



Modern Programming for Generative Design

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

Agradecimentos

Agradeço... I thank...

Ao Instituto Superior Técnico que permitiu ao rapaz que sempre gostou de engenharia informática e programação realizar o sonho de se tornar engenheiro informático.

À Fundação para a Ciência e a Tecnologia (FCT) pelo apoio financeiro na participação na conferência *ACADIA 11* em Banff, Canadá.

Aos meus irmãos António, Ana e Miguel que sempre me apoiaram e sempre se disponibilizaram para me ajudar no meu trabalho.

Aos meus pais pelo apoio e paciência. Ao meu pai digo, em tom de brincadeira, que embora tenha acabado a tese de mestrado em segundo lugar, um desafio o espera no doutoramento. À minha mãe que a sua experiência e sabedoria sublinhe para todo o sempre esta dedicatória.

Ao Prof. Dr. António Leitão pelo seu imenso apoio nos momentos mais difíceis e pelo seu eficaz, ainda que difícil, equilíbrio entre critismo e elogio. Mas acima de tudo, pela sua honestidade intelectual e carácter ético, e pela sua facilidade em olhar para além da linha que separa um professor de um amigo.

Aos meus amigos Bernardo, David, Jaime, João e Miguel, pelo seu apoio nos momentos mais difíceis do curso, e pela sua infindável amizade e confiança.

Gosia, for her love and support, the Poland, Hong Kong, Macau, London, and Lisbon, adventures and scary times, the happiness of being close and the sadness of wishing to be closer, but above all for the dream and promise of being together.

Lisboa, July 4, 2012

José Lopes

“We think in language.
And so the quality of our
thoughts and ideas can only
be as good as the quality of
our language.”

George Carlin

Resumo

Cada vez mais arquitectos transitam dos processos tradicionais e das codificações arquitecturais clássicas para uma nova área chamada Desenho Generativo. Desenho Generativo (DG) é a aplicação de métodos computacionais na geração de objectos arquitecturais. Nesta área, designers escrevem programas que quando executados produzem modelos geométricos. Este movimento é claramente visível no mundo académico, com a adopção de cadeiras de programação no currículo de arquitectura, e no mundo da indústria, com estúdios de arquitectura a substituir processos tradicionais por aplicações de computador. Consequentemente, arquitectos e designers precisam desesperadamente de um sistema moderno para DG. Infelizmente, a maior parte dos sistemas actuais não é capaz de responder a esta necessidade porque (1) ou estão desactualizados ou obsoletos; (2) ou obrigam à adopção de métodos de programação inadequados; (3) ou não são pedagógicos. Para ultrapassar este problema, esta tese propõe um conjunto de princípios de desenho que um sistema para DG deve implementar para ter sucesso, nomeadamente, (1) portabilidade de programas; (2) rigor matemático; e (3) forte correlação entre programas e modelos. Porque actualmente não existem sistemas para DG que implementem estes princípios com o devido suporte, um novo ambiente de programação, chamado Rosetta, é proposto. O Rosetta suporta múltiplas Linguagens de Programação e múltiplas aplicações de Desenho Assistido por Computador (DAC), fornecendo diferentes paradigmas e técnicas de programação, e ricas funcionalidades linguísticas. Esta tese utiliza o Rosetta para implementar e validar os princípios de desenho propostos, mostrando as vantagens deste ambiente de programação moderno comparativamente aos sistemas mais utilizados para DG.

Abstract

Increasingly more architects are moving from the traditional architectural processes and the classic forms of architectural coding to a modern area called Generative Design. Generative Design (GD) is the application of computational methods to design architectural structures or objects. In this area, designers write programs that when executed produce geometric models. This movement is clearly visible both in the academia, with current architecture curricula adopting programming courses, and in the industry, with architecture studios replacing traditional processes with computer applications. As a result, architects and designers are in desperate need of a modern system for GD. Unfortunately, the most used systems are not capable of responding to this need because either (1) they are old or obsolete, or (2) they enforce inadequate programming methods, or (3) they are not pedagogic. In order to overcome this problem, this thesis proposes a set of design principles that a GD system must implement in order to be successful, namely, (1) portability of programs, (2) mathematical correctness, and (3) strong correlation between programs and models. Because currently there are no GD systems that implement these principles with the necessary support for GD, a new programming environment, called Rosetta, is proposed. Rosetta supports multiple Programming Languages and multiple CAD applications, providing different programming paradigms and techniques, and rich linguistic features. This thesis uses Rosetta to implement and validate the proposed design principles, clearly showing the advantages of this modern programming environment over the most used systems for GD.

Palavras Chave Keywords

Palavras Chave

Linguagens de Programação

Desenho e Implementação de Linguagens de Programação

Computação Gráfica

Desenho Generativo

Desenho Assistido por Computador

Arquitectura

Keywords

Programming Languages

Programming Language Design and Implementation

Computer Graphics

Generative Design

Computer Aided Design

Architecture

Contribuições

No âmbito desta tese de mestrado, foram realizadas cinco publicações científicas, nomeadamente, (1) o artigo *Programming Languages For Generative Design: A Comparative Study* publicado na revista *International Journal of Architectural Computing* (Leitão, Santos, & Lopes, 2012b) para o qual o autor desta tese contribuiu com as partes relativas ao Rosetta e ao CityEngine CGA; (2) o artigo *Portable Generative Design for CAD Applications* publicado na conferência ACADIA 11: *Integration through Computation* (Lopes & Leitão, 2011b); (3) o artigo *Essential Language Features for Generative Design* publicado no III Simpósio de Informática (INForum 2011) (Lopes & Leitão, 2011a); e, finalmente, (4) o artigo *Collaborative Digital Design* aceite na conferência eCAADe 2012: *Digital Physicality — Physical Digitality* (Leitão, Santos, & Lopes, 2012a).

Contributions

During the development of this master thesis, five scientific articles were written, namely, (1) the article *Programming Languages For Generative Design: A Comparative Study* published in the journal *International Journal of Architectural Computing* ([Leitão et al., 2012b](#)) for which the author of this thesis contributed the *Rosetta* and *CityEngine CGA* sections; (2) the article *Portable Generative Design for CAD Applications* published in the conference *ACADIA 11: Integration through Computation* ([Lopes & Leitão, 2011b](#)); (3) the article *Essential Language Features for Generative Design* published in the *III Simpósio de Informática (INFOrum 2011)* ([Lopes & Leitão, 2011a](#)); and, finally, (4) the article *Collaborative Digital Design* accepted in the conference *eCAADe 2012: Digital Physicality — Physical Digitality* ([Leitão et al., 2012a](#)).

Contents

1 Generative Design	1
1.1 Coding in Architecture	1
1.2 Generative & Parametric Design	1
1.3 Programming Languages & Environments	2
2 Related Work	5
2.1 Programming Languages & Generative Design	5
2.2 Textual Programming Languages	5
2.2.1 AutoLISP	5
2.2.2 RhinoScript	6
2.2.3 GDL	7
2.2.4 MAXScript	7
2.2.5 PLaSM	8
2.2.6 Processing	9
2.2.7 Python	9
2.2.8 SDL	10
2.2.9 TikZ	10
2.2.10 VisualScheme	11
2.2.11 Textual Language Analysis	12
2.3 Visual Programming Languages	12
2.3.1 Grasshopper	14
2.3.2 GenerativeComponents	16
2.3.3 CGA	16
2.3.4 Hypergraph	16
2.3.5 GD System Analysis	17

3 Modern Programming Environment	23
3.1 Design Principles	23
3.2 Portability	24
3.3 Parametric Elements	24
3.4 Functional Operations	25
3.5 Dimension Independent Operations	26
3.6 Algebra of Sets	26
3.7 Algebraic Equivalences	28
3.8 Traceability	28
3.9 Immediate Feedback	29
4 Rosetta	31
4.1 Design Requirements	31
4.2 Software Architecture	31
4.3 Editor	33
4.4 Backends	34
4.4.1 AutoCAD and Rhinoceros3D	34
4.4.2 OpenGL	36
4.5 Frontends	37
4.5.1 RosettaRacket	38
4.5.2 AutoLISP and JavaScript	38
4.6 Shapes and Operations	38
4.7 Traceability	39
4.8 Visual Widgets	41
5 Programming Paradigms & Techniques	43
5.1 Multi-Paradigm Programming Language	43
5.1.1 Example 1: Tessellators and Coordinate Generators	43
5.1.2 Example 2: Automated Rendering	46
5.2 Programming Techniques	48
5.2.1 Example 1: Combination of Programming Techniques	48

5.2.2	Example 2: Higher-Order and Anonymous Functions	49
5.2.3	Example 3: Monads	52
5.2.4	Example 4: Non-Deterministic Programming	54
6	Multiple Frontends and Backends	57
6.1	Portability	57
6.2	TikZ	58
6.3	RosettaFlow	58
7	Practical Experiments	61
7.1	TPLs <i>vs.</i> VPLs	61
7.2	Program Conversion and Analysis	64
8	Conclusions	69
8.1	Conclusions	69
8.2	Current and Future Work	71
A	RosettaLang	79

List of Figures

2.1	<i>Grasshopper</i> program for computing the coordinates of a conical spiral	12
2.2	<i>Grasshopper</i> program fragment with tangled connectors and wireless endpoints	15
2.3	<i>CGA</i> editor showing textual and visual representations of a shape grammar	17
3.1	Symmetric difference of cylinders	25
3.2	Morphing a cylinder into a disk, a line, and a point	27
3.3	Hollow cylinder subtracted by a sphere	28
4.1	<i>Rosetta</i> with a <i>JavaScript</i> program and corresponding geometry in <i>AutoCAD</i>	32
4.2	<i>DrRacket</i> window	33
4.3	<i>RosettaRacket</i> program and corresponding geometry in <i>AutoCAD</i>	34
4.4	<i>RosettaRacket</i> program and corresponding geometry in <i>Rhinoceros3D</i>	35
4.5	<i>RosettaRacket</i> program and corresponding geometry in <i>OpenGL</i>	35
4.6	Hollow sphere pierced by several cones	36
4.7	Orthogonal cones	37
4.8	Möbius truss	37
4.9	Scriptecture	37
4.10	Symmetric difference: definition (on the left) and implementation (on the right)	39
4.11	Dimension independent symmetric difference	40
4.12	Relating program expressions to the generated shapes. The highlighted cylinders (on the left, in yellow) are generated by the highlighted program text (on the right, in blue)	40
4.13	Relating shapes to the program expressions that generated them. The highlighted cylinders (on the left, in yellow) are generated by the highlighted program flow (on the right, using red arrows)	41
4.14	Using sliders and the <i>OpenGL</i> backend to interactively generate different models	42
5.1	Space frames and the Wimbledon Stadium roof	44

5.2	Arc surface space frame	45
5.3	Möbius space frame	46
5.4	Sphere made of cones	49
5.5	Sphere pierced by a set of cones	49
5.6	Balcony with a saw teeth variation	51
5.7	Balcony with an oscillating variation	52
5.8	Balcony with a sinusoidal variation	52
5.9	Giant panda <i>Lego</i> example	53
5.10	Tubes in sphere	54
5.11	Blocks in sphere	55
5.12	Tubes in cube	55
5.13	Tubes in cylinder	55
6.1	<i>TikZ</i> example	58
6.2	Changing sliders in <i>RosettaFlow</i> causes the geometric model to update in real-time	60
7.1	First task of the experiment	61
7.2	Second task of the experiment	61
7.3	<i>Grasshopper</i> solutions for first (on the left) and second (on the right) tasks, with changes highlighted in orange	62
7.4	<i>VisualScheme</i> solution for the first task of the experiment	63
7.5	<i>VisualScheme</i> modifications for the second task of the experiment	63
7.6	<i>Turning Torso</i> with different parametrizations	65
7.7	Catalog of <i>Rosetta</i> programs	66
7.8	Catalog of <i>Rosetta</i> programs (continuation)	67
A	RosettaLang	79
A.1	<i>RosettaLang</i> catalog	79
A.2	<i>RosettaLang</i> documentation	80

List of Tables

2.1	Distinguishing features of GD languages	13
2.2	Survey legend	17
2.3	Survey of shapes	18
2.4	Survey of operations and other GD objects	19
4.1	Time (in milliseconds) needed to update the generated design	37

Nomenclature

- API Application Programming Interface. The Application Programming Interface is the set of variables, functions, methods, and types, exported by a system which clients of that system can use to interface with it.
- CAD Computer Aided Design. Computer Aided Design is the use of software to create architectural models.
- GD Generative Design. Generative Design is a modern form of Computer Aided Design in which architectural models are generated using algorithms.
- GUI Graphical User Interface. A Graphical User Interface is the set of windows and dialogs of an application that allow a user to interact with it.
- JIT Just In Time. Just In Time compiler is a programming language compiler that optimizes code on demand at runtime.
- PL Programming Language.
- TPL Textual Programming Language. A Textual Programming Language is a programming language in which programs are a linear sequence of characters.
- VPL Visual Programming Language. A Visual Programming Language is a programming language in which programs are usually a bidimensional representation of iconic elements that can be interactively manipulated according to some spatial grammar.

1

Generative Design

1.1 Coding in Architecture

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.

1.2 Generative & Parametric Design

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.

One of the problems of GD is that, in some cases, 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.

1.3 Programming Languages & Environments

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) primitives, (2) combination mechanisms, and (3) abstraction mechanisms. The most used PLs, such as, *C*, *C++*, *Java*, and *C#*, 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*, *Rhino-Script*, 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. Examples of domain-specific features for GD include (1) primitives, such as, points, curves, surfaces, and solids; and (2) abstractions, such as, coordinate, matrix, and coordinate system. PLs that provide domain-specific features can more closely match the human thinking process and, therefore, are easier to use. Examples of such PLs include *Grasshopper* and *VisualScheme*. 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. For example, an important feature of the *AutoLISP* and *Grasshopper* programming environments is the good integration with their host CAD application, namely, *AutoCAD* and *Rhinoceros3D*, respectively. This integration makes it possible to complement the functionality of the programming environment with that of the CAD application. Unfortunately, not all GD languages follow this approach.

This thesis proposes a set of principles, called design principles, that are essential for a modern GD system. Because currently there are no GD systems that implement the proposed principles with the necessary support for GD, a new system, called *Rosetta*, is also proposed. *Rosetta* is a modern programming environment for GD that implements the proposed principles, clearly showing the advantages of this environment over the most used GD systems.

In order to design a successful programming environment for GD, it is first necessary to understand the features and limitations of current systems. Some of them have already been discussed, such as, automation and domain-specific features. The next section presents a survey of the most used GD systems, detailing the most relevant features and problems. The results of this survey were used as a guide for defining the requirements for a modern programming environment. In order to evaluate these

requirements, they were implemented in *Rosetta*, the programming environment proposed in this thesis.

After the survey, *Rosetta* is explained, including its software components, features, and advantages over current GD systems. Finally, *Rosetta* is evaluated resorting to several GD programs, new software components, and two practical experiments.

2 Related Work

2.1 Programming Languages & Generative Design

In order to design a successful programming environment for Generative Design (GD), it is first necessary to understand what kind of GD systems exist, including Computer Aided Design (CAD) applications, 3D modeling applications, GD languages, renderers, and geometric libraries. For example, CAD applications, such as *AutoCAD*, have a very good knowledge of the GD domain in terms of geometric shapes and operations, such as, line, sphere, extrusion, and union. Therefore, studying the features and limitations of these applications and their programming capabilities is a necessary starting point for designing a new programming environment. To this end, a study was devised to analyze both Textual PLs, such as, *RhinoScript* and *PLaSM*, and Visual PLs, such as, *Grasshopper* and *GenerativeComponents*. This section presents the results of this study.

2.2 Textual Programming Languages

In a Textual Programming Language (*TPL*), a program is a one-dimensional representation consisting of a linear sequence of characters. This section presents several *TPLs* for GD. Some of these PLs, such as *AutoLISP* for *AutoCAD*, are integrated in GD systems, while others, such as, *PLaSM*, are standalone PLs.

2.2.1 AutoLISP

Similarly to other CAD applications, *AutoCAD* provides a programming environment, called *VisualLISP*, which gives designers scripting capabilities to extend the functionality of *AutoCAD* and create geometric models procedurally. *VisualLISP* has a text editor, an interpreter, a debugger, and the *AutoLISP* PL, a very old dialect of *LISP*, which is a family of PLs notably known for the fully parenthesized syntax. Despite being one of the most used GD languages, *AutoLISP* is obsolete and has several problems, most of which have a historical reason.

AutoLISP uses dynamic scope, which is a source of many problems, particularly when combined with higher-order functions (Cabeçinhas, 2010). Examples of these problems include the downward and upward *funarg* problems (Moses, 1970).

There is little support for data structures apart from lists. Therefore, design concepts, such as,

coordinates and vectors, are implemented with lists and, as a result, *AutoLISP* programs suffer from limited abstraction and performance problems.

Symbols are also problematic because while statically typed languages, such as, C and *Java*, do not even compile when an undeclared symbol is used, *AutoLISP* not only accepts them but the computation proceeds normally without issuing any dynamic error. As a result, even small typos are accepted, leading to errors that are very difficult to detect.

Other errors that are equally difficult to detect are integer overflow/underflow because arithmetic operators fail silently. This a problem in *AutoLISP* but not in the majority of *LISP* dialects. For example, *Common LISP*, another *LISP* dialect, overcomes this problem with arbitrary-size integers.

AutoLISP communicates with *AutoCAD* via three different mechanisms, namely, the command primitive, entity manipulation, and the *Component Object Model* (COM).

The command primitive is a function that receives a variable number of arguments that are injected in *AutoCAD* as if the user had typed them at the *AutoCAD* prompt. This function has a significant performance penalty and its semantics is too general, making it difficult to use.

Entities are manipulated via `entmake`, `entget`, `entmod`, `entnext`, and `entlast`, which create geometric objects and access their internal structure. It is difficult to use these functions because users must learn and remember a large set of numerical codes or constantly read technical documentation. Moreover, in some cases, it is necessary to combine these functions with the command primitive. A typical scenario consists of creating shapes using command and obtaining references to them using `entlast` and `entnext`, and then reading and modifying their properties using `entget` and `entmod`.

Finally, COM is a programming technology that allows programs to be packaged as software components and easily integrated into applications written in other PLs. With COM, it is possible to write applications external to *AutoCAD* that interact with this application by remote procedure calls.

2.2.2 RhinoScript

Rhinoceros3D is one of the most used CAD applications. It provides *RhinoScript* ([Rutten, 2007](#)), a variant of *VBScript* for GD with domain-specific features. Similarly to *Fortran* and *Pascal*, *RhinoScript* follows the old tradition of having different declaration and call syntax for functions and subroutines. This distinction is confusing and industry-wide languages, such as, C and *Java*, prove it is unnecessary. Moreover, by default, function parameters are passed *by value*. Therefore, arrays are automatically copied when passed to functions. Because this has a severe performance penalty, users must learn the difficult concepts of passing parameters *by value* and *by reference*, and declarations become more verbose.

Arrays are the main data structure and they are used to implement design concepts, such as, coordinates and vectors. Statically-sized arrays are initialized with the upper bound index, instead of

the array size, and dynamically-sized arrays require an extra re-dimension operation. Arrays can be multidimensional or nested, and array operations must be called accordingly.

2.2.3 GDL

In *ArchiCAD*, another popular CAD application, geometric objects, called library parts, are organized in a hierarchy with property inheritance, similar to that of object-oriented programming. Each library part contains a set of parameters and a *GDL* script.

GDL ([Watson, 2009](#)) is a descendant of *BASIC*. Contrary to its siblings, such as, *VisualBasic* and *VisualBasic .NET*, *GDL* did not evolve, providing very little abstraction mechanisms, namely, variables, subroutines, and macros. Subroutines cannot define local variables, receive parameters, or return values, therefore, global variables are used instead. Because *GDL* provides insufficient abstraction mechanisms, and commands are mainly low-level, there is a significant effort spent in writing scripts, which quickly become verbose and difficult to read.

The parameters of library parts can be modified through a properties window or through *hotspots*, which are widgets that appear in the *ArchiCAD* window and can be interactively selected and dragged. Because the parameters are defined outside the *GDL* script, it is possible to experiment with different values without changing the program.

The *GDL* script associated with a library part contains a sequence of commands that describe geometric shapes. Similarly to *OpenGL*, transformations are implemented by a matrix stack. As a result, transformations are accumulated on the stack, such that, when creating a new shape, the position, orientation, and scaling, of that shape are taken from the stack. However, there is inconsistency between 2D and 3D shapes because for 2D shapes the position is specified by a parameter.

2.2.4 MAXScript

MAXScript ([Autodesk, 2006](#)) is an imperative, object-oriented PL that combines the simple syntax of *VisualBasic* and *C* with the powerful semantics of *LISP*, being both easy to use and rich in abstraction mechanisms. Because it was designed to work closely with the scene graph of *3ds Max*, it gives control over modeling, animation, materials, and rendering, and multiple objects/properties are globally accessible using wildcards.

The object-oriented system implements inheritance and polymorphism, and methods include (1) single dispatch, (2) mapped methods (i.e., implicit `foreach`), and (3) parameters *by value* and *by reference*. Parameters are (1) positional: mandatory and imposing a position to formal and actual arguments; or (2) keyword: optional with the formal argument identifier preceding the actual argument.

