J. P., & Coutinho, F. (2011). Decoding De re aedificatoria: Using grammars to trace Alberti's influence on Portuguese classical architecture. *Nexus Network Journal*, *13*, 171-182.

Leitão, A., Santos, L., & Lopes, J. (2012, March). Programming languages for generative design: A comparative study. *International Journal of Architectural Computing*, *10*(1), 139-162.

Maeda, J. (1996). *Design by numbers*. Cambridge, MA, USA: MIT Press.

Rocker, I. (2006). When code matters. *Architectural Design*, *76*(4), 16-25.

Shea, K., Aish, R., & Gourtovaia, M. (2005, March). Towards integrated performance-driven generative design tools. *Automation in Construction*, *14*(2), 253-264.

Terzidis, K. (2003). *Expressive form: A conceptual approach to computational design*. London, UK and New York, NY, USA: Spon Press.