Additional features include (1) exception handling similar to Java; (2) libraries for 3D vector, matrix, and quaternion, which combine powerful computation with ease of use; (3) array and bit-array literal initializers; and (4) default initialization value for variables, making errors easier to detect. However, code blocks are parenthesis delimited, instead of bracket delimited, resulting in no visual clue to distinguish between code blocks and expressions.

Moreover, *MAXScript* is an interpreted PL, therefore, programs might execute slowly. In this case, users are officially advised to use C++ instead. However, this language is impractical for designers because of its difficult syntax and semantics.

2.2.5 PLaSM

PLaSM ([Paoluzzi & Sansoni, 1992](#)), the Programming LAnguage for Symbols Modeling, is a functional PL that descends from the *Function Level (FL)* PL ([Backus, Williams, Wimmers, Lucas, & Aiken, 1989](#); [Aiken, Williams, & Wimmers, 1993](#)), with differences in syntax, the use of multiple lists of formal arguments and longer lists of actual arguments than the ones declared ([Cabecinhas, 2010](#)).

What makes *PLaSM* different from traditional PLs, such as, *C* and *Java*, is the ability to operate at the function level: functions can be combined to create other functions. For example, compact expressions can be created using sophisticated operators, such as, *apply-in-composition* and *apply-in-sequence*, and extensive mathematical notation. Despite this expressiveness, users with little programming experience will find it difficult to read programs and master such advanced programming concepts. Functions can also be higher-order, overloaded, or partially applied (curried), but free scope nesting is not allowed ([Cabecinhas, 2010](#)).

PLaSM has a small set of primitives but powerful, dimension-independent operators. Together, they form a polyhedral algebra embedded in the language. For example, (1) `CUBOID` creates hyper-parallelepipeds of any dimension, such as, line segment, rectangle, cube, and 4D hyper-parallelepiped; (2) `SIMPLEX` creates shapes from the smallest convex hull of a set of points, such as, point, line segment, triangle, and tetrahedron; (3) `CYLINDER` creates a 3D cylinder of specified radius, height, and number of faces; (4) `MKPOL` creates polyhedral complexes, namely, lines, polygons, and solids, using vertices, cells, and polyhedra, where vertices are points in the same space, cells are convex hulls of vertices, and polyhedra are unions of cells; (5) `JOIN` returns the convex hull of a sequence of polyhedral complexes in the same space; (6) `EMBED` inserts polyhedra complexes in an Euclidean space of higher dimension; (7) `STRUCT` creates hierarchical assemblies by composing polyhedral complexes, thus introducing local coordinate systems such that geometric transformations are propagated along the hierarchy; and (8) `ALIGN`, `TOP`, `LEFT`, etc, are used for relative positioning. Another interesting operator is `MAP`, which applies a function to the vertices of a polyhedron. With this operator, it is possible to create parametric shapes, such as, splines and surfaces.

In most cases, shapes are represented in parametric form, which is exact. However, it is not always possible to use this representation. For example, Boolean operations between curved shapes compute a faceted approximation of the actual result. Nevertheless, this problem is not specific to *PLaSM*: most CAD applications, such as, *AutoCAD* and *Rhinoceros3D*, also suffer from this limitation.

2.2.6 Processing

Processing ([Reas & Fry, 2010](#)) is a PL and an interactive programming environment specialized for the production of images and animations. Initially designed for sketching, it has grown to become a professional tool. The PL is a simplified version of *Java* and is capable of generating applets for *Java*-enabled browsers or standalone applications. The standard library resembles *OpenGL* with a higher-level of abstraction and an event-based programming model for handling, for example, user input.

The fact that *Processing* was conceived for computer graphics, and not GD, reflects on the very restricted set of shapes that are provided, including only (1) 2D shapes, namely, arcs, ellipses, lines, points, rectangles, and triangles; (2) curves, namely, beziers, and splines; and (3) 3D shapes, namely, boxes and spheres. This limited set prevents from developing minimally sophisticated GD programs because not even the most basic operations, such as intersection, subtraction, and union, are provided. Moreover, *Processing*, and *PLaSM*, do not integrate with CAD applications.

2.2.7 Python

Python ([Rossum & Drake, 2003](#)) is a general purpose PL, not a GD language. Therefore, this section focuses on the GD domain-specific features provided by *Blender* for *Python* scripts. *Python* is compared to other PLs, namely, *Tcl* (desktop applications), *Perl* (systems), *Ruby* and *Java* (web-based applications), and *Scheme* (teaching). The key features of *Python* include (1) clear syntax, strong introspection capabilities, and several standard and community developed libraries; (2) easy integration with applications written in other PLs to provide built-in facilities, such as, interpreters; and (3) language extensibility through C and C++ (*Python*), *Java* (*Jython*), or .NET (*Iron Python*), depending on the implementation.

Blender uses a scene graph ([Cunningham & Bailey, 2001](#)) to organize the scene objects. A scene graph is an abstract data type usually implemented as a tree or a directed acyclic graph (DAG). Scripts can access the scene graph, for example, to add objects, such as, shapes, cameras, and constraints, and apply transformations, such as, translation, rotation, and scaling. *Blender* exposes a number of *Python* types in addition to the scene graph, such as, group, bone, and mesh. With these facilities, scripts can create geometric scenes procedurally.

The mesh type is a sequence of interconnected points that represent a polygonal mesh. It is composed of vertices, edges, and faces, where (1) vertices are Cartesian tuples; (2) edges are line segments between vertices; and (3) faces are sets of edges, in most cases, triangles. While flat surfaces are easily

represented by meshes, curved surfaces are usually a faceted approximation where the number of surfaces determines the approximation degree. Meshes can also be created using prefabricated primitives, such as, sphere, cone, cube, and cylinder, and manipulated by a set of operators, such as, extrusion, screw, spin, split, and subdivide.

Scripts can also define drawing algorithms using the *Python* bindings for *OpenGL*. The difference between a polygonal mesh and a drawing algorithm is that the former describes the geometry to be drawn by a generic algorithm and the latter makes it possible to render objects that cannot be (easily) described by polygonal meshes, for example, infinite surfaces.

2.2.8 SDL

*POV-Ray*¹ is an open source ray-tracer that is programmable via *SDL*. *SDL* programs are divided in a declarative description of the scene to be created and an imperative part for programming the ray-tracer. The declarative part is clearer and more concise than the imperative one, but it requires prior knowledge of the scene, which is not always the case. Macros are the main abstraction mechanism and require special care. Functions are also provided but encapsulate only mathematical expressions. Moreover, macros and functions combine poorly due to several language restrictions.

Being a ray-tracer, *POV-Ray* can provide geometric shapes that most CAD applications cannot, for example, (1) infinite solid primitives, such as, polynomial surfaces, paraboloids, and hyperboloids, described by n^{th} order polynomials; and (2) isosurface and parametric objects described by implicit functions and parametric equations, respectively.

Operations are very limited: there is little more than affine and Constructive Solid Geometry (CSG). For example, it is not possible to perform (1) lofts, sweeps, and extrusions; and (2) shape introspection, namely, solid, face, and edge explosion. Nevertheless, *POV-Ray* is mostly concerned with ray-tracing, therefore, sophisticated geometry is imported from modeling applications ([Cabeçinhas, 2010](#)). Finally, ray-tracers are, in most cases, over computationally intensive for interactive parameter experimentation, which is essential for GD.

2.2.9 TikZ

TikZ is a 2D graphics *LATeX* package for drawing and embedding pictures in documents. *TikZ* programs are a sequence of commands that draw shapes, such as, lines, curves, rectangles, circles, and nodes, with variety of options for changing their behavior.

TikZ pragmatics can be metaphorically described as a user controlled pen that walks within the picture, drawing and placing shapes. The pen trajectory is described by a path, which is composed

¹<http://www.povray.org/>

of (1) coordinates, used to specify relevant points in the picture, for example, the center of a circle; (2) path extension operations, which make use of brief syntax to connect coordinates, for example, straight (–), perpendicular (–|, |-) or curve lines (., arc); and (3) path construction operations, which place shapes, such as, circles and rectangles.

Another essential concept is the node, which represents a small part of a picture. It can be used for placing text and drawing shapes. More importantly, nodes are used for establishing relative positioning using an anchoring system that gives a fine control over placement. The anchoring system combines cardinal directions with bounding rectangles. For example, the north-east anchor of an object is the top-right corner of its bounding rectangle. When creating a node at a particular coordinate, *TikZ* can be asked to place that node shifted around in such a way that a certain anchor is at a specified coordinate.

There are different options to change the behavior of a command. For example, parameters can change position, rotation, and shading. For some values, such as distance, units of measurement, for example, centimeter (*cm*) and millimeter (*mm*), can also be used.

Several visual options, such as shading, can be grouped in a named structure, called style, which can be used on different commands. A style can also be parametrized, such that, even after its definition, parameters can still be added or overridden. There are several predefined styles, such as decorations, which can be used to customize the appearance without changing the program, and there are also built-in templates for creating automata, Petri nets, calendars, mind-maps, and trees.

For programs that perform complex geometric calculations, *TikZ* exposes its internal math engine, which includes, for example, functions for calculating intersections of straight and/or curved lines.

TikZ introduces scopes and the `foreach` statement, in addition to the control structures of *LATeX*. Transformations and options have a local effect, and are automatically created according to paired brackets, although manual control is also possible.

2.2.10 VisualScheme

VisualScheme (Leitão, Cabecinhas, & Martins, 2010) is programming environment for GD. It inherits the pedagogic qualities of the *Scheme* PL (Chen, 1992; Berman, 1994; Felleisen, Findler, Flatt, & Krishnamurthi, 2004a, 2004b; Marceau, Fisler, & Krishnamurthi, 2011) to provide a platform that can be used to teach programming to architecture students, who do not have a background in Computer Science.

VisualScheme integrates with CAD applications and provides several features for GD. For example, it implements different coordinate systems, namely, Cartesian, polar, cylindrical, and spherical. The use of an adequate coordinate system reduces the need for trigonometric calculations. It includes also sophisticated abstraction mechanisms, recursion, higher-order and anonymous functions, visual widgets, and the ability to use different programming paradigms. These features simplify the programming task and the resulting programs are more concise and elegant.

Despite the identified advantages, there are three important drawbacks, namely, (1) users have to spend time learning a *TPL*, namely *Scheme*, that is relatively unknown in the GD community; (2) it becomes difficult to share and reuse programs written in different languages; and (3) similarly to most GD languages, *VisualScheme* forces the use of a particular CAD package, contributing to a known problem of current CAAD-education ([Penttilä, 2003](#)): students become experts in a single CAD application.

2.2.11 Textual Language Analysis

Table 2.1 summarizes the distinguishing features of the *TPLs* discussed in this section. Several features were excluded from this analysis because they were either identical in all *TPLs* or less relevant for the comparison, namely, (1) error systems based on numerical codes and exceptions; (2) memory management; (3) integer, float, and string data types; (4) native or C-like Boolean expressions; (5) arithmetic, logical, and relational operators; (6) global and local variables; (7) point, vector, plane, curve, surface, mesh, and solid data structures; and (8) maps, trees, and arbitrary precision numbers. From the observation of the table, we can conclude 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.

2.3 Visual Programming Languages

In addition to *TPLs*, there are Visual Programming Languages (*VPLs*). A *VPL* program is a bi-dimensional representation consisting of iconic components that can be interactively manipulated by the user according to some spatial grammar ([Myers, 1990](#)). In general, these components are boxes, which represent shapes and operations, and connectors linking these boxes, which establish dataflow between components, such that the output of a component is the input of another. As an example, Figure 2.1 shows a *Grasshopper* program that computes the coordinates of a conical spiral. *Grasshopper* is one of the most popular *VPLs* for GD.

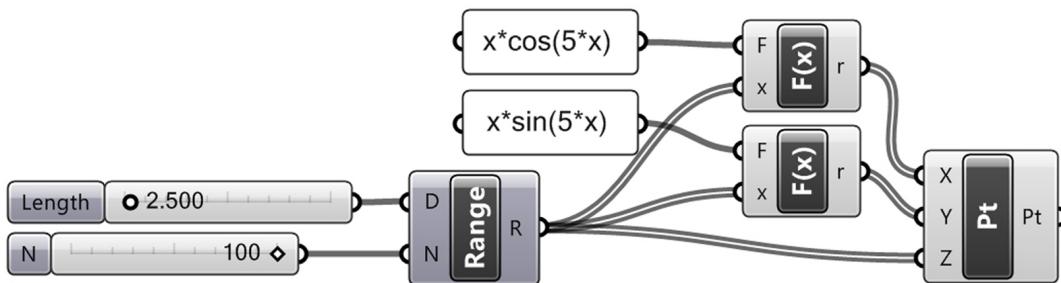


Figure 2.1: *Grasshopper* program for computing the coordinates of a conical spiral

Textual and visual PLs have significant differences in the learning curve. While *VPLs* have shorter

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 2.1: Distinguishing features of GD languages

learning curves, making them more adequate for novice programmers, large and complex *VPL* programs require more time to understand, maintain, and adapt to changing requirements, than *TPL* programs (Leitão & Santos, 2011). As a result, the time invested in a *TPL* is quickly recovered once the complexity of the design task becomes sufficiently large.

Grasshopper is one of the most popular *VPLs* for GD, therefore, it will be used as representative of *VPLs* to exemplify several problems of the visual approach. *Grasshopper* is based on the dataflow paradigm, which is too restrictive, making it difficult to express some control structures, such as, iteration or recursion. In *Grasshopper*, this is not a serious problem because most components implicitly map operations over sequences of values, which is enough in some cases. Moreover, *Grasshopper* implements a large set of primitive components, such as, ranges, mappings, and geometric operations, some of them with a high degree of sophistication, allowing an effective reduction in the implementation effort.

However, it has been repeatedly reported (Chok, 2011; Miller, 2011; Stouffs & Chang, 2010) that it might be difficult or impossible to describe an algorithm without resorting to custom components. *Grasshopper* provides custom components for advanced tasks, which are textually scripted by the user, thus contradicting the visual nature of the language.

Moreover, with increasingly large and complex programs, connectors become tangled, making programs very hard to read and maintain. In this case, users are advised to replace connectors with wireless endpoints, which do not have a visible line (Figure 2.2). Nevertheless, these endpoints make the situation considerably worse because users must inspect components one by one to understand which endpoints are paired. In general, the readability and maintainability problems arise from the visual nature of these languages. But there are relevant features particular to each *VPL* that were not discussed so far. The rest of this section details *Grasshopper*, *GenerativeComponents*, *CGA*, and *Hypergraph*.

2.3.1 Grasshopper

Grasshopper provides interactive input widgets, which are a better alternative to number and string literals, and lists of values, which are the only options in most *PLs*. For example, numerical inputs can be connected to sliders. When a slider is dragged, the program is recomputed with the new values. For small programs, it is possible to adjust the sliders interactively and see the model regenerate in real-time. However, for large and complex programs, the time needed to propagate the values is unreasonable. As a result, there is a severe break in interactivity and even the most patient designer will avoid this feature.

Another form of feedback occurs when a component is added to a program, causing the *Rhinoceros3D* model to be immediately updated. Moreover, when the user selects a component, the geometry generated by that component is highlighted in that model. Even more helpful would be the converse association but, unfortunately, this is not supported: it is not possible to select a shape in the model to identify the corresponding program component.

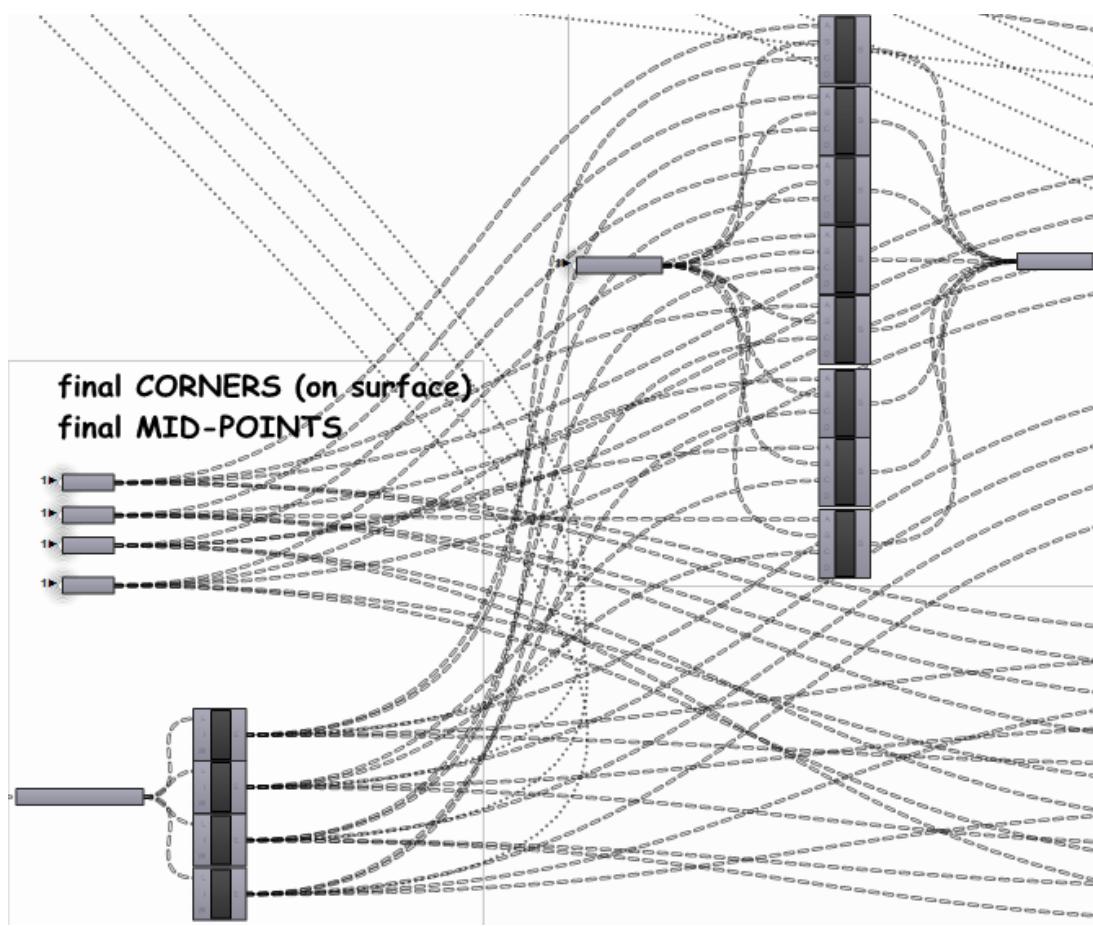


Figure 2.2: *Grasshopper* program fragment with tangled connectors and wireless endpoints

Most operations handle transparently a single value, or a list of values, as parameter. For example, a list of spheres connected to the translation operation implicitly translates all the spheres in the list. Even though this behavior is very useful, it can introduce difficult semantics. For example, the addition operation applied to lists of different lengths has several possible outcomes, namely, (1) trim the result according to the shortest list; (2) perform addition after padding the shorter lists according to the longest list; and (3) perform addition on the cross product of the lists. Data matching strategies are used to specify how the operation should handle each of these cases. However, these strategies are confusing and users must inspect operations one by one to understand the meaning of the program.

Due to the poor abstraction mechanisms, users rely extensively on *copy/paste*, which easily propagates errors and makes programs longer than they should be. There is a special component, the cluster, which allows users to treat a subset of components, including other clusters, as a single component. This can have a significant impact in the clarity of programs and it improves the reuse of its parts. Unfortunately, it still requires *copy/paste* and does not really represent an abstraction: each cluster is independent from its copies, thus preventing centralized modifications.

2.3.2 GenerativeComponents

GenerativeComponents (Aish & Woodbury, 2005) is a parametric and associative GD system used to generate geometry and implement several processes related to architecture and civil engineering, such as, measurement, evaluation, configuration, and fabrication. It provides three forms of user interaction, namely, (1) direct manipulation of geometry; (2) definition of relationships among geometric elements; and (3) textually scripted algorithms. These forms of interaction correspond to different, but synchronized, views of a single model (Menges, 2006).

The recommended learning path for *GenerativeComponents* consists of (1) designing using the interactive Graphical User Interface (GUI); (2) writing simple scripts using the formula bar and *GCScript*; and (3) developing complex programs through C#, a *TPL* mainly used for large-scale software development. This shows that, similarly to *Grasshopper*, for complex design tasks, users need to become proficient in a *TPL*. However, it is generally admitted that this need comes later in *Grasshopper* than in *GenerativeComponents*, which might explain why the former is considered easier to learn and use.

2.3.3 CGA

CityEngine (Müller, Wonka, Haegler, Ulmer, & Gool, 2006) is a modeling application for buildings and cities. It features a *Python* scripting interface and a dedicated PL for procedural modeling: CGA (Computer Graphics Architecture). This PL allows the definition of shape grammars using derivation rules, parameters, and attributes, and it can be considered a *VPL/TPL* hybrid because the built-in editor supports both textual and visual interaction (Figure 2.3).

Similarly to *Grasshopper*, the visual representation used in CGA results in readability and maintainability problems for large and complex programs. Moreover, the single paradigm used in CGA makes the language too restrictive for GD in general.

2.3.4 Hypergraph

Maya is a popular modeling application with a built-in scripting editor. Textual scripts can be written in *MEL* and *Python*. When users interact with *Maya*, via a script or the GUI, an interactions history is created, which can then be visualized and manipulated via visual editors such as *Outliner* and *Hypergraph* (Wilkins, Kazmier, & Osterburg, 2005). *Hypergraph* provides two editable views: (1) the hierarchy graph displays scene items according to their parent-child relationships; and (2) the dependency graph represents model construction history. However, unlike *GenerativeComponents* and CGA, the relationship between the scripts and the editors is unidirectional, meaning that changes in these editors are not reflected on the scripts. Despite the visual editing features of *Hypergraph*, a *TPL* is still required for creating more complex algorithms.

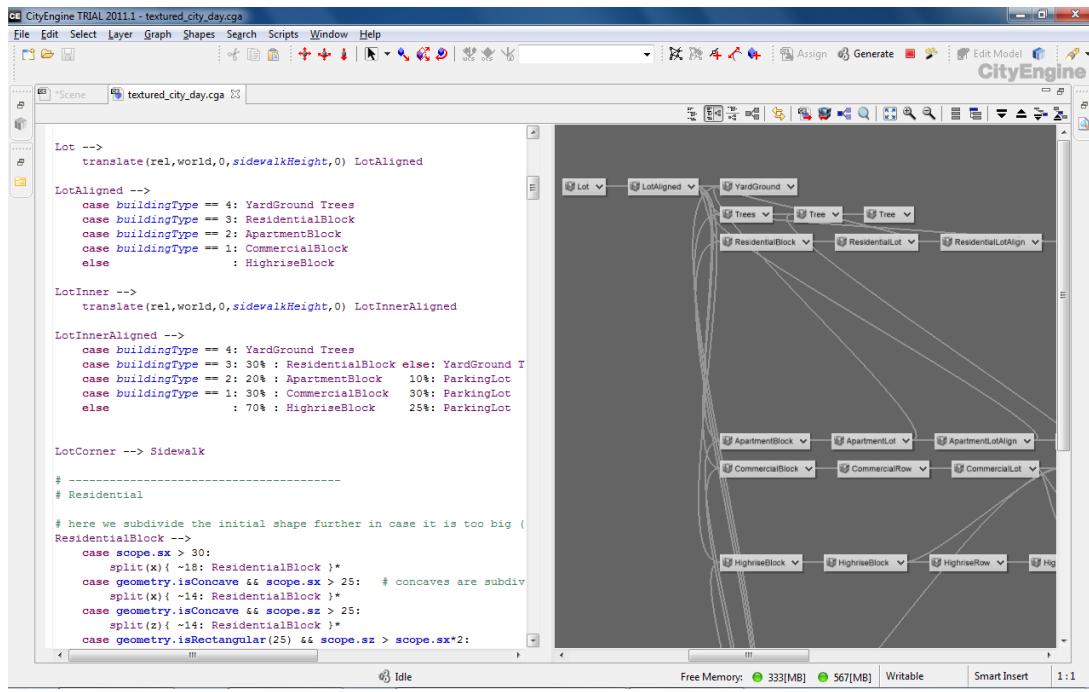


Figure 2.3: CGA editor showing textual and visual representations of a shape grammar

2.3.5 GD System Analysis

This section presents a survey on the domain-specific features of the most used GD systems, namely, *AutoCAD*, *Blender*, *3ds Max*, *Maya*, *Rhinoceros3D*, *TikZ*, *PLaSM*, and *ArchiCAD*. This survey is important to decide what primitives are relevant for a modern programming environment. The survey focused on (1) geometric shapes, such as, arc, rectangle, box, and sphere; (2) parametric curves and surfaces, such as, splines and NURBS; (3) geometric operations, such as, union, intersection, sweep, and extrusion; and (4) other architectural and computer graphics objects, such as, particle systems, doors, and windows. Note that operations focused on CSG, therefore, operations that deform shapes, such as, bend, bevel, cross section, and skew, were not considered. The results of the survey are in Table 2.3 and Table 2.4. These tables use several symbols whose meaning is in Table 2.2.

Symbol	Description
blank	No match
✓	Match
≈	Partial match
~	Possible match
footnote	Match and footnote
≈footnote	Partial match and footnote

Table 2.2: Survey legend

A description of the notes used in the survey tables follows:

- 1 All primitives are either meshes, or metaobjects (implicitly defined), or NURBS

	AutoCAD	Blender [1]	3ds Max	Maya	Rhinoceros3D	TikZ	PLaSM	GDL
2D shapes								
Arc	✓		17	✓	✓	✓		✓
Elliptical arc	✓		✓			✓		
Circle	✓	22	17	10	✓	✓		✓
Donut	✓		17					
Ellipse	✓		17		✓	✓		
Helix 2D [6]	✓	25	17	✓	35			
Line segment	✓		17	~	✓	✓	52,53	✓
NGon	✓	✓	17	~	✓			✓
NURBS	✓	✓	✓	✓	✓			
Rectangle	✓	✓	17	✓	✓	✓	52	✓
Spline			✓	✓	~			✓
Star			17		42			
Text	✓	✓	17	✓	✓	50		✓
3D shapes								
Box	✓	✓	✓	✓	✓	52	✓	
Cone	✓		✓	✓	✓			✓
Cut cone	✓				38			✓
Cylinder	✓	✓	✓	✓	✓	✓	✓	✓
Elliptical cone	✓							✓
Gengon [23]			✓					
Helix 3D			✓	7				
Mesh	✓	✓	✓	✓	✓			✓
Paraboloid [37]	≈24	≈24	≈21		✓			✓
Pipe [5]		✓	13	✓	13			
Platonic solids			14	8	≈43			
Pyramid	✓	2	✓	✓	✓		49	
Sphere	✓	3	12	✓	✓			✓
Spindle [16]			✓					
Superellipsoid			15		36			
Torus	✓	✓	✓	✓	✓			
T. icosahedron				✓	≈43			
Wedge	✓	4	11	✓	47			✓

Table 2.3: Survey of shapes

Operations	AutoCAD	Blender [1]	3ds Max	Maya	Rhinoceros3D	TikZ	PLaSM	GDL
Bend	✓	✓	✓	✓	✓			
Bevel		✓	✓	✓	44			
Cross section	✓	≈34	✓		39			
Extrusion	✓	✓	✓	✓	✓	✓	✓	
Guided loft	✓			✓	✓			
Intersection	✓	✓	✓	✓	✓	✓	✓	
Lattice [19]			✓					
Loft	✓	48		✓	✓			
Mirror	✓	✓	✓	9,29	✓	51		
Move	✓	✓	✓	✓	✓	✓	✓	
Offset	✓	20	✓	✓	✓			
Path loft	✓		✓					
Revolution	✓	✓	18	✓	✓			✓
Rotation	✓	✓	✓	✓	✓	✓	✓	✓
Scale	✓	✓	✓	✓	✓	✓	✓	✓
Skew		26	✓	30	✓	54		
Slice	✓	27	✓	31	40			✓
Subtraction	✓	✓	✓	✓	✓	✓	✓	
Sweep	✓	✓	✓	✓	✓			41
Thicken	✓	33	✓	≈32	≈45			
Union	✓	✓	✓	✓	✓	✓	✓	
<hr/>								
Other Objects								
Particle systems		✓	✓	✓				
Doors	28		✓		≈46		✓	
Windows	28		✓		≈46		✓	
Stairs	28		✓		≈46			
Railing	28		✓		≈46			
Wall	28		✓		≈46		✓	
Foliage	28		✓		≈46			

Table 2.4: Survey of operations and other GD objects

- 2** A pyramid is a faceted approximation of a cone
- 3** i.e., UVSphere, IcoSphere e NURBS sphere.
- 4** A prism is a faceted approximation of a cylinder
- 5** Result of subtracting two concentric cylinders of different radius
- 6** i.e., spiral
- 7** Helix 2D and 3D are the same
- 8** Tetrahedron, octahedron, dodecahedron, icosahedron
- 9** The same can be achieved through scaling with negative values
- 10** Dedicated NURBS tools
- 11** Triangular prism
- 12** Sphere and GeoSphere
- 13** i.e., tube
- 14** i.e., tetrahedron, cube, octahedron, dodecahedron, icosahedron, star
- 15** i.e., Chamfer box, Chamfer cylinder, oil tank, capsule
- 16** i.e., cylinder with conical caps
- 17** Through spline tool
- 18** i.e., lathe
- 19** The lattice modifier converts the segments or edges of a shape or object into cylindrical struts with optional joint polyhedra at the vertices
- 20** Through array modifier
- 21** Through a plug-in:
<http://www.creativecrash.com/3dsmax/downloads/scripts-plugins/modeling/c/quadratic-primitives-nurbs>
- 22** Through NURBS
- 23** Use Gengon to create an extruded, regular-sided polygon with optionally filleted side edges
- 24** Through a *Python* script:
http://wiki.blender.org/index.php/Extensions:2.4/Py/Scripts/Add/Add_Mesh_Paraboloid
- 25** Through screw tool
- 26** Through PET and shear
- 27** Through knife tool
- 28** Available in *AutoCAD* Architecture
- 29** Through mirror geometry and mirror cut
- 30** Through shear
- 31** Through cut faces tool
- 32** Through extrude tool
- 33** Through solidify selection
- 34** Through a *Python* script:

<http://wiki.blender.org/index.php/Extensions:2.4/Py/Scripts/System/CrossSection>

35 Helix and spiral

36 i.e., ellipsoid

37 i.e., cone-like ellipsoid

38 i.e., truncated cone

39 Section and CSec

40 Through wire cut; cutplane only creates planes, it does not perform the actual cut

41 Through sweep or tube

42 Through polygon

43 Through downloadable files:

<http://www.rhino3d.com/pythposter/pyth3dm-eng.html>

44 Through chamfer surface

45 Through T-Splines which is a commercial product:

<http://www.tsplines.com/products/tsplines-for-rhino.html>

46 Through VisualARQ plug-in:

<http://www.rhino3d.com/resources/display.asp?language=&listing=4424>

47 i.e., prism

48 i.e., skinning

49 Implemented as an operation, not a primitive

50 *TikZ* is embedded in *LATEX* so text can also be achieved through Latex

51 Through negative scaling

52 Through cuboid primitive

53 Through simplex primitive

54 Through shear primitive

3

Modern Programming Environment

3.1 Design Principles

Architecture coding is evolving to GD and leading to the adoption of computational methods, clearly showing that designers are in desperate need of a modern programming environment specifically tailored for GD. Unfortunately, as shown in the previous section, current systems are not capable of responding to this need because either they are old or obsolete, or they enforce particular programming methods that are inadequate, or they are not pedagogic, meaning that they are not designed for the particular programming skills of the GD community.

Most GD systems are attached to a CAD application. For example, *AutoLISP* is attached *AutoCAD*, and *RhinoScript* and *Grasshopper* to *Rhinoceros3D*. This combination of GD and CAD tools has significant advantages, including leveraging already acquired experience and facilitating the subsequent use of the generated model. Unfortunately, traditional CAD technology is preventing GD tools from evolving freely. The fact that CAD applications were designed for the interaction between a human and a computer ([Sutherland, 1963](#)) is also a source of several problems. But it is not the only source: *TPLs*, such as, *AutoLISP*, *RhinoScript*, and *GDL*, are old and obsolete, providing insufficient support for GD. And modern *VPLs*, such as, *Grasshopper* and *GenerativeComponents*, are very restrictive and scale poorly with the size and complexity of the design task, making them difficult to use. Finally, advanced PLs, such as *PLaSM*, target the programming skills of mathematicians and expert software engineers, making them unsuitable for designers.

In order to overcome this problem, 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 the design principles that are explained in this section.

3.2 Portability

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. Portability and reusability are qualities that allow software to adapt to new environments. For example, if a given CAD application replaces a long provided PL with a new one, all programs written in the first PL become useless. In this case, designers have two options: either discard the programs or rewrite them in a supported PL. Unfortunately, both options are inadequate.

Software reuse is also important because it increases productivity and, in some cases, it is the key factor for the survival of small- and medium-sized communities. Several years of software engineering research and practice have shown the importance of collaborative programming environments. For example, the success of mainstream PLs, namely, *Java* and *C#*, is due in large part to the extensive API provided by these PLs and the thousands of libraries available. The *Perl* PL is another example: this PL is supported by an online collection of software and documentation called the Comprehensive *Perl* Archive Network (*CPAN*). Through *CPAN*, *Perl* users have access to tens of thousands of modules that practically cover every programming task. *CPAN* relies on the contributions and collaborative development of the *Perl* community and it is one of the reasons for the success and survival of *Perl*.

Even though there are thousands of *AutoLISP* scripts available on the Internet, the fact is that there is no centralized infrastructure similar to *CPAN*, and the *AutoLISP* programming environment provides no mechanisms for collaborative development. The same can be said about *RhinoScript* and several other GD languages. As a result, 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. To this end, this programming environment should support multiple CAD applications, which can be used interchangeably to display the generated models, and multiple PLs, in which users can write their GD programs. But it is also necessary to rethink the geometric support of these PLs.

3.3 Parametric Elements

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. In GD, it is possible to write algorithms that work with parametric elements. For example, the parametric function $f(t) = (t, 0, \sin(t))$ together with the domain $[0, 2\pi]$ represents a sinusoidal curve segment on the XZ plane. Ideally, this function could be used, for example, as the path in a sweeping operation together with a circle profile to create a sinusoidal tube.

However, CAD applications cannot handle functions directly. As a result, designers must first transform the circle profile and the sinusoidal path into shapes. While the former is natively supported by most CAD tools, the latter usually requires the designer to interpolate the sinusoidal function on a set of sampling points, distracting him from the essence of the design task and making the problem unnecessarily complex, especially when the required sampling is nonlinear. In order to overcome this complexity, geometric operations should accept not only geometric shapes but also parametric objects, and any sampling and interpolation required by the CAD application should be handled automatically by the programming environment.

3.4 Functional Operations

Another problem of current CAD applications is that, in some cases, objects are consumed by geometric operations to create other objects. For example, the intersection operation applied to two solids might consume those solids to produce the result, or consume one of them and destructively modify the other to become the result. Several operations, such as, loft and revolve, have similar behavior. Other operations, such as, sweep, can have mixed behavior, where the path is consumed but the profile is not.

Even though this behavior might make sense in the interaction between a human and a CAD application, GD algorithms must rely on a consistent behavior in order to be correct. For example, consider the union of two cylinders R_0 and R_1 , subtracted by their intersection (Figure 3.1).

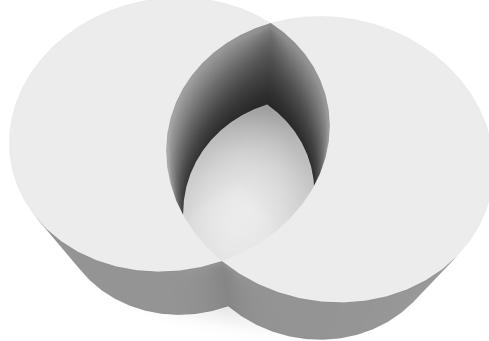


Figure 3.1: Symmetric difference of cylinders

In mathematical terms, this example is the *symmetric difference* (Δ) of R_0 and R_1

$$\Delta(R_0, R_1) = (R_0 \cup R_1) - (R_0 \cap R_1)$$

where \cup , \cap , and $-$, represent the union, intersection, and subtraction of shapes, respectively.

Unfortunately, this mathematical definition is not valid when the union between R_0 and R_1 consumes R_0 , R_1 , or both, because it makes the subsequent intersection and subtraction impossible to compute. In order to overcome this problem, it is necessary (1) to prevent shapes from being consumed, or

(2) to create copies of the original shapes beforehand. Both alternatives require manual intervention from the designer, who must carefully place deletion or copying commands throughout his program. Knowing which objects must be copied/deleted is equivalent to a *garbage collection* process (Jones & Lins, 1996), a task that should be done automatically by the programming environment. In order to overcome this problem, all geometric operations must be functional, meaning that they should not consume their arguments.

3.5 Dimension Independent Operations

In most GD languages, there is no uniform treatment for one-, two-, and three-dimensional shapes. For example, depending on the dimension of each shape, their intersection might be implemented by a large number of distinct procedures, for example, curve-curve, curve-surface, surface-surface, curve-solid, surface-solid, and solid-solid, as well as special cases for planar surfaces, straight lines, closed coplanar lines, and mesh-based surfaces. As a result, it is difficult for designers to write generic programs. For example, if all operations are generic, in the sense that they work uniformly in n -dimensional space, a user-defined symmetric difference operation is also generic. Unfortunately, this is not the case of current GD languages. 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. As exemplified in Figure 3.2, a vertically aligned cylinder can be described by the center and radius of its base and by its height: (1) when both radius and height are positive, the shape is three-dimensional; (2) when the radius is positive but the height is zero, it becomes a bi-dimensional circle; (3) when the height is positive but the radius is zero, it becomes a one-dimensional line segment; and (4) when both radius and height are zero, it becomes a zero-dimensional point. For mathematical (and computational) correctness, it is important that all operations accept and properly handle all cases, even though in some cases it might be impossible to visualize the result.

3.6 Algebra of Sets

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 admit identity and absorbing elements, for example, (1) for union, the identity element is the empty set \emptyset ($S \cup \emptyset = S$) and the absorbing element is the universal set U ($S \cup U = U$); (2) for intersection, the identity element is the universal set ($S \cap U = S$) and the absorbing element is the empty set ($S \cap \emptyset = \emptyset$); and (3) the subtraction operation has the empty set as both right-identity ($S - \emptyset = S$) and left-absorbing element ($\emptyset - S = \emptyset$).

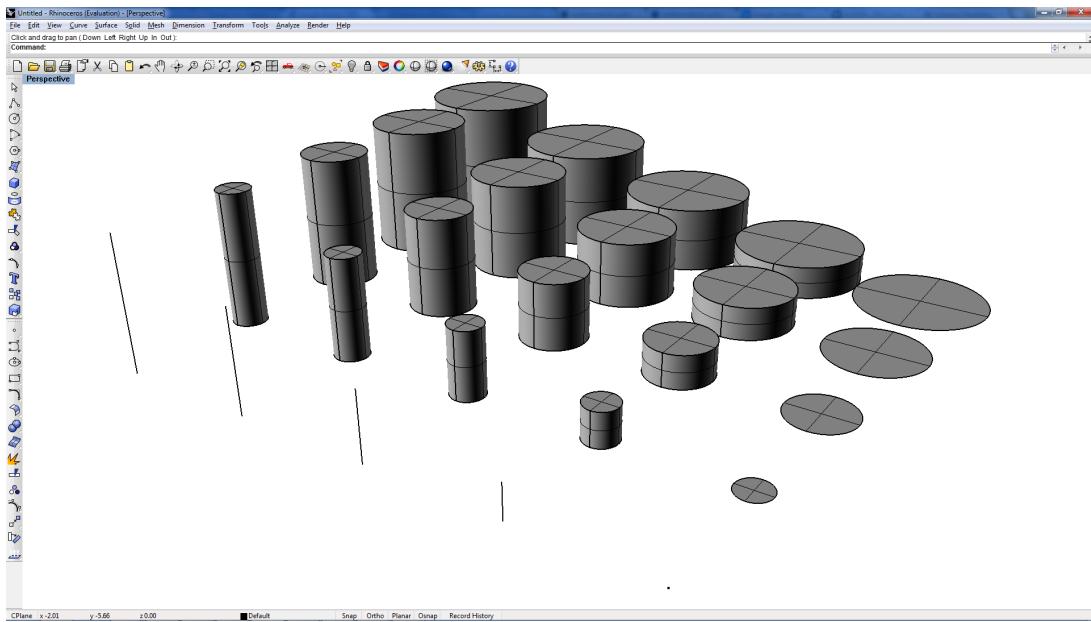


Figure 3.2: Morphing a cylinder into a disk, a line, and a point

Unfortunately, CAD applications do not implement this mathematical representation. For example, reconsider Figure 3.1 and imagine the result of moving the cylinders apart from each other or, alternatively, shrinking the radius of the cylinders: the inner hole becomes increasingly smaller until it disappears, resulting in the union of two cylinders. When the cylinders are separated, their intersection becomes empty:

$$(R_0 \cup R_1) - (R_0 \cap R_1) \equiv (R_0 \cup R_1) - \emptyset \equiv R_0 \cup R_1$$

Despite this equivalence, in *AutoCAD* and *Rhinoceros3D* an empty intersection results in an error. In *Grasshopper*, an empty intersection issues a warning but it does not fail. Unfortunately, the subsequent subtraction fails.

In fact, the majority of GD languages do not even implement the concepts of empty or universal set. Although they seem irrelevant in CAD tools, they are essential in GD languages to fully define certain operations. For example, the union of a set of shapes can be defined with recursion, but the recursion needs a base case that explicitly uses the empty set:

$$\begin{aligned} \bigcup(\{S_0, S_1, \dots, S_n\}) &= S_0 \cup \bigcup(\{S_1, \dots, S_n\}) \\ \bigcup(\{\}) &= \emptyset \end{aligned}$$

A similar reasoning applies to the intersection operation and the universal set.

3.7 Algebraic Equivalences

While sets allow understanding shapes from their mathematical point of view, algebraic equivalences are important to understand operations. For example, Figure 3.3 shows a shape consisting of a cylinder R_0 subtracted by a smaller cylinder R_1 and a sphere R_2 :

$$(R_0 - R_1) - R_2 \equiv R_0 - (R_1 \cup R_2)$$

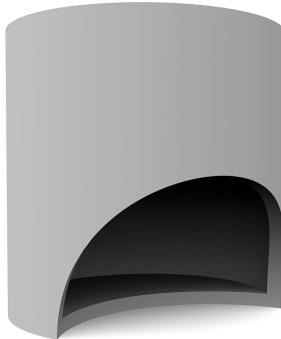


Figure 3.3: Hollow cylinder subtracted by a sphere

The previous equivalence shows that, in order to get the intended shape, one can either remove the sphere R_2 from the hollow cylinder $R_0 - R_1$ or remove the union of the inner cylinder R_1 and the sphere R_2 from the outer cylinder R_0 .

Unfortunately, this basic algebraic equivalence is invalid in several CAD applications. *Rhinoceros3D*, for example, currently cannot model hollow solids unless there is an interconnecting surface between the outer and inner surfaces. This means that a *Rhinoceros3D* user cannot use the first approach. Given the dependency on *Rhinoceros3D*, *Grasshopper* also suffers from this limitation. Other CAD tools, such as *AutoCAD*, do not have this problem and designers can use both approaches.

Nevertheless, what is significant is not the number of possible modeling approaches but their equivalence, which gives the designer freedom of choice. This property is 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.

3.8 Traceability

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. The programming environment should be capable of displaying this relationship because it is essential for program comprehension, maintenance, and debugging. Otherwise, it can be very difficult to find the causes of errors, or to identify the changes needed to adapt a GD program to some additional or different purpose, or to understand the impact of changes in a program. A capable programming environment should allow 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. This association is called traceability and few PLs fully implement it. For example, *Grasshopper* implements traceability, but only in the direction from the program to the model.

3.9 Immediate Feedback

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. With this mechanism, designers can not only better correlate the program and the model but also endeavor in design exploration. Several GD systems provide immediate feedback. For example, in *Grasshopper*, when a designer drags a slider the program is automatically re-executed. However, the model is updated only in real-time for very simple GD programs.

4 Rosetta

4.1 Design Requirements

The previous section explained the design principles for a successful programming environment for GD. These principles can be summarized in (1) portability of programs, (2) mathematical correctness, and (3) strong correlation between programs and models. Because currently GD systems do not implement these principles with the proper support for GD, a new programming environment, called *Rosetta*, was created. This section explains *Rosetta*, its main features, and the approach taken for each principle.

4.2 Software Architecture

Rosetta is a modern programming environment for GD designed to overcome the limitations of the most used GD systems. One of the most important limitations concerns the portability of programs written in the PLs provided by GD systems. For example, an *AutoLISP* program will execute in different versions of *AutoCAD*, sometimes requiring changes in few lines of code, but it will not execute in *Rhinoceros3D*. In this case, users must translate their programs to a PL that is supported by the target GD system, for example, *RhinoScript*. Translating programs might be a common task for a software engineer, but it is a difficult and error-prone task for a designer. Unable to overcome this limitation, designers become locked-in to a particular family of CAD applications and cannot reuse programs of other families.

Rosetta overcomes this 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. For example, Figure 4.1 shows *Rosetta* with a *JavaScript* program and corresponding geometry in *AutoCAD*.

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*.

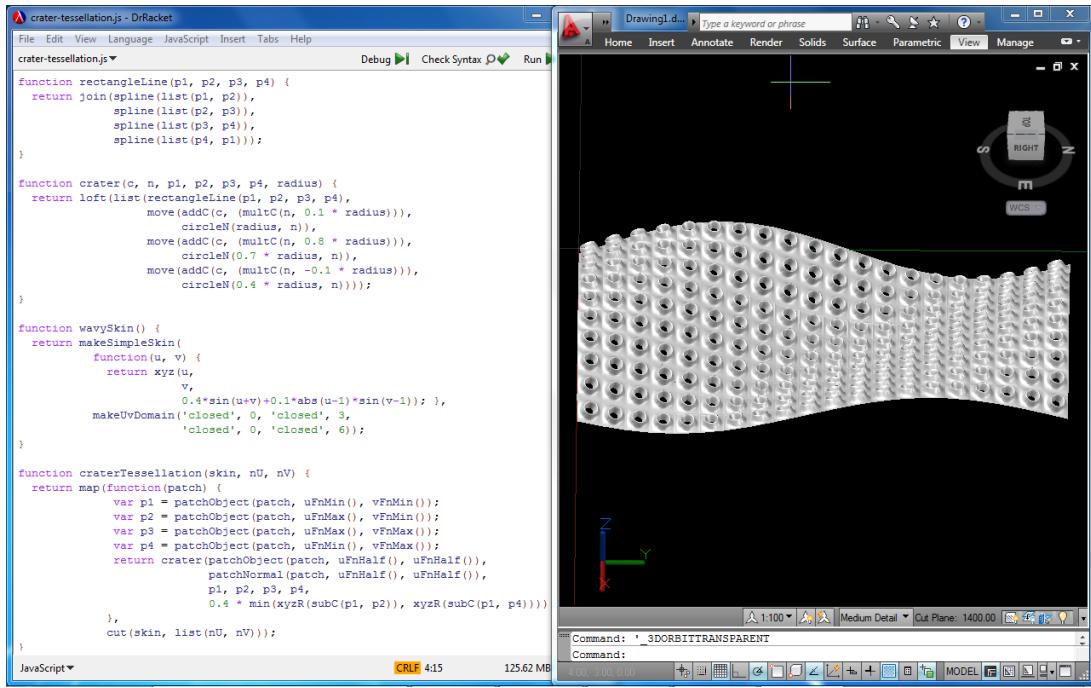


Figure 4.1: Rosetta with a *JavaScript* program and corresponding geometry in *AutoCAD*

In order to support this approach, the software architecture has a core component and several loosely coupled components for each frontend and backend: (1) the core component defines functionality that is common to all components; (2) a frontend component represents one PL and implements its linguistic and editing tools, such as, lexer, parser, compiler, and syntax highlighting; and (3) a backend component represents one CAD application and implements the connection between the programming environment and that application. While changes in the core component strongly affect all other components, changes in frontend or backend components are merely local. As result, new programming languages and CAD applications can be added to extend the programming environment without changing the remaining components or existing programs.

The core component provides an abstraction layer with the functionality that is common to the programming environment and all programs. This abstraction layer defines a portable API, which is the key for virtualizing different backends, and it includes (1) general purpose data types, such as, list, vector, and interval; (2) geometric data types, such as, shape, coordinate, matrix, bounding box, color, and material; (3) shape constructors, such as, point, circle, and box; and (4) geometric transformations, such as, translation, loft, extrusion, and sweep.

However, this virtualization results in that functionality that is not common to most CAD applications cannot be provided in the portable API. In order to overcome this problem, the abstraction layer does not restrict the use of CAD-specific functionality and designers can choose whether or not to use it. Giving access to CAD-specific functionality might result in programs that are not portable but it makes

Rosetta available to a broader audience of designers. In the following sections, the supported frontends and backends are explained in detail. Moreover, *DrRacket*, the editor used by *Rosetta*, is also explained.

4.3 Editor

DrRacket (Figure 4.2) is a programming environment designed to be pedagogic and to simplify the implementation of new PLs. It provides a text editor with the standard programming features, such as, text formatting, and syntax checking and highlighting. Programs are written in the *Definitions Window* and the *Run* button compiles the program and its dependencies, and initiates its execution. During execution, the *Interactions Window* becomes available, providing an interactive evaluator which can be used to quickly test the running program, add new definitions to the session, experiment with different parameters or, ultimately, evaluate any kind of expression. This evaluator is fundamental for incremental development and interactive testing, being one of the main sources of feedback during program development. There is also a debugger and a *JIT* compiler, which optimizes code on demand at runtime. *DrRacket* provides several PLs, such as, *Scheme*, *Scribble*, and *Racket*. *Racket* is a modern, functional PL and it is the main language of *DrRacket*.

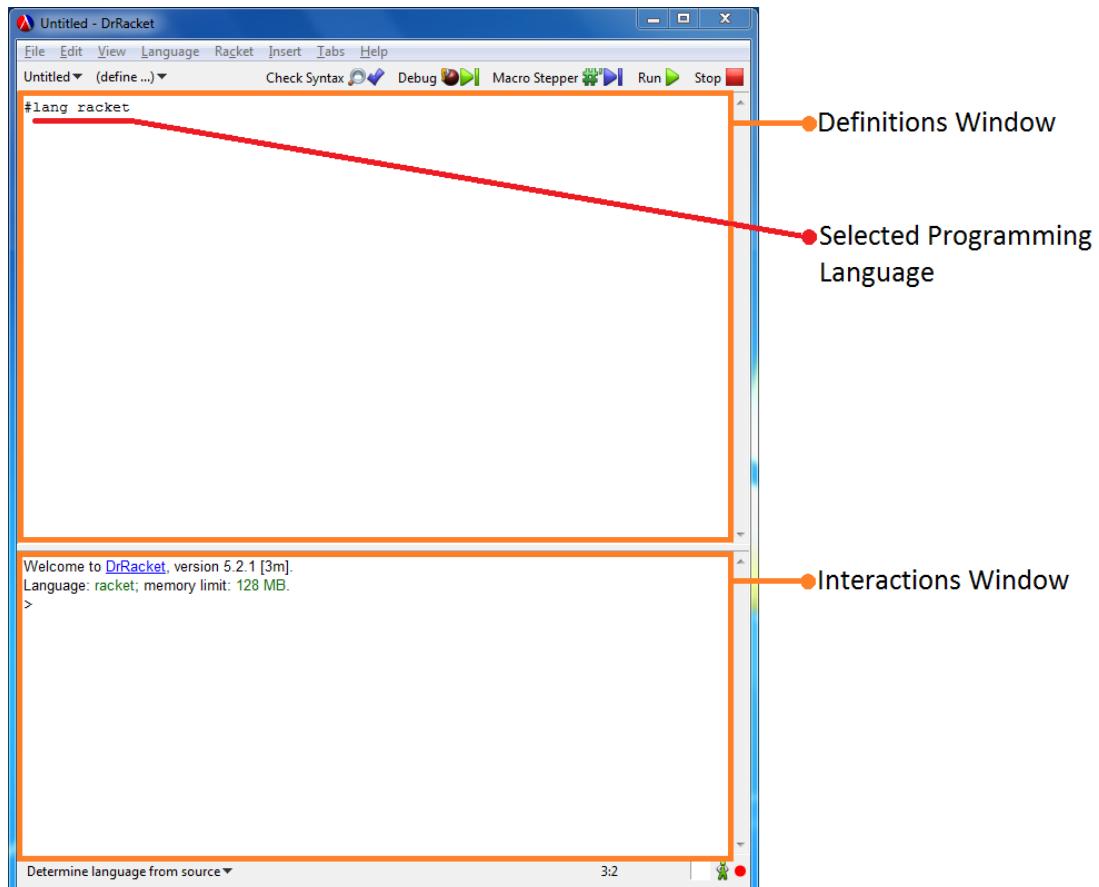


Figure 4.2: *DrRacket* window

4.4 Backends

Currently, *Rosetta* implements three backends, namely, *AutoCAD*, *Rhinoceros3D*, and *OpenGL*. Backends serve as modeling target and geometric kernel for complex calculations. These backends were implemented for different reasons, 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; Programs can use either backend with no additional modifications and, more importantly, the geometry created in each backend is identical (Figure 4.3, Figure 4.4 and Figure 4.5).

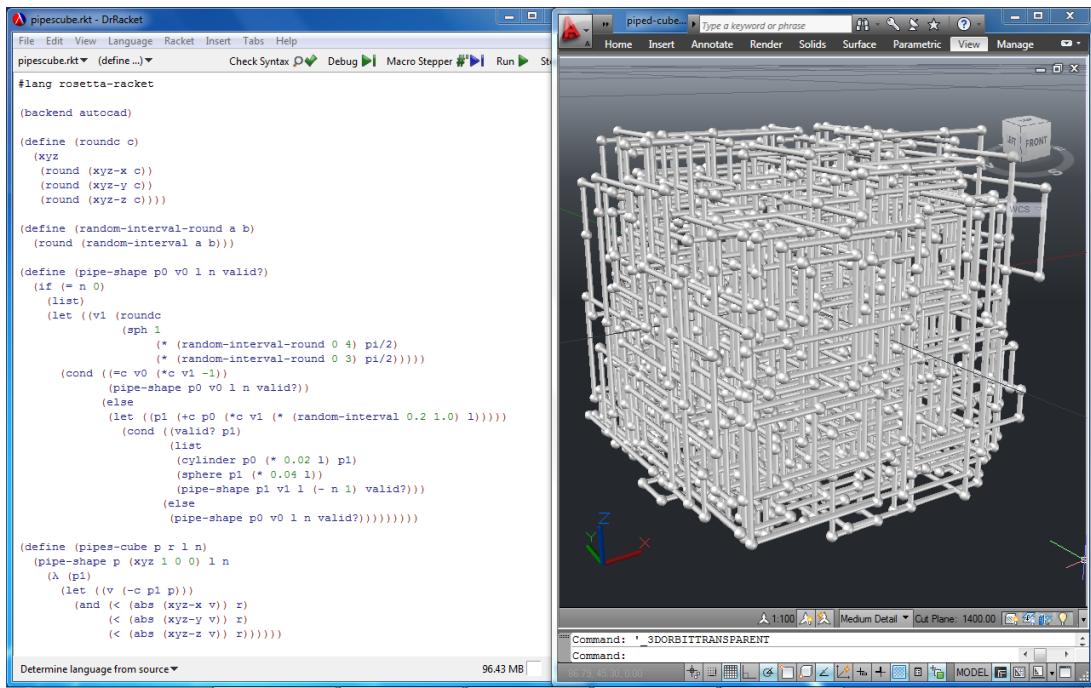


Figure 4.3: *RosettaRacket* program and corresponding geometry in *AutoCAD*

4.4.1 AutoCAD and Rhinoceros3D

AutoCAD and *Rhinoceros3D* provide more or less the same functionality but, in some cases, with different semantics. For example, a solid sphere in *AutoCAD* is the space enclosed by the surface of the sphere, whereas the same sphere in *Rhinoceros3D* is merely the spherical surface. This difference has very important consequences: for example, in *AutoCAD*, the subtraction of two concentric spheres of different radius results in a hollow sphere, whereas in *Rhinoceros3D*, the same operation results in an error because the two spherical surfaces do not intersect.

In order to overcome these semantic differences, each backend understands the limitations of the corresponding CAD application and provides ways around those limitations. As an example, consider

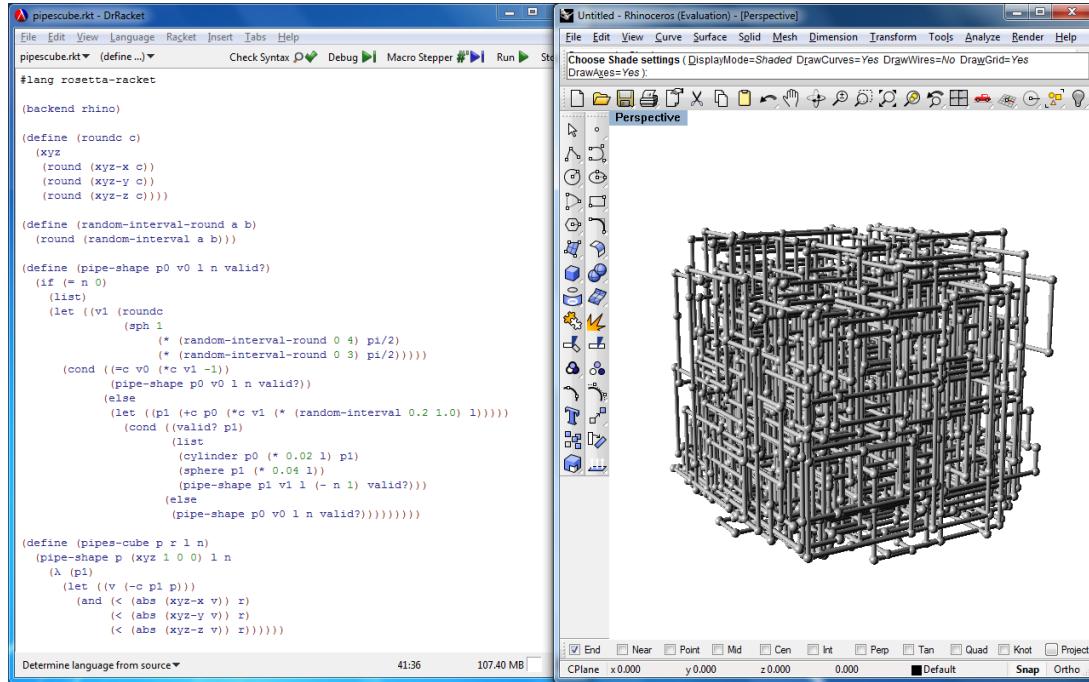


Figure 4.4: *RosettaRacket* program and corresponding geometry in *Rhinoceros3D*

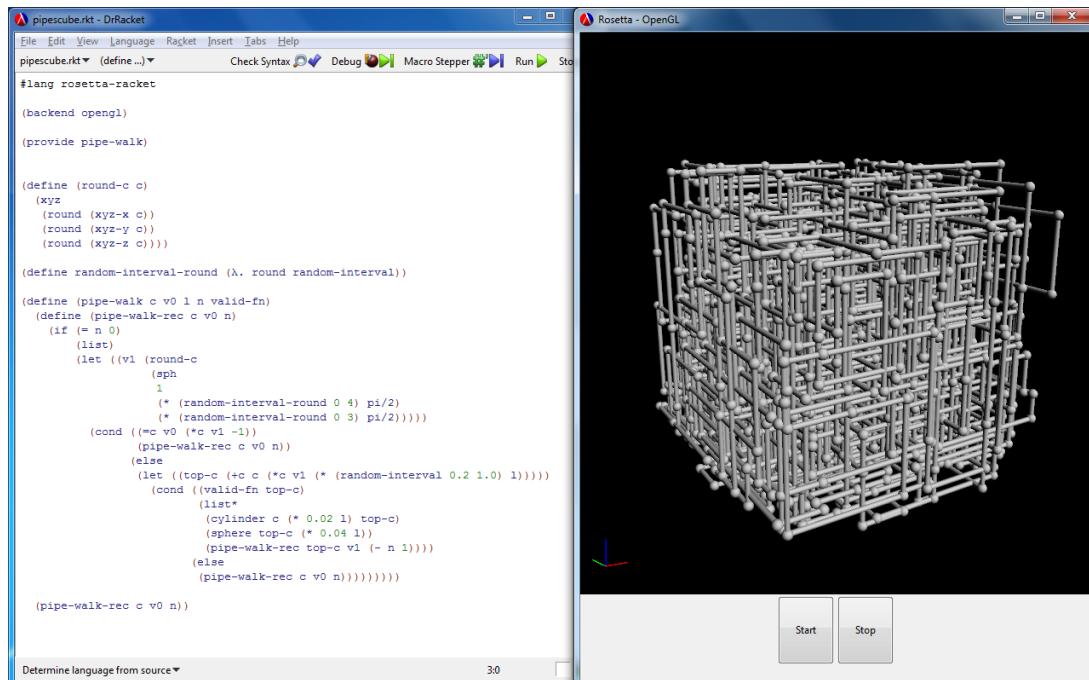


Figure 4.5: *RosettaRacket* program and corresponding geometry in *OpenGL*

the shape in Figure 4.6. This shape can be produced by subtracting cones from a hollow sphere. However, *Rhinoceros3D* cannot model hollow shapes (Section 3.7). Therefore, when using *Rhinoceros3D* as backend, *Rosetta* delays the creation of the hollow sphere until one of the cones perforates a hole in the outer sphere. This reordering of operations is automatically done by *Rosetta*, by application of algebraic rules, in order to correctly generate the intended geometric model. In general, *Rosetta* gives designers the illusion that they can use different modeling approaches. To this end, GD programs are evaluated in such a way that they can be executed within the limitations of that CAD application.



Figure 4.6: Hollow sphere pierced by several cones

However, the fact is that without proper CAD support certain operations are simply impossible. For example, while computing a hollow solid in *Rhinoceros3D*, *Rosetta* can eliminate the inner shape to avoid the subtraction error if it can prove that no other Boolean operations take place and that the inner shape is completely contained, i.e., not visible. However, in general the latter test is non-trivial.

4.4.2 OpenGL

The *OpenGL* backend does not depend on a fully fledged CAD application. Instead, it connects (almost) directly to the graphics device of the computer. This backend allows much faster rendering and, as a result, the designer can enjoy real-time feedback for larger inputs and for a broader spectrum of programs. When satisfied with the design, he can then switch to a normal CAD backend, such as, *AutoCAD* or *Rhinoceros3D*, and continue working as before. Table 4.1 shows the time taken by different backends for updating identical geometry (Figure 4.7, Figure 4.8 and Figure 4.9). It is visible that the *OpenGL* backend is considerably faster than the other backends. Note that this backend is still in a prototypical phase and further optimizations can be implemented to improve the running times.

Even though the *OpenGL* backend is faster than the other backends, it does not provide the same functionality. *OpenGL* is a rendering library, therefore, it provides several drawing procedures to op-

Example/Backend	AutoCAD	Rhinoceros3D	OpenGL
Orthogonal cones	1022	191	1
Möbius truss	28837	9235	4446
Scriptecture	21920	5088	210

Table 4.1: Time (in milliseconds) needed to update the generated design

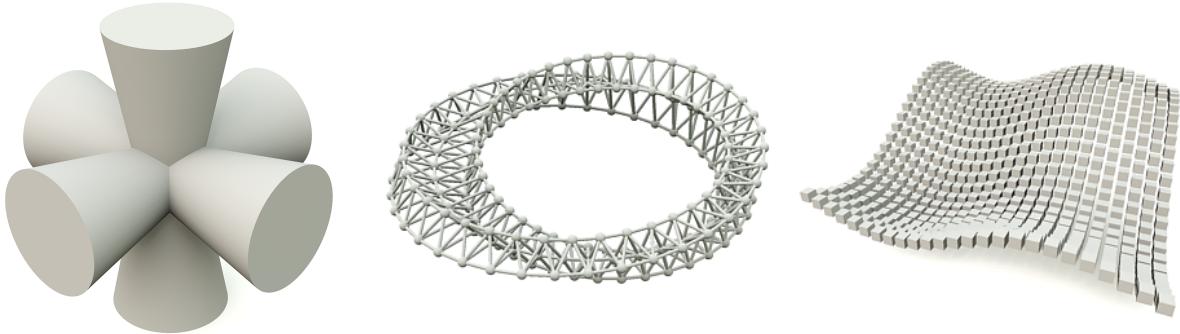


Figure 4.7: Orthogonal cones

Figure 4.8: Möbius truss

Figure 4.9: Scriptecture

erate in image space but, in most cases, objects do not have a parametric representation. This makes it difficult to implement some geometric transformations, such as, loft and sweep. Nevertheless, affine transformations are trivial because they map directly to *OpenGL* matrices, and Constructive Solid Geometry (CSG) can be performed in image space. The library *OpenCSG* is an example of such functionality, and it is planned to integrate *Rosetta*.

4.5 Frontends

Currently, *Rosetta* implements three frontends, namely, *AutoLISP*, *JavaScript*, and *RosettaRacket*, which is a customized version of *Racket*. *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 new syntactic forms that expand to *Racket* code. As a result, different PLs can interoperate because they are part of the same ecosystem.

With this approach, it is possible to virtualize the syntax of new PLs. However, some syntactic problems cannot be avoided. For example, it is difficult to define portable names that follow the naming convention of every PL because identifiers have different syntax. As an example, consider the coordinate addition operation. In *Racket*, a valid name for this operation is `+c`. Although this name is also valid in *AutoLISP*, it is invalid in *JavaScript* because the syntax of identifiers does not include the symbol `+`. On the other hand, a valid choice for *JavaScript* is `addC`, which does not follow the *Racket* or *AutoLISP* naming conventions, making it unsuitable for these PLs.

To overcome this problem, all names exported by *Rosetta* follow some naming convention and each

frontend is responsible for defining the proper bindings that translate the exported name into a valid name in the PL they provide. For example, for the coordinate addition operation (1) *Rosetta* defines the name *add-c*; (2) *Racket* and *AutoLISP* define the binding *+c*; and (3) *JavaScript* defines the binding *addC*. The following sections detail each of the frontends provided by *Rosetta*.

4.5.1 RosettaRacket

RosettaRacket is a PL that extends *Racket* to include bindings that follow the *Racket* naming convention, as well as to allow the definition of new syntactic forms without compromising the remaining frontends. Another advantage is that by using this frontend, users no longer need to manually import the *Rosetta* library. This has a significant impact in the reduction of the number of prerequisites for learning *Rosetta*.

4.5.2 AutoLISP and JavaScript

AutoLISP and *JavaScript* implement the syntax and semantics of the corresponding PLs and provide the proper bindings. The main purpose of these frontends is to attract the large community of designers that learned and used these PLs in the past and to simplify their transition to *Rosetta*.

AutoLISP is one of the most used PLs for GD. However, this PL has some shortcomings that *Rosetta* overcomes. For example, one of the most frequent mistakes is to accidentally misspell the name of some variable or function. Because *AutoLISP* treats the use of undefined names as automatically bound to a default value, it will silently accept the mistake. Obviously, something will go wrong, but in general it will not be easy for the user to understand the cause of the error. In this regard, the syntax checker and the static debugger provided by *Rosetta* will immediately point out the cause of the error even before running the program. Other similar problems that are automatically (and statically) detected include syntax errors and a subset of type errors.

4.6 Shapes and Operations

Rosetta implements the majority of geometric shapes and operations supported by the most used CAD applications, namely, *AutoCAD* and *Rhinoceros3D*. However, *Rosetta* goes further: it overcomes the limitations of these CAD applications to provide shapes and operations with mathematical and geometric correctness. For example, *Rosetta* implements the empty and universal sets as special geometric shapes. Boolean operations automatically recognize and handle these shapes, implementing the identity and absorbing elements of the intersection, subtraction, and union, operations. Moreover, the empty and universal shapes are embedded in the language expressions, therefore, they are transparent to the designer who, in most cases, is unaware such shapes are being used. However, while the empty shape

actually produces the correct result in the CAD application (i.e., nothing), the universal shape cannot be represented. Nevertheless, apart from the mathematical correctness, visualizing this shape is irrelevant.

Operations are also functional, meaning that designers do not have to worry whether or not shapes are consumed by operations and GD programs do not need to be flooded with copy/delete commands. As a result, 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, ensuring that the final model reflects the intention of the designer, and all necessary copying/deletion is handled automatically according to the requirements of that 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.

Finally, 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. As an example, reconsider the symmetric difference in Figure 3.1. With functional operations, the implementation of this operator is straightforward (Figure 4.10). The result of this operator is exemplified in Figure 4.11.

$$\Delta(A, B) \Rightarrow (A \cup B) - (A \cap B)$$

```
(define (symmetric-difference a b)
  (subtract (union a b) (intersect a b)))
```

Figure 4.10: Symmetric difference: definition (on the left) and implementation (on the right)

4.7 Traceability

Rosetta implements traceability, therefore, it is possible to (1) point to program elements to identify the corresponding model elements; and (2) point to model elements to identify the corresponding program elements. Designers can use both approaches at the same time, moving from one to other as necessary, thus accelerating the development process.

Figure 4.12 illustrates a typical scenario where the user selects an expression in his program and *Rosetta* shows the set of shapes that resulted from that expression. Note that this set contains all shapes that were created by the expression during the complete execution of the program.

Figure 4.13 illustrates the converse scenario, in which a user selects an element of the model in the CAD application and *Rosetta* highlights the corresponding program elements.

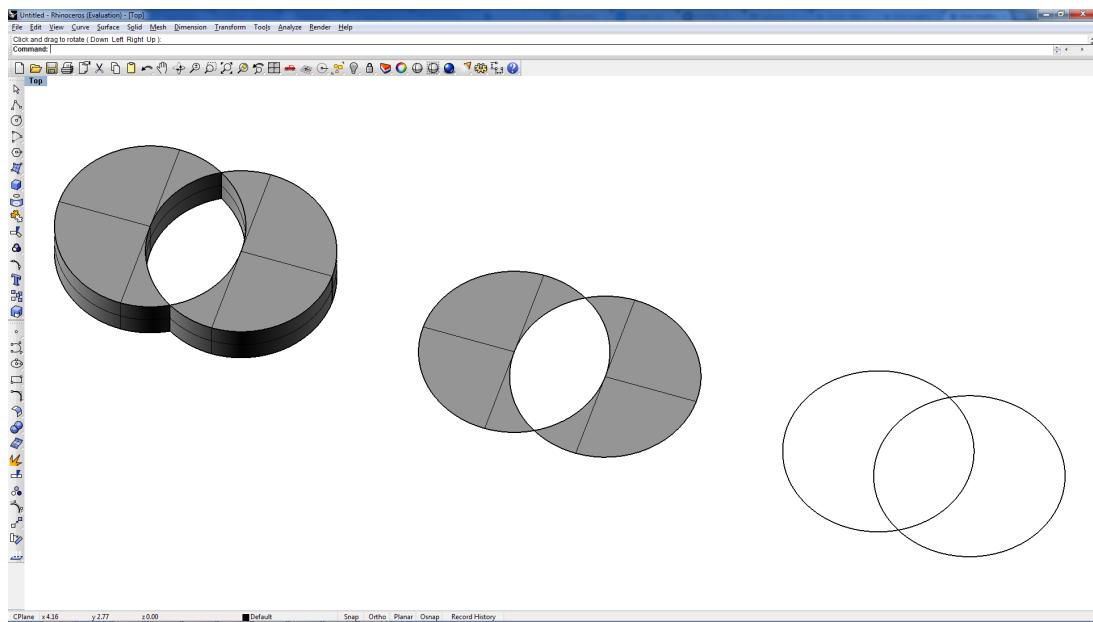


Figure 4.11: Dimension independent symmetric difference

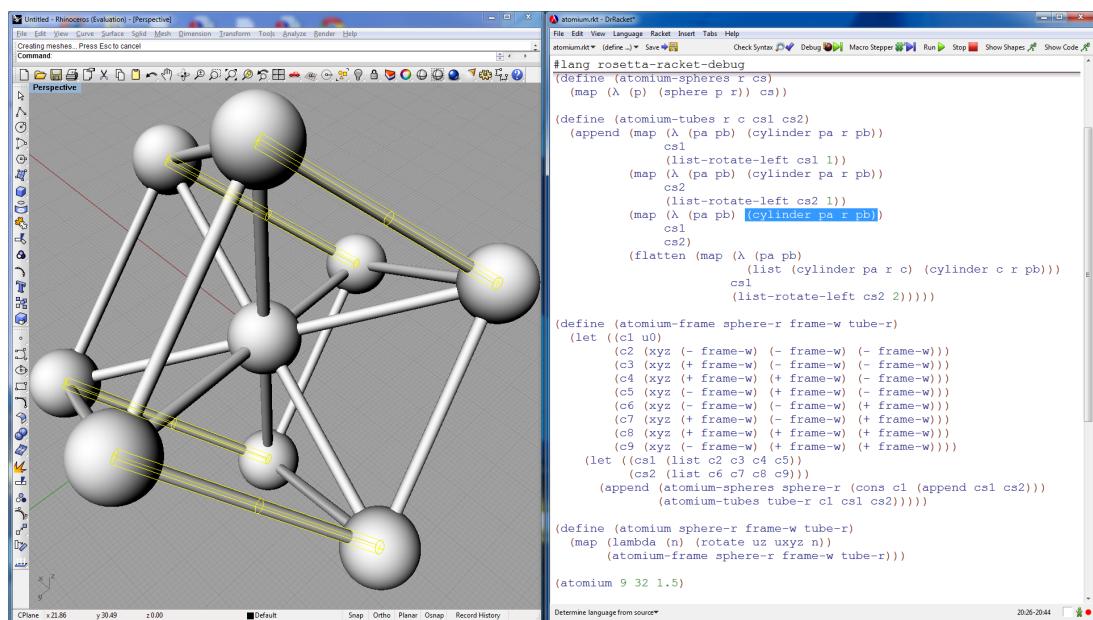


Figure 4.12: Relating program expressions to the generated shapes. The highlighted cylinders (on the left, in yellow) are generated by the highlighted program text (on the right, in blue)

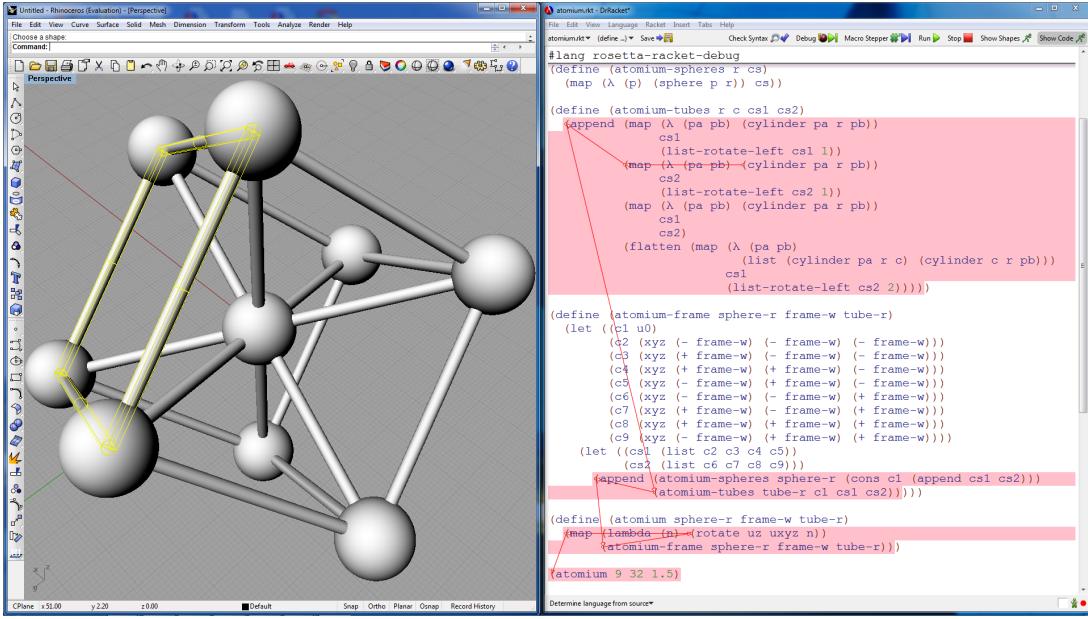


Figure 4.13: Relating shapes to the program expressions that generated them. The highlighted cylinders (on the left, in yellow) are generated by the highlighted program flow (on the right, using red arrows)

4.8 Visual Widgets

Similarly to *Grasshopper*, *Rosetta* provides sliders (Figure 4.14) 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.

Nevertheless, there are two approaches to minimize this problem, namely, (1) introduce parallel computations, and (2) reduce the time needed for each basic operation. Unfortunately, for GD tools that operate on top of classic CAD tools, such as, *Grasshopper* for *Rhinoceros3D* and *AutoLISP* for *AutoCAD*, both of these approaches are difficult to implement: the first, because most CAD tools are not thread-safe, and the second because most CAD tools were designed for the speed of human operation and not for the large volume of operations required by some GD programs. *Rosetta* improves this situation by providing the *OpenGL* backend, which sidesteps most of the functionality of traditional CAD applications and focuses mainly on rapid generation and visualization of the geometric model.

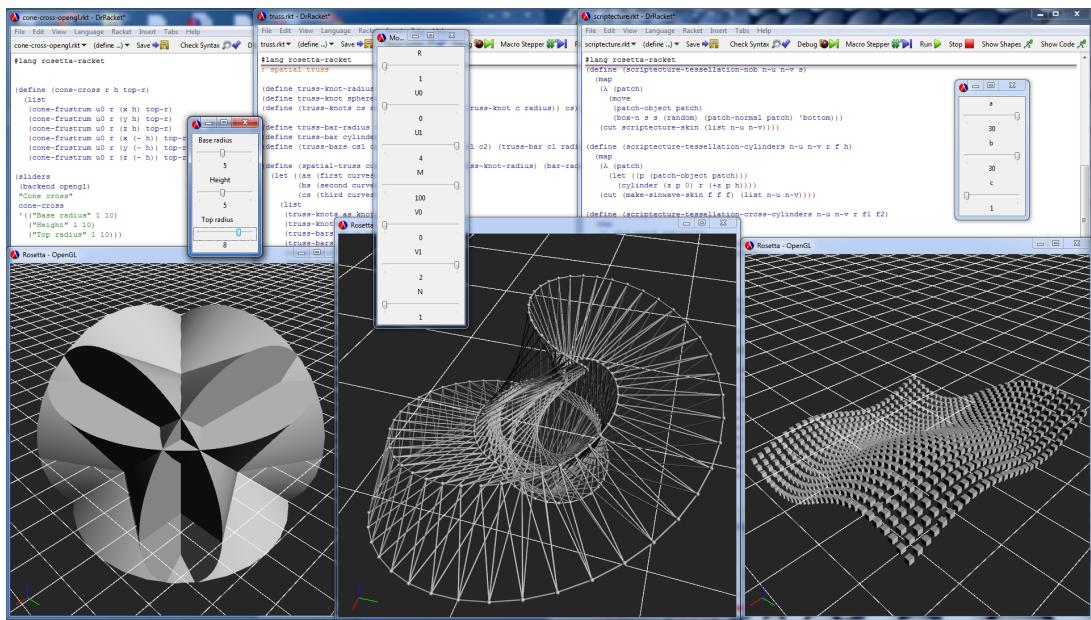


Figure 4.14: Using sliders and the *OpenGL* backend to interactively generate different models

Programming Paradigms & Techniques

5.1 Multi-Paradigm Programming Language

The previous part of this thesis argued that there are several design principles that a programming environment for GD must implement in order to be successful. One of these principles is the ability to provide multiple approaches to GD. This section shows how *Rosetta* can be used to explore different programming paradigms.

Rosetta provides *RosettaRacket* (Section 4.5.1), a multi-paradigm PL that is an extension to *Racket*. Even though *RosettaRacket* puts a strong emphasis on the functional paradigm, it does not restrict the use of other paradigms because there are many situations in which they are more suitable. For example, consider a GD program that has been executed. At this moment, the geometric models are already in the CAD application and the designer wants to experiment with the view and rendering parameters. In an imperative approach, the programmer merely has to invoke the procedures that control the view and rendering with different parameters, because these procedures do not modify the scene.

However, in a functional approach, these parameters must be specified at the same time the scene is created. In other words, when the programmer wants to modify a parameter, the entire scene must be recreated in the CAD application. Recreating large-scale or complex scenes requires significant time, making this experiment non-interactive. Therefore, in this case, the imperative approach is preferable.

GD tasks have different requirements and, for each task, there is a paradigm that is more suitable. To this end, *Rosetta* allows the programmer to choose from a variety of paradigms that include functional, imperative, object-oriented, and declarative. Moreover, several primitives have both imperative and functional versions. The following sections show examples of functional and imperative programming.

5.1.1 Example 1: Tessellators and Coordinate Generators

Trusses are pyramids made of joints and bars, and they adjust to and tessellate over straight and curved surfaces while maintaining strong structural properties. Trusses are a common element in GD with many applications, for example, in architecture and structural engineering (Figure 5.1).

Trusses can be represented, for example, by the set of joint coordinates. It is possible to generalize this representation to space frames (multiple trusses connected together) simply by extending this representation to multiple sets of coordinates. The following program implements space frames:

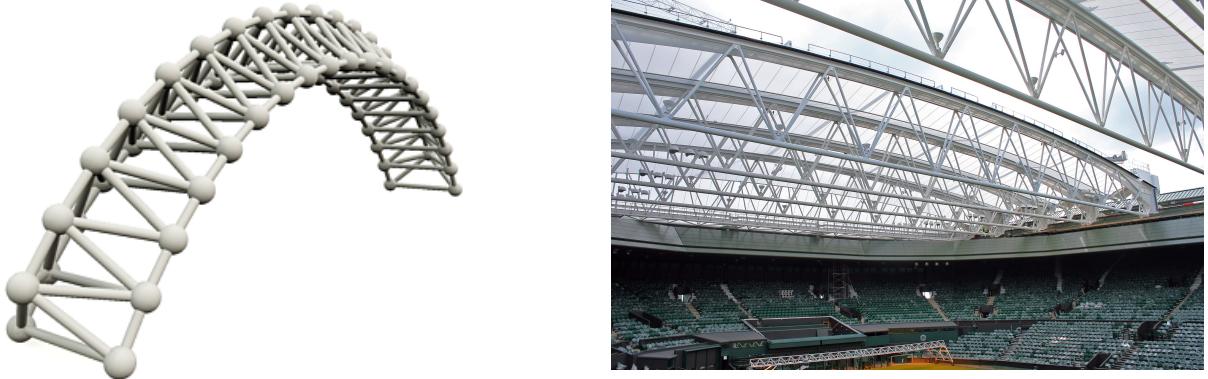


Figure 5.1: Space frames and the Wimbledon Stadium roof

```
(define (truss-knots cs radius)
  (map (lambda (c) (sphere c radius)) cs))

(define (truss-bars cs1 cs2 radius)
  (map (lambda (c1 c2) (cylinder c1 radius c2)) cs1 cs2))

(define (spatial-truss curves (knot-r truss-knot-radius) (bar-r truss-bar-radius))
  (let ((as (first curves))
        (bs (second curves))
        (cs (third curves)))
    (list
      (truss-knots as knot-r)
      (truss-knots bs knot-r)
      (truss-bars as cs bar-r)
      (truss-bars bs (drop-right as 1) bar-r)
      (truss-bars bs (drop-right cs 1) bar-r)
      (truss-bars bs (rest as) bar-r)
      (truss-bars bs (rest cs) bar-r)
      (truss-bars (rest as) (drop-right as 1) bar-r)
      (truss-bars (rest bs) (drop-right bs 1) bar-r)
      (if (empty? (cdr cs))
          (list
            (truss-knots cs knot-r)
            (truss-bars (rest cs) (drop-right cs 1) bar-r))
          (list
            (truss-bars bs (first (drop cs 3)) bar-r)
            (spatial-truss (drop cs 2) knot-r bar-r)))))))
```

The function `spatial-truss` creates space frames given any set of coordinates for the joints, meaning that it can create truss tessellations for any kind of surface. Note that this function is recursive, a feature that is typical of functional programming. To create the set of coordinates for the joints, consider, for example, an arc-shaped surface with coordinates given by the following program:

```
(define (arc-cs c r phi th1 th2 dth)
  (map
    (lambda (th) (+sph c r phi th))
    (: < th1 .. dth .. th2 <)))

(define (arc-surface-cs c ra rb phi th1 th2 e n)
```

```
(let ((dth (/ (- th2 th1) n)))
  (list
    (arc-cs (+pol c (/2 e) (+ phi pi/2)) ra phi th1 th2 dth)
    (arc-cs c rb phi (+ th1 (/2 dth)) (- th2 (/2 dth)) dth)
    (arc-cs (+pol c (/2 e) (- phi pi/2)) ra phi th1 th2 dth))))
```

The function `arc-cs` calculates the coordinates of an arc using spherical coordinates, which eliminates the need for trigonometric expressions and makes the code more elegant and concise. Similarly to *VisualScheme*, *Rosetta* implements the Cartesian, cylindrical, polar, and spherical coordinate systems. The function `arc-surface-cs` follows the same example, using the polar system to calculate the coordinates of an arc-shaped surface. This function can be combined with the function that creates trusses, `spatial-truss`, to produce an arc-shaped space frame.

However, the following program shows that combining these two functions results in all parameters being duplicated. It is possible to avoid this duplication by making the composition of functions `arc-surface-cs` and `spatial-truss` explicit. This is achieved using function composition, via `compose`, an operator seen in advanced functional PLs, such as, *PLaSM* and *Haskell*. Because composition might be considered too advanced for designers, *Rosetta* provides both mechanisms (Figure 5.2).

```
; implicit composition, with duplicated parameters
(define (arc-surface-truss c ra rb phi th1 th2 e n)
  (spatial-truss
    (arc-surface-cs c ra rb phi th1 th2 e n)))

; explicit composition via "compose"
(define arc-surface-truss
  (compose trusses arc-surface-cs))
```



Figure 5.2: Arc surface space frame

By separating functions that perform tessellations (tessellators), such as `spatial-truss`, from functions that calculate coordinates (coordinate generators), such as `arc-surface-cs`, it is possible to explore different combinations of tessellators and coordinate generators, and to reuse these functions in different contexts. For example, Figure 5.3 shows a program that reuses the truss tessellator to create a space frame along the Möbius band.

The function `spatial-truss-insert-apex` is a utility function that creates space frames without requiring the programmer to supply the coordinates of the truss apexes. Instead, it uses the function `insert-pyramid-apexes` to calculate the apexes for each patch of the surface according to the surface normal defined at the center of the patch. The function `moebius-cs` calculates the coordinates of the Möbius band using cylindrical coordinates, via `cyl`, and list comprehensions, via `enumerate-m-n`, an-

```
(define (spatial-truss-insert-apex cs)
  (let ((c1 (first (first cs)))
        (c2 (first (second cs)))
        (c4 (second (first cs))))
    (spatial-truss (insert-pyramid-apexes cs)))

(define (moebius-cs u1 u2 m v1 v2 n)
  (enumerate-m-n
   (lambda (u v)
     (cyl (* 4 (+ 1 (* v (cos (/2 u))))))
     u
     (* 4 (* v (sin (/2 u))))))
   u1 u2 m v1 v2 n))

(define moebius-truss
  (compose spatial-truss-insert-apex moebius-cs))
```

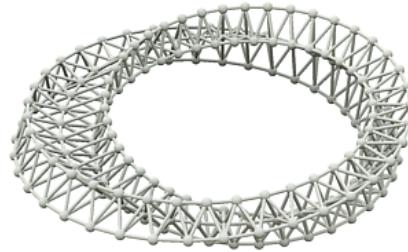


Figure 5.3: Möbius space frame

other feature of functional PLs. The function `moebius-truss` is the composition of the truss tessellator with the Möbius band coordinate generator.

Many GD tasks can be expressed using coordinate generators and tessellators, which suggests that this programming model can be considered a design pattern. While this programming model enables programmers to explore different combinations of tessellators and coordinates generators, creating geometry is only useful if it can be visualized. The following section describes an important visualization method called rendering.

5.1.2 Example 2: Automated Rendering

One of the most recurring tasks in GD is rendering geometry with a high level of realism. However, it is impractical for designers to configure rendering engines because, in most cases, there are dozens of parameters to control, for example, the ray-tracer, lighting, and shading properties. In order to overcome this problem, *Rosetta* provides a rendering script that simplifies and automates the rendering process. The script consists of a set of procedures that (1) establish rendering properties; (2) create the geometry to be rendered in the proper layers; and (3) control the rendering process.

As mentioned in Section 4.6, geometric shapes are composed in a scene graph and the time at which this scene graph is evaluated is not controlled by the designer. However, the rendering script must ensure that the shapes to be rendered are created before the rendering process starts, otherwise, the rendering result will be a blank image. To this end, the script forces the evaluation of the shapes using `evaluate`. For example, the following procedure creates the geometric shapes to be rendered in the proper layer:

```
(define (make-render-shapes node/nodes)
  (evaluate (layer shapes-layer node/nodes)))
```

Shadows increase the realism of the scene. But for shadows to be visible, there must be a surface on which they reflect. To this end, the script creates a reflective floor that spans under the entire scene, such that objects, when hit by sunlight, cast a shadow on that floor. The following procedure uses the bounding boxes of all objects in the scene to create the reflective floor at the appropriate location:

```
(define (make-render-floor w h col)
  (let ((z-min (reduce min
                        (map bbox-min-z
                             (map bbox (clone-shapes (get-shapes)))))))
    (evaluate
     (layer (floor-layer col) (rectangle-surface (z z-min) w h)))))
```

The function `bbox` returns the bounding box of a shape but it has the undesired side-effect of consuming (deleting) it. Therefore, shapes must be manually duplicated beforehand via `clone-shapes`. In most cases, the programming environment has a functional behavior, therefore, these side-effects are invisible. However, with `evaluate` the program becomes imperative and side-effects must be handled.

The following procedure defines lighting properties and controls the rendering process, combining the previous procedures for creating the geometry to be rendered and the reflective floor:

```
(define (render-shapes-floor node/nodes w h path type fl-w fl-h fl-col)
  (begin0
   (make-render-shapes node/nodes)
   (let ((floor (make-render-floor fl-w fl-h fl-col)))
     (set-sky-status! sky-status-background-and-illumination)
     (render-shapes w h path type)
     (delete-shape floor))))
```

Before the rendering process starts, via `render-shapes`, the reflective floor is created. Therefore, after the rendering process finishes, it is necessary to delete this floor, via `delete-shape`, otherwise, the user will see a shape in the selected backend he did not create. The manual/forced evaluation and deletion of the reflective floor is another example of imperative programming.

Finally, because the procedure `render-shapes-floor` requires a large number of parameters, the script provides a convenience procedure, `render-with-floor`, that defines default values for most parameters and allows these values to be overridden with keyword arguments or through a configuration file. The following program illustrates how to render the examples from the previous section, namely, the arc and Möbius space frames, using *AutoCAD* and default parameter values:

```
(define (render-examples)
  (render-with-floor (arc-trusses u0 10 9 0 (- pi/2) pi/2 1 20) "arc.png")
  (render-with-floor (moebius-trusses 0 (* pi 4) 80 0 0.3 1) "moebius.png"))
```

The use of the imperative paradigm to implement the rendering script requires few primitives from *Rosetta*. Other paradigms could have been used instead, but additional or more sophisticated primitives

would be necessary. A functional implementation of the rendering script would have to conceal or detach the side-effects of changing the rendering properties and creating and destructing the reflective floor. The following program is an example of a functional implementation of the same script:

```
(define (make-render-shapes node/nodes)
  (layer shapes-layer node/nodes))

(define (make-render-floor w h col)
  (let ((z-min (reduce min (map bbox-min-z (map bbox (get-shapes)))))))
    (layer (floor-layer col) (rectangle-surface (z z-min) w h)))

(define (render-shapes-floor node/nodes w h path type fl-w fl-h fl-col)
  (with-temporary-shape (make-render-floor fl-w fl-h fl-col)
    (parameterize ((sky-status sky-status-background-and-illumination))
      (render-shapes w h path type (make-render-shapes node/nodes)))))
```

Compared to the imperative version, the functional implementation is more concise, in part, because manual evaluation and deletion are no longer necessary. However, there are significant changes, namely, (1) a new primitive, `with-temporary-shape`, is necessary to conceal the side-effects of creating and destructing the reflective floor; (2) the procedure `set-sky-status!` is replaced by a dynamically scoped variable which is used together with `parameterize`; and (3) `render-shapes` must be a function, instead of an imperative procedure, and it must receive the geometric shapes.

The functional version might seem more adequate for designers because it is simpler and more concise. However, this script is part of the programming environment and it is intended that designers use it as a black-box. Therefore, the implementation details of the script will not affect the design workflow. Moreover, the imperative version is easier to implement in *Rosetta* because the primitives and semantic differences demanded by the functional version require additional implementation effort.

5.2 Programming Techniques

Rosetta provides multiple paradigms to address the broad spectrum of design task requirements. In addition to paradigms, there are several programming techniques that can dramatically simplify the programming effort, such as, higher-order and anonymous functions, monads, and non-deterministic programming. The following sections show how to use and combine these programming techniques.

5.2.1 Example 1: Combination of Programming Techniques

As mentioned before, in *Rosetta* it is possible to use different paradigms. In fact, it is even possible to combine them in the same program. For example, Figure 5.4 shows a program which places several cones inside a sphere. The function `sphere-of-cones` uses list comprehensions to iterate along the parametric dimensions of a sphere. Each pair of `th` and `phi` is used to calculate the base center of the cones placed by the function `cone-sph`, using spherical coordinates.

```
(define (cone-sph r h phi th)
  (cone (sph h phi th) r u0))

(define (sphere-of-cones r h n)
  (flatten
    (for/list ((th (: < 0 .. (/ 2pi n) .. pi >)))
      (for/list ((phi (: < 0 .. (/ 2pi n) .. 2pi >)))
        (cone-sph (+ r (* r (sin th))) h phi th))))
```

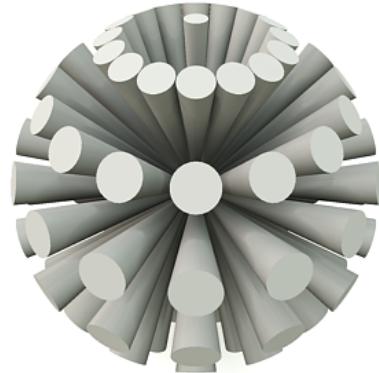


Figure 5.4: Sphere made of cones

Figure 5.5 shows a second program, which consists of a hollow sphere from which the cones calculated in the first program were subtracted. The function `pierced-sphere` uses the `for` cycle to iterate through the list of cones returned by `sphere-of-cones`. And it uses assignment (`set!`) to accumulate the subtractions in the variable `hollow-sphere`. After the last iteration, the variable `hollow-sphere` contains the hollow sphere pierced by several cones: the pierced sphere.

```
(define (pierced-sphere r e rc n)
  (let ((hollow-sphere
         (subtract (sphere r) (sphere (- r e)))))
    (cones
      (sphere-of-cones rc (* 1.1 r) n)))
    (for ((cone cones))
      (set! hollow-sphere
            (subtract hollow-sphere cone)))
    hollow-sphere))
```



Figure 5.5: Sphere pierced by a set of cones

While the first program is an example of functional paradigm, the second program is an example of the imperative paradigm. This shows that with *Rosetta* it is possible to use multiple paradigms and techniques, and combine them in the same program without additional effort.

5.2.2 Example 2: Higher-Order and Anonymous Functions

Programs are decomposed in structural and behavioral concerns. The majority of programs have static structure, which means that the set of all possible behavioral variations, determined by the program parameters, is finite. Therefore, all possible behavioral variations a program can contemplate must be either accounted *a priori* in the program structure or delegated to a parameter.

Designers need to experiment with different variations of a geometric scene without changing the program structure. Therefore, programs must be as parametric as possible so that variations can be easily achieved simply by changing the parameters of a program. Moreover, programs are constantly changing to address new requirements. Generalizing a program and making it more parametric can minimize the impact of such changes and, therefore, the implementation effort. For example, a program that creates a sphere can be generalized to create a given number of spheres that is determined by a parameter. In the case of *Grasshopper*, the number of spheres could be specified by a slider.

It is possible to further generalize this program to create a given number of any kind of shape. Unfortunately, in *Grasshopper*, this generalization cannot be implemented because of insufficient abstraction mechanisms. This limitation forces designers to manually modify the program structure when a new behavioral change is necessary. As a result, in most cases, what appears to be simple parametrizations of the same program is, in fact, a collection of similar programs that create almost identical shapes. Moreover, in most cases, these programs are created using *copy/paste*, meaning that they are redundant and, therefore, have maintenance problems. Using the right programming techniques, it is possible not only to avoid code duplication but also to write programs that can cope with behavioral changes without suffering structural modifications.

Modern PLs, such as, *Racket* and *Scala*, provide mechanisms for addressing behavioral changes, namely, (1) higher-order functions, which are functions that receive other functions as parameters and/or produce functions as result; and (2) anonymous functions, which are functions without a name. While higher-order functions allow programs to delegate behavior to a parameter, anonymous functions allow these parameters to be concisely defined in-place without polluting the program namespace.

As an example, consider that a designer wants to create a building but he does not know yet what kind of balconies the building should have. Instead, he wants first to create the building without balconies and, only afterward, experiment with different kinds of balconies. In *Rosetta*, the designer addresses this problem by implementing a high-order function that creates the building but delegates the responsibility of creating the actual balcony to a parameter, which is another function. The following program illustrates the higher-order function `balcony` that creates generic balconies, where `fn` is the parameter that decides the kind of balcony to create:

```
(define (balcony c fn x0 x1 dx ly slab-h guard-h handrail-l handrail-h bobs-d)
  (list
    (slab c fn x0 x1 bobs-d ly slab-h)
    (guard (+xyz c 0 handrail-l slab-h)
           fn x0 x1 dx ly guard-h handrail-l handrail-h bobs-d)))
```

This function is decomposed in two other functions. The first function, `slab`, extrudes the floor plan to create the floor:

```
(define (slab c fn x0 x1 dx ly lz)
  (let ((cs (coords-fn (+xy c (- x0) (- ly)) fn x0 x1 dx)))
    (let ((p0 (first cs))
          (p1 (last cs)))
      (extrude
        lz
        uz
        (surface
          (join
            (spline cs)
            (line p0 c (+x c (- x1 x0)) p1)))))))
```

The second function, `guard`, creates the handrail and bobs:

```
(define (handrail c fn x0 x1 dx ly l h)
  (sweep
    (spline (coords-fn (+xy c (- x0) (- ly)) fn x0 x1 dx))
    (surface (line-closed u0 (y l) (yz l h) (z h)))))

(define (bobs c fn x0 x1 dx ly h r)
  (for/list ((ci (coords-fn (+xy c (- x0) (- ly)) fn x0 x1 dx)))
            (cylinder ci r (+z ci h)))))

(define (guard c fn x0 x1 dx ly guard-h handrail-l handrail-h bobs-d)
  (list
    (handrail (+z c guard-h) fn x0 x1 bobs-d ly handrail-l handrail-h)
    (bobs c fn x0 x1 bobs-d ly guard-h (/2 handrail-l))))
```

With this approach, it is easy to create several variations of balconies simply by supplying a different function as parameter. As a first example, consider a balcony with a *saw teeth* variation:

```
(define (saw a b c x)
  (abs (+ a (* b (- x c)))))

(define (saw-balcony-example)
  (balcony
    u0
    (lambda (x) (saw 0 0.3 (* 3 pi) x))
    0 (* 4 pi) 0.5
    4 0.2 1 0.04 0.02 0.4))
```



Figure 5.6: Balcony with a saw teeth variation

Apart from higher-order functions, this example also uses anonymous functions, via `lambda`. With higher-order and anonymous functions, designers can parametrize behavior of GD programs, incorporating variations easily without changing the program. This shows the first relative advantage of *Rosetta* over GD languages, such as, *Grasshopper* and *RhinoScript*. The following programs use higher-order and anonymous functions to create other kinds of variations.

```
(define (oscillator a b c x)
  (* a (exp (- (* b x))) (sin (* c x)))))

(define (oscillator-balcony-example)
  (balcony
    (xy 8 8)
    (lambda (x) (oscillator -2 0.1 2 x))
    0 (* 4 pi) 0.2
    4 0.2 1 0.04 0.02 0.2))
```



Figure 5.7: Balcony with an oscillating variation

```
(define (sinusoidal a omega phi x)
  (* a (max -0.6 (min 0.6 (sin (+ (* omega x) phi))))))

(define (sinusoidal-balcony-example)
  (balcony
    (xy 16 16)
    (lambda (x) (sinusoidal 1 1 0 x))
    0 (* 4 pi) 0.2
    4 0.2 1 0.04 0.02 0.4))
```

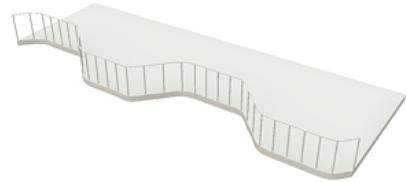


Figure 5.8: Balcony with a sinusoidal variation

5.2.3 Example 3: Monads

Writing programs that create *Lego* toys is not a recurring task in GD. However, several concepts used in *Lego* programs are, in fact, common in GD, namely, (1) world, where geometric shapes exist; (2) object and world coordinate systems; (3) absolute and relative coordinate systems; and (4) symmetry, gravity, and collision. For example, (1) a *Lego* toy exists in a world with a certain width and length (and possibly height) measured in bricks; (2) some bricks float and have an absolute position, other bricks are relative to the boundaries of the world or other bricks; and (3) some bricks are placed on top of other bricks.

A *Lego* program simply places *Lego* bricks in certain positions of the world. Representing a *Lego* world in a program is quite simple due to its discrete nature. However, defining the set of operations that manipulate it, in a concise and elegant fashion, can be quite challenging. A naive solution would be to define an imperative implementation of a procedure `lay!` that drops a brick on a specified position of the world. As an example of this implementation, consider the following program that creates a *Lego* world of size 10×10 and places two bricks of size $2 \times 2 \times 1$ on top of each other:

```
(define lego-world (make-lego-world 10 10))
(lay! lego-world 2x2x1 (xy 1 2))
(lay! lego-world 2x2x1 (xy 1 2))
```

From this example, it is clear that the *Lego* world variable must be manually passed to all functions that add bricks. Because *Lego* programs have a large number of bricks, eliminating one parameter can have a significant impact in program size, especially if that parameter is the same for most programs. Overcoming this problem by making the *Lego* world a global variable is not viable because the program becomes non-reentrant, which, in turn, could lead to conflicts between unrelated programs that use the *Lego* library. Instead, the imperative implementation can be replaced with a state monad. The following program illustrates the state monad applied to the previous program:

```
(lego
  (make-lego-world 10 10)
  (lay 2x2x1 (xy 1 2))
  (lay 2x2x1 (xy 1 2)))
```

An interesting consequence of the monadic implementation is that the resulting program is functional, even though it changes the state of the world. Figure 5.9 shows a complete example with 136 bricks. The original program has 136 commands, one for each piece. However, several functions were introduced and the simplified program has 36 commands for the same 136 pieces ($\approx 74\%$ reduction).



Figure 5.9: Giant panda *Lego* example

Some of these simplifications exploit relative coordinates and symmetries by means of several functions, namely, (1) `right-xy` and `back-xy` produce coordinates relative to the positive `x` and positive `y` boundaries of the world, respectively, and can be composed; (2) `right` and `back` change the center of the brick in its object coordinate system; (3) `with-world-cs` changes the location of the origin of

the world in a lexically enclosed scope; (4) `with-world-size` changes the apparent size of the world in a lexically enclosed scope; (5) `lay` drops bricks and is aware of gravity and collision, while `put` can place floating bricks; (6) `lay-x-sym` and `lay-y-sym` drop pairs of symmetrically positioned bricks, following the semantics of `right-xy` and `back-xy`, respectively.

5.2.4 Example 4: Non-Deterministic Programming

One of the most recurring tasks in GD is design exploration: the designer has a basic notion of the design he wants to achieve and, starting from a rough draft, he continues experimenting with different parameters until the concept has evolved into a solid idea. In most cases, there is not a concrete idea about the result, therefore, the path in which the design evolves is not always known beforehand.

It is possible to find a computational model that behaves similarly to the design exploration approach, namely, non-deterministic programming. Programs that make use of this programming model inherit automatically certain properties that facilitate design exploration, namely, the ability of achieving all, possibly infinite, modeling variations that satisfy a particular set of design requirements.

For example, consider that a designer wants to fill a spherical volume with a sequence of non-intersecting pipes (Figure 5.10). However, the designer does not know the actual path the pipes should take. What he knows is that the sequence of pipes cannot self-intersect or escape the spherical volume. Therefore, instead of writing a sophisticated search algorithm with backtracking, he can simply use non-deterministic programming and focus on implementing these design requirements.

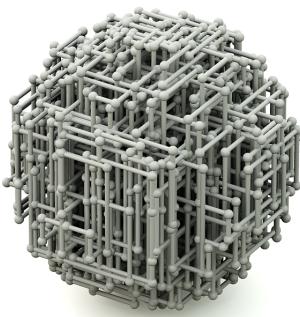


Figure 5.10: Tubes in sphere

Non-deterministic programming is based on (1) the ambiguous operator (`amb`), which receives several computations and chooses one of them non-deterministically to proceed (McCarthy, 1963); and (2) the fail operator (`fail`), which aborts the current computation and non-deterministically chooses another computation to proceed from the ones that were passed as argument to `amb`. In case there are no more computations left, the program aborts.

The following program computes a list of coordinates that make up the pipe path.

```
(define (explore c d n fn? visited-cs)
  (if (= n 0)
      (reverse (cons c visited-cs))
      (if (fn? c visited-cs)
          (explore (amb (neighbors c d)) d (- n 1) fn? (cons c visited-cs))
          (fail))))
  (define (connect-blocks c d r n fn?)
    (pipe (explore c d n fn? (list)) r)))
```

This program uses `neighbors` to generate the next coordinates on the path, which correspond to the possible computations (note the use of surrounding `amb`); and the parameter `fn?` is a function that implements the design requirements, such that if this predicate fails, the computation will `fail`. Finally, the `connect-blocks` is another example of the coordinate generators and tessellators design pattern, combining `explore`, which calculates the coordinates of the pipe path, with `pipe`, which creates a sequence of pipes and spheres given a list of coordinates. With this generic algorithm, it is easy to implement the design requirements for Figure 5.10 using, for example, an anonymous function (`lambda`):

```
(connect-blocks
  (xyz 0 0 0) 4 1 30
  (lambda (c visited-cs)
    (and (not (visited? c visited-cs)) (< (xyz-r c) 10))))
```

Using the coordinate generators and tessellators design pattern, and higher-order functions, it is simple to implement different variations, simply by replacing `pipe` with another tessellator or by replacing the `lambda` function with different design requirements (Figure 5.11, Figure 5.12, and Figure 5.13).

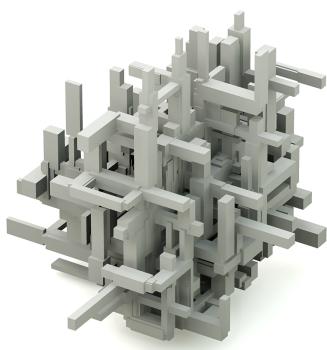


Figure 5.11: Blocks in sphere

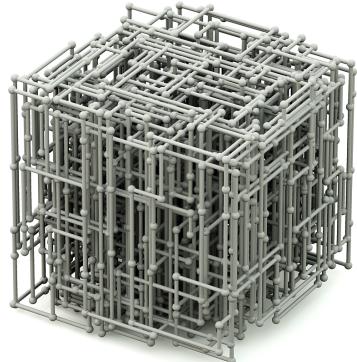


Figure 5.12: Tubes in cube

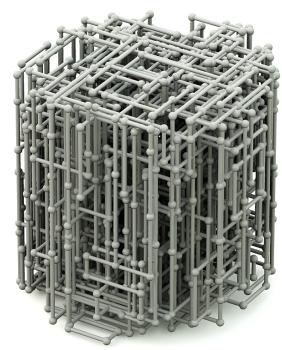


Figure 5.13: Tubes in cylinder

Multiple Frontends and Backends

6.1 Portability

Rosetta has several advantages over other GD languages, such as, the ability to provide multiple PLs and multiple CAD applications, the multitude of programming paradigms and rich linguistic features, and the ability to overcome several limitations of current CAD technology. These design decisions have a very strong impact, for example, in program portability. Even though there are millions of *AutoLISP* scripts on the Internet, they are useless outside the *AutoCAD* family unless they are rewritten in a different PL, which implies learning that PL, learning a new set of primitives, and possibly overcoming several of the limitations of either that PL or the respective CAD application. The same happens with *Rhinoceros3D* and the thousands of *RhinoScript* programs also available on the Internet. Because *Rosetta* supports several PLs as frontend, it is possible to reuse scripts written in different PLs and to use scripts across different PLs. For example, a program in *JavaScript* can use an *AutoLISP* script. Moreover, all frontends share the same set of primitives, therefore, designers can learn new PLs without having to worry about a new API, thus reducing the learning curve of that PL.

The same can be said about CAD applications. Traditionally, moving from one CAD application to another entails changing programming environments and learning a new PL which, in some cases, is not the PL the designer wanted. An effective change of CAD application forces designers to either discard or translate the incompatible programs, but either option is inadequate. However, with *Rosetta*, designers can change CAD applications and reuse their previous programs without additional modifications. This means that designers are no longer locked-in to a particular CAD application but it also has the advantage that users of different CAD applications can use the *Rosetta* program as a medium for exchanging models. As a result, they avoid the problematic conversion between the file formats supported by those particular CAD applications.

At first glance, it would be expected that in order to be a portable environment *Rosetta* could only provide the functionality that was supported by all backends. For example, *Rosetta* could only provide the sweep operation if both *AutoCAD* and *Rhinoceros3D* supported it. This is not entirely true because *Rosetta* emulates several functionality that is partially supported by CAD applications. For example, *Rosetta* is capable of performing Boolean operations of non-intersecting shapes in both *AutoCAD* and *Rhinoceros3D*, even though this functionality is not directly provided by *Rhinoceros3D* (Section 3.7). This also means that designers can use *Rosetta* only to overcome the limitations of their CAD application. In

the end, the functionality provided by *Rosetta*, which consists of the portable and emulated functionality, is more than enough for complete GD. In fact, it has been reported in several cases that it is faster to develop geometric complex programs in *Rosetta* than to manually create that geometry in a CAD application, showing that *Rosetta* can also be considered a tool for rapid prototyping. Moreover, *Rosetta* programs have the advantage of being parametric, while geometric models of CAD applications are not.

6.2 *TikZ*

As mentioned before, *Rosetta* can be extended with additional backends and frontends. In order to demonstrate this capability, a backend for *TikZ* (Section 2.2.9) was implemented. *TikZ* is a 2D graphics library for L^AT_EX. Even though CAD applications can render both 2D and 3D objects, *TikZ* is a better choice for rendering illustrations (Figure 6.1) for technical articles and research papers. Moreover, with an additional library, called *Sketch*, it is even possible to create 3D illustrations. *Rosetta* was designed to support multiple backends, therefore, it was very simple to add support for *TikZ*: this backend merely has to implement a set of procedures that create shapes and apply transformations. Moreover, programs that existed before this backend was created are automatically compatible without additional changes.

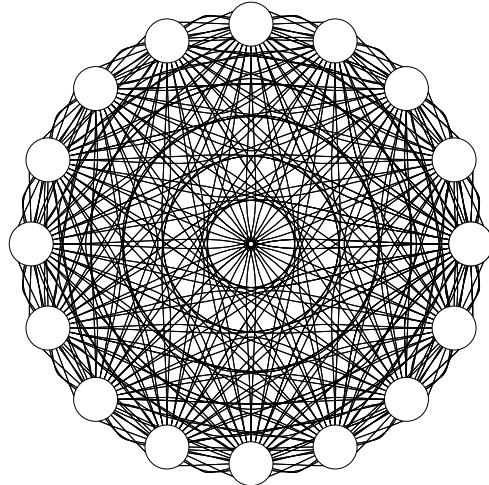


Figure 6.1: *TikZ* example

6.3 *RosettaFlow*

The frontends that are currently provided by *Rosetta* are all *TPLs*. However, *Rosetta* does not restrict the use of visual approaches or even the integration of external applications in the programming environment. In order to prove this point, a prototype of a *VPL* was implemented as a standalone application, using .NET. This *VPL*, called *RosettaFlow*, was inspired in *Grasshopper* and *GenerativeComponents*. Similarly to these *VPLs*, *RosettaFlow* programs consist of iconic elements, which represent geometric shapes

and transformations, associated by dataflow connectors. Because *RosettaFlow* is a standalone application, it is difficult to take advantage of the linguistic tools provided by *Racket*, such as the macro system. Therefore, *RosettaFlow* is responsible for (1) compiling the dataflow program into *Racket* code; (2) starting a separate process with the *Racket* environment; and (3) loading the compiled *Racket* code into this environment. Moreover, similarly to *Grasshopper* (Section 2.3.1), whenever designers drag a slider or change the program, the model is immediately updated. Figure 6.2 shows a *RosettaFlow* program that subtracts two spheres whose radii are controlled by sliders. Moving the second slider from left to right results in a larger portion of the spheres to intersect and a larger portion to be subtracted.

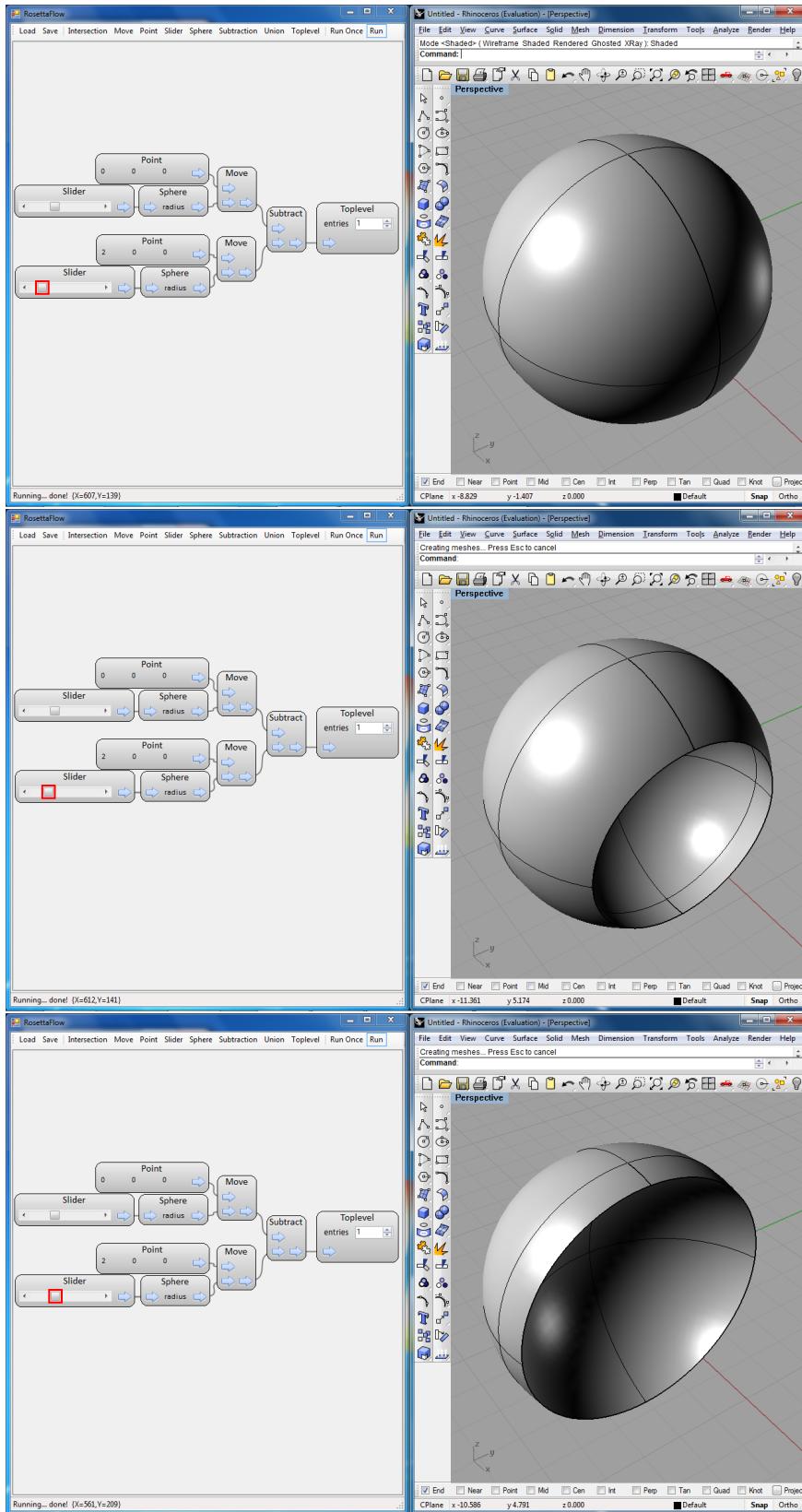


Figure 6.2: Changing sliders in *RosettaFlow* causes the geometric model to update in real-time

7 Practical Experiments

7.1 *TPLs* vs. *VPLs*

The previous sections presented an evaluation based on the development of GD programs and extension of the programming environment. This section completes this evaluation with practical experiments, namely, a comparison of *TPLs* and *VPLs*, and conversion and analysis of *AutoLISP* programs.

Rosetta is a direct descendant of *VisualScheme*, maintaining the same pedagogical concerns, but with a different approach: *Racket*, formerly called *Scheme*, is still used for the general audience of designers that do not have a background in Computer Science. However, additional PLs are included and multiple CAD applications are provided. An additional advantage is the automatic compatibility of the chosen language with the programming environment and the portability of the programs written in it.

Leitão et al. ([Leitão et al., 2012b](#)) compares *TPLs* and *VPLs* using *VisualScheme* and *Grasshopper* as representative of *TPLs* and *VPLs*, respectively. This comparison is based on a practical experiment to test program maintenance and adaptability consisting of two tasks, namely, (1) writing a program to create cylindrical towers (Figure 7.1); and (2) modifying that program to create conical towers with sinusoidal variations (Figure 7.2). Despite being one of the most popular GD languages, this comparison identifies several problems in *Grasshopper* and clearly shows the advantages of *VisualScheme* over that PL. Being a descendant of *VisualScheme*, *Rosetta* inherits these advantages.



Figure 7.1: First task of the experiment

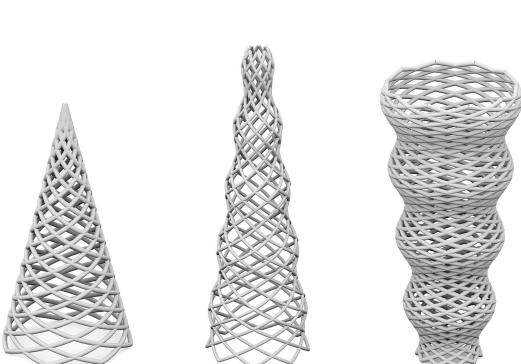


Figure 7.2: Second task of the experiment

There were several *Grasshopper* solutions for this experiment (Figure 7.3). In general, these solutions show redundancy resultant of extensive *copy/paste* and the amalgam of wire connections makes the code

difficult to understand, leading to future maintenance problems (Davis, Burry, & Burry, 2011).

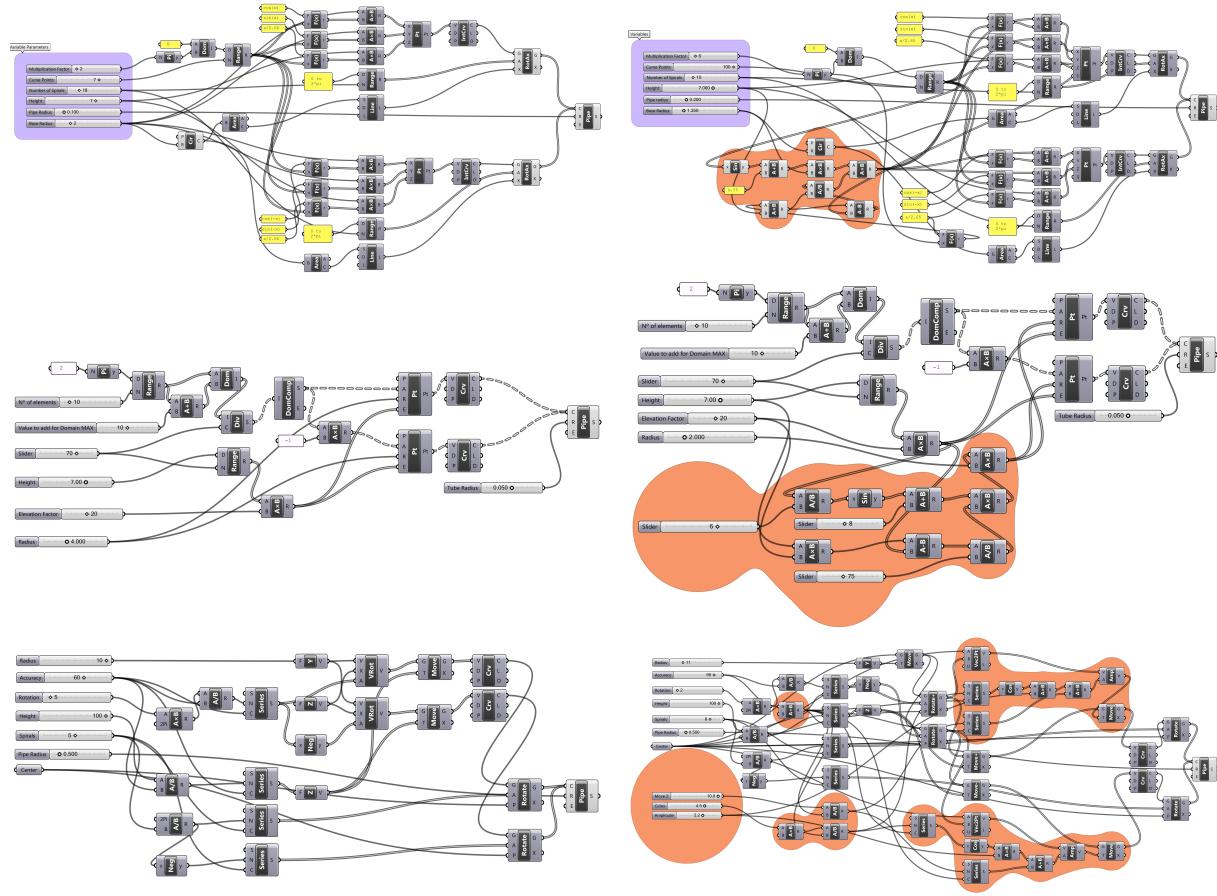


Figure 7.3: *Grasshopper* solutions for first (on the left) and second (on the right) tasks, with changes highlighted in orange

The *VisualScheme* solution for the first task (Figure 7.4) is more analytic and modular than the *Grasshopper* ones. One advantage of *VisualScheme*, and modern TPLs in general, is that concepts that are independent of particular problems, such as the linear variation, can be reused in different contexts. This is more difficult to do in *Grasshopper* because of the limited abstraction mechanisms. An important advantage of this approach is that each additional definition has an applicability that transcends the actual problem it is addressing, thus promoting code reuse. Higher-order functions also contribute to code maintainability by facilitating the implementation of alternative behaviors.

The *VisualScheme* solution for the second task (Figure 7.5) adds sinusoidal variations (`sinusoidal`), and linear composition of functions (`+fx`). The only part of the first solution that was actually modified was the function `spiral-points` to combine the `linear` and `sinusoidal` variations.

Compared to the *Grasshopper* solutions, the *VisualScheme* solution was adapted from the first to the second task with very little, localized modifications. In fact, it is still possible further improve the *VisualScheme* solution by delegating the variations in `spiral-points` to parameters so that new variations

```

(define (linear a b)
  (lambda (t) (+ a (* t (- b a)))))

(define (variation f n)
  (map f (range 1 n)))

(define (spiral-points r0 r1 phi0 phil h n)
  (map cyl
    (variation (linear r0 r1) n)
    (variation (linear phi0 phil) n)
    (variation (linear 0 h) n)))

(define (spirals r0 r1 h s t n)
  (map (lambda (phi)
    (spiral-points r0 r1 phi (+ phi (* 2 pi t)) h n))
    (variation (linear 0 (* 2 pi)) s)))

(define (spirals-cs r0 r1 h s t n)
  (append (spirals r0 r1 h s t n)
    (spirals r0 r1 h s (- t) n)))

(define (spirals-mesh css r)
  (map (lambda (cs) (sweep (spline cs) (circle r))) css))

```

Figure 7.4: *VisualScheme* solution for the first task of the experiment

```

(define (sinusoidal d omega)
  (lambda (t) (* d (sin (* 2 pi omega t)))))

(define (+fx f g)
  (lambda (x) (+ (f x) (g x)))))

(define (spiral-points r0 r1 phi0 phil h d omega n)
  (map cyl
    (variation (+fx (linear r0 r1) (sinusoidal d omega)) n)
    (variation (linear phi0 phil) n)
    (variation (linear 0 h) n)))

```

Figure 7.5: *VisualScheme* modifications for the second task of the experiment

can be added without even changing this function. Unfortunately, *Grasshopper* cannot do the same.

The *VisualScheme* solutions are compatible with *Rosetta*. But in *Rosetta*, it is still possible to further simplify these solutions using parametric elements (Section 3.3 and Section 4.6). Instead of calculating the coordinates for the spirals that make up the tower, the parametric functions of the spirals are used. The following program shows a possible *Rosetta* solution for the first task using parametric functions. Note that (1) the parameter *n* used to specify the number of coordinates for the splines was removed, (2) most functions produce other functions, or lists of functions, as result, and (3) the function *sweep* does not receive a spline as before, but a parametric function instead. In essence, the sampling strategy was removed, because it is automatically handled by *Rosetta*.

```
(define (linear a b)
  (lambda (x) (+ a (* x (- b a)))))

(define (spiral-points r0 r1 phi0 phil h)
  (lambda (x)
    (cyl
      ((linear r0 r1) x)
      ((linear phi0 phil) x)
      ((linear 0 h) x)))))

(define (spirals r0 r1 h s t)
  (for/list ((phi (: < 0 .. < s > .. 2pi >)))
    (lambda (x)
      ((spiral-points r0 r1 phi (+ phi (* 2pi t)) h) x)))))

(define (spirals-fns r0 r1 h s t)
  (append (spirals r0 r1 h s t)
          (spirals r0 r1 h s (- t)))))

(define (spirals-mesh fns r)
  (for/list ((fn fns))
    (sweep (function-curve fn) (circle r))))
```

7.2 Program Conversion and Analysis

In addition to this experiment, several *AutoLISP* projects have been converted to *Rosetta*. After the conversion, *Rosetta* detected several problems present in the original *AutoLISP* code that had not been detected. For example, in *AutoLISP*, local variables must be declared in the function parameter list. This is a tedious task and failure to do it results in other problems that are very difficult to detect. Unlike *AutoLISP*, in *Rosetta* local variables must always be declared beforehand, therefore, the static analysis detected at compile time several situations of undeclared variables in the converted programs.

The static analysis is also present in the *AutoLISP* frontend. There is an *AutoLISP* library used in the course *Programação e Computação para Arquitetura* at *Instituto Superior Técnico* that is supplied by the teaching staff for the students to use for the project of this course. Even though this library has been used for many years, the static analysis was still capable of detecting errors that students never found.

Another problem detected by the static analysis was name clash. It is very common for users working on the same project to choose the same names for variables or functions. *AutoLISP* does not help with collaborative development because (1) there is no concept of modular compilation unit that can be dissociated from the rest of the program; and (2) names used in variables and functions can be redefined, such that if two objects have the same name the second overwrites the first. *Rosetta* overcomes this problem with modules, and within a module there can be no two objects with the same name.

It was also detected that some *AutoLISP* programs were trying to emulate linguistic constructs that are not provided by this PL, such as, closures. Because *AutoLISP* implements dynamic scope, the use of anonymous functions can lead to problems called the upward and downward *funarg* problems. *Rosetta* implements lexical closures, therefore, these problems do not apply. *Rosetta* implements also macros that allow users to define new linguistic constructs even when the PL does not provide them. This means that with *Rosetta* users can write programs but also extend the PL they use to write those programs.

Finally, *Rosetta* supports large and complex projects. As an example, consider the *Turning Torso* (Figure 7.6), whose program has 452 lines of *RosettaRacket* code and makes extensive use of the functionality provided by *Rosetta*, namely, (1) shapes, such as, arc, cone frustum, cylinder, line, and spline; (2) Boolean operations, namely, intersection, subtraction, and union; (3) geometric transformations, such as, translation, rotation, join, loft, and sweep; (4) cylindrical and polar coordinate systems; (5) lists, as the main data structure; (6) linguistic features, namely, global variables, functions, list comprehensions, function mappings, anonymous functions, recursion and iteration, higher-order functions, and function currying; and (7) the coordinate generators and tessellators design pattern.



Figure 7.6: *Turning Torso* with different parametrizations

Apart from the *Turning Torso* and the figures that illustrate this thesis, several other GD programs were written to demonstrate and test the capabilities of *Rosetta*, some of which are shown in the following catalog (Figure 7.7 and Figure 7.8).



Figure 7.7: Catalog of *Rosetta* programs



Figure 7.8: Catalog of *Rosetta* programs (continuation)

8 Conclusions

8.1 Conclusions

Coding has always been present in the architectural design process, for example, in statutory, representation, and production codes. 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*, *C++*, *Java*, and *C#*, are inadequate because they are general purpose languages, providing few predefined abstractions for GD.

Moreover, the survey of the most used GD systems (Chapter 2) 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 correctness, performance, and 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 multiple CAD applications. Moreover, this thesis proposed a set of design principles (Chapter 3), 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 (Chapter 4) is modern programming environment designed to overcome the problems of current GD systems. To this end, *Rosetta* provides (1) multiple frontend 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 (Chapter 5), 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 (Chapter 5), 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.

Currently provided backends include *AutoCAD*, *Rhinoceros3D*, and *OpenGL*. While *AutoCAD* and *Rhinoceros3D* provide enough functionality for CAD programming, *OpenGL* was added for fast rendering and immediate feedback.

Currently provided frontends include *AutoLISP*, *JavaScript*, and *RosettaRacket*. They virtualize the syntax of these PLs and use the macro system to compile programs written in these PLs to a compatible environment in which they can interoperate. Moreover, this approach has the additional advantage that these frontends can take advantage of the syntax checker and static debugger. For example, the *AutoLISP* frontend can detect undeclared names, even though the original *AutoLISP* cannot.

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. Immediate feedback is provided by 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 (Chapter 5). 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 *TikZ* backend and the *RosettaFlow* frontend were implemented (Chapter 6). Finally, this evaluation was completed with practical experiments, namely, a comparison of *TPLs* and *VPLs*, and conversion and analysis of *AutoLISP* programs, including large and complex projects, namely, the *Turning Torso*.

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.

8.2 Current and Future Work

At the moment, there are already students using *Rosetta* as a research platform for their M.Sc. theses. For example, one student is linking *Rosetta* to CGAL ([Fabri, Giezeman, Kettner, Schirra, & Schönherr, 1996](#)) in order to take advantage of the exact computation paradigm, thus avoiding the errors that result from the limited precision numerical representations used in CAD applications. CGAL provides also several interesting features including specialized data structures and geometric algorithms.

Another student is exploring fully detailed projects that model both macro objects, such as, walls and windows, and micro objects, such as, door knobs, in the same project.

Moreover, *Rosetta* will also be used to teach programming to architecture students in the course *Programação e Computação para Arquitectura* at *Instituto Superior Técnico*, replacing the *AutoLISP* library that has been used so far.

Regarding the future development of *Rosetta*, there are several possible paths, such as, (1) explore a full parametric approach to GD, in which all objects are parametric instead of geometric; (2) link *Rosetta* with a geometric kernel, such as *ACIS*, which implements several geometric algorithms with high performance; (3) link *Rosetta* with a ray-tracer to have finer control over rendering and minimize dependency on CAD applications; (4) design a syntax for a new frontend to further simplify GD programming; and (5) explore other adaptive sampling strategies for parametric elements.

Bibliography

- Aiken, A., Williams, J., & Wimmers, E. (1993, September). *The FL project: The design of a functional language* (Tech. Rep.). San Jose, CA, USA: IBM Almaden Research Center.
- Aish, R., & Woodbury, R. (2005). Multi-level interaction in parametric design. In A. Butz, B. Fisher, A. Krüger, & P. Olivier (Eds.), *Smart Graphics, 5th international symposium, SG 2005, Frauenwörth cloister, Germany, August 22-24, 2005, Proceedings* (Vol. 3638, p. 151-162). Berlin, Germany: Springer Verlag.
- Autodesk. (2006). *3Ds Max MAXScript essentials* (Second ed.). Elsevier Focal Press.
- Backus, J., Williams, J., Wimmers, E., Lucas, P., & Aiken, A. (1989, October). *FL language manual: Parts 1 and 2* (Tech. Rep. No. IBM Research Report RJ 7100). San Jose, CA, USA: IBM Almaden Research Center.
- Berman, M. (1994, February). Does Scheme enhance an introductory programming course?: Some preliminary empirical results. *SIGPLAN Notices*, 29(2), 44-48.
- Bicalho, A., & Feltman, S. (2000). *Mastering MAXScript and the SDK for 3D studio max*. USA: Sybex.
- Cabecinhas, F. (2010). *A high-level pedagogical 3D modeling language and framework*. Unpublished master's thesis, Instituto Superior Técnico (IST), Technical University of Lisbon (UTL), Lisboa, Portugal.
- Chandler, R. (1990). A recursive technique for rendering parametric curves. *Computers & Graphics*, 14(3/4), 477-479.
- Chen, N. (1992). High school computing: The inside story. *The Computing Teacher*, 19(8), 51-52.
- Chok, K. (2011, October). Progressive spheres of innovation: Efficiency, communication and collaboration. In J. Johnson, J. Taron, V. Parlac, & B. Kolarevic (Eds.), *Acadia 11: Integration through computation [Proceedings of the 31st annual conference of the Association for Computer Aided Design in Architecture (ACADIA)]* (p. 234-241). DE, USA: Association for Computer Aided Design in Architecture.
- Cunningham, S., & Bailey, M. (2001, February). Lessons from scene graphs: Using scene graphs to teach hierarchical modeling. *Computers & Graphics*, 25(4), 703-711.

- Davis, D., Burry, J., & Burry, M. (2011). Untangling parametric schemata: Enhancing collaboration through modular programming. In P. Leclercq, A. Heylighen, & G. Martin (Eds.), *CAAD futures 2011: Designing together* (p. 55-78). Liège, Belgium: Les Editions de l'Université de Liège, Liège, Belgium.
- Deursen, A. van, Klint, P., & Visser, J. (2000, June). Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6), 26-36.
- Fabri, A., Giezeman, G.-J., Kettner, L., Schirra, S., & Schönherr, S. (1996, May). The CGAL kernel: A basis for geometric computation. In M. Lin & D. Manocha (Eds.), *Applied computational geometry: Towards geometric engineering Proceedings (WACG'96), Philadelphia, May, 27th-28th* (p. 191-202). Berlin, Germany: Springer-Verlag.
- Felleisen, M., Findler, R., Flatt, M., & Krishnamurthi, S. (2004a, July). The structure and interpretation of the Computer Science curriculum. *Journal of Functional Programming*, 14(4), 365-378.
- Felleisen, M., Findler, R., Flatt, M., & Krishnamurthi, S. (2004b). The TeachScheme! project: Computing and programming for every student. *Computer Science Education*, 14(1), 55-77.
- Hudak, P. (1998). Domain specific languages. In *Handbook of programming languages, Volume III: Little languages and tools* (p. 39-60). Indianapolis, IN, USA: MacMillan.
- Jones, R., & Lins, R. (1996). *Garbage collection: Algorithms for automatic dynamic memory management*. England, UK: Wiley.
- 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., Cabecinhas, F., & Martins, S. (2010, September). Revisiting the architecture curriculum: The programming perspective. In G. Schmitt, L. Hovestadt, & L. Van Gool (Eds.), *Future cities: Proceedings of the 28th conference on Education in Computer Aided Architectural Design in Europe* (p. 81-88). Zurich, Switzerland: vdf Hochschulverlag ETH Zurich.

- Leitão, A., & Santos, L. (2011, September). Programming languages for generative design: Visual or textual? In *Respecting fragile places: Proceedings of the 29th conference on Education in Computer Aided Architectural Design in Europe* (p. 549-557). Brussels: eCAADe (Education and Research in Computer Aided Architectural Design in Europe) and UNI Ljubljana, Faculty of Architecture.
- Leitão, A., Santos, L., & Lopes, J. (2012a, September). Collaborative digital design. In *Digital physicality — physical digitality [proceedings of the 30th conference on education in Computer Aided Architectural Design in Europe]*.
- Leitão, A., Santos, L., & Lopes, J. (2012b, March). Programming languages for generative design: A comparative study. *International Journal of Architectural Computing*, 10(1), 139-162.
- Lopes, J., & Leitão, A. (2011a). Essential language features for generative design. In *Iii simpósio de informática (inforum 2011)*. Coimbra, Portugal: Departamento de Engenharia Informática da Universidade de Coimbra.
- Lopes, J., & Leitão, A. (2011b, October). Portable generative design for CAD applications. In J. Johnson, J. Taron, V. Parlac, & B. Kolarevic (Eds.), *Acadia 11: Integration through computation [proceedings of the 31st annual conference of the association for computer aided design in architecture (acadia)]* (p. 196-203). DE, USA: Association for Computer Aided Design in Architecture.
- Maeda, J. (1996). *Design by numbers*. Cambridge, MA, USA: MIT Press.
- Marceau, G., Fisler, K., & Krishnamurthi, S. (2011). Measuring the effectiveness of error messages designed for novice programmers. In T. Cortina, E. Walker, L. King, & D. Musicant (Eds.), *Proceedings of the 42nd ACM technical symposium on Computer Science education* (p. 499-504). New York, NY, USA: ACM.
- McCarthy, J. (1963). A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*, 33-70.
- Menges, A. (2006). Instrumental geometry. *Architectural Design*, 76(2), 42-53.
- Miller, N. (2011). The Hangzhou tennis center: A case study in integrated parametric design. In J. Cheon, S. Hardy, & T. Hemsath (Eds.), *Proceedings of the 2011 Association for Computer Aided Design in Architecture (ACADIA) Regional conference*. DE, USA: Association for Computer Aided Design in Architecture.
- Moses, J. (1970, July). The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem. *SIGSAM Bulletin*(15), 13-27.
- Müller, P., Wonka, P., Haegler, S., Ulmer, A., & Gool, L. V. (2006). Procedural modeling of buildings. In *ACM SIGGRAPH 2006 papers* (p. 614-623). New York, NY, USA: ACM.

- Myers, B. (1990, March). Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1), 97-123.
- Paoluzzi, A., & Pascucci, V. (2003). *Geometric programming for computer aided design*. England, UK: Wiley.
- Paoluzzi, A., & Sansoni, C. (1992). Programming language for solid variational geometry. *Computer-Aided Design*, 24(7), 349-366.
- Penttilä, H. (2003, January). Architectural-IT and educational curriculums: An European overview. *International Journal of Architectural Computing*, 1(1), 102-111.
- Rawls, R., & Hagen, M. (1998). *AutoLISP programming: Principles and techniques*. IL, USA: Goodheart-Willcox.
- Reas, C., & Fry, B. (2010). *Getting started with Processing*. O'Reilly.
- Rocker, I. (2006). When code matters. *Architectural Design*, 76(4), 16-25.
- Rossum, G. van, & Drake, F. (2003). *The Python language reference manual*. UK: Network Theory Limited.
- Rutten, D. (2007). *Rhinoscript™101 for Rhinoceros 4.0*. Seattle, WA, USA: Robert McNeel Associates.
- Shea, K., Aish, R., & Gourtovaia, M. (2005, March). Towards integrated performance-driven generative design tools. *Automation in Construction*, 14(2), 253-264.
- Stouffs, R., & Chang, W.-T. (2010, July). Representational programming for design analysis. In W. Tizani (Ed.), *Computing in Civil and Building Engineering: Proceedings of the international conference* (p. 351-359). Nottingham, UK: Nottingham University Press.
- Sutherland, I. (1963). Sketchpad: A man-machine graphical communication system. In E. C. Johnson (Ed.), *Proceedings of the May 21-23, 1963, spring joint computer conference* (Vol. 23, p. 329-346). Baltimore, MD, USA: Spartan Books Inc.
- Terzidis, K. (2003). *Expressive form: A conceptual approach to computational design*. London, UK and New York, NY, USA: Spon Press.
- Watson, A. (2009). *GDL handbook: A comprehensive guide to creating powerful ArchiCAD objects*. New Zealand: Cadimage Solutions.
- Wilkins, M., Kazmier, C., & Osterburg, S. (2005). *MEL scripting for Maya animators*. Morgan Kaufmann.

Author Index

- Aiken, A., 8, 73
Aish, R., 1, 16, 73, 76
Autodesk, 7, 73

Backus, J., 8, 73
Bailey, M., 9, 73
Berman, M., 11, 73
Bicalho, A., 73
Burry, J., 62, 74
Burry, M., 62, 74

Cabecinhas, F., 5, 8, 10, 11, 73, 74
Chandler, R., 39, 73
Chang, W.-T., 14, 76
Chen, N., 11, 73
Chok, K., 14, 73
Coutinho, F., 1, 74
Cunningham, S., 9, 73

Davis, D., 62, 74
Deursen, A. van, 2, 74
Drake, F., 9, 76
Duarte, J. P., 1, 74

Fabri, A., 71, 74
Felleisen, M., 11, 74
Feltman, S., 73
Findler, R., 11, 74
Fisler, K., 11, 75
Flatt, M., 11, 74
Fry, B., 9, 76
Giezeman, G.-J., 71, 74

Gool, L. V., 16, 75
Gourtovaia, M., 1, 76

Haegler, S., 16, 75
Hagen, M., 76
Hudak, P., 2, 74

Jones, R., 26, 74

Kazmier, C., 16, 76
Kettner, L., 71, 74
Killian, A., 1, 74
Klint, P., 2, 74
Kolarevic, B., 1, 74
Krüger, M., 1, 74
Krause, J., 1, 74
Krishnamurthi, S., 11, 74, 75

Leitão, A., XIII, XV, 11, 14, 61,
 74, 75
Lins, R., 26, 74
Lopes, J., XIII, XV, 75
Lucas, P., 8, 73

Maeda, J., 1, 75
Marceau, G., 11, 75
Martins, S., 11, 74
McCarthy, J., 54, 75
Menges, A., 16, 75
Miller, N., 14, 75
Moses, J., 5, 75
Müller, P., 16, 75
Myers, B., 12, 76

Osterburg, S., 16, 76

Paoluzzi, A., 8, 76
Pascucci, V., 76
Penttilä, H., 12, 76

Rawls, R., 76
Reas, C., 9, 76
Rocker, I., 1, 76
Rossum, G. van, 9, 76
Rutten, D., 6, 76

Sansoni, C., 8, 76
Santos, L., XIII, 14, 75
Schirra, S., 71, 74
Schönherr, S., 71, 74
Shea, K., 1, 76
Stouffs, R., 14, 76
Sutherland, I., 23, 76

Terzidis, K., 1, 76

Ulmer, A., 16, 75

Visser, J., 2, 74

Watson, A., 7, 76
Wilkins, M., 16, 76
Williams, J., 8, 73
Wimmers, E., 8, 73
Wonka, P., 16, 75
Woodbury, R., 16, 73



*RosettaLang*¹ is a website created to support *Rosetta*, providing documentation, tutorials, examples, and a mailist list (Figure A.1 and Figure A.2).

The screenshot shows the Google Code project page for 'rosetta-lang'. The header includes the project logo (a blue icon with a white 'P'), the name 'rosetta-lang', and a subtitle 'Rosetta, a visual programming language for DrRacket'. Below the header is a navigation bar with links: Project Home (highlighted), Downloads, Wiki, Issues, Source, Summary, Updates, and People. The main content area has a sidebar on the left containing 'Project Information' such as 'Starred by 1 user', 'Activity' (High), 'Project feeds', 'Code license' (New BSD License), 'Labels' (Student, Racket, Design), and 'Members' (jabolo...@gmail.com, antonio...@gmail.com). The main content area features three examples: 'Turning Torso' (a tall, curved, perforated cylinder), 'Conical sphere' (a spherical shape composed of many small cylinders radiating from a central point), and 'Crater Tessellation' (a curved surface with a grid of circular indentations).

Figure A.1: *RosettaLang* catalog

¹<http://code.google.com/p/rosetta-lang>

 **rosetta-lang**
Rosetta, a visual programming language for DrRacket

[Project Home](#) [Downloads](#) [Wiki](#) [Issues](#) [Source](#)

Search Current pages ▾ for

★ Shelter Updated Oct 9 (4 days ago) by [jabolo...@gmail.com](#)



Documentation

```
(shelter w l h nx ny nz r) + any/c
w : (and/c real? positive?)
l : (and/c real? positive?)
h : (and/c real? positive?)
nx : (and/c integer? positive?)
ny : (and/c integer? positive?)
nz : (and/c integer? positive?)
r : any/c
```

Subtracts a sphere of radius r from a set of tubes in a grid layout. The grid of tube has width w , length l and height h . The parameters nx , ny and nz specify the number of tubes in the grid in the directions of x , y and z , respectively.

Definitions

```
#lang racket
(require rosetta)
(provide shelter)

(define (cut-pipe x y z p r 1)
  (subtract
    (move (xyz x y z) p)
    (move (p^n uy (- 1)) (sphere r)))))

(define (shelter w l h nx ny nz r)
  (let* ((p-r (/ w nx 2))
         (p-1 (/ l ny))
         (p (rotate (/ pi 2) ux
                    (pipe (- p-r 0.1) p-r p-1))))
        (for/list ((zi (range nz)))
          (for/list ((yi (range ny)))
            (for/list ((xi (range nx)))
              (let ((x (- (+ (* p-r 2) xi) p-r) (/ w z)))
                (y (* p-1 yi))
                (z (+ (* p-r 2) zi) p-r)))))))
    (cut-pipe x y z p r 1))))
```

Figure A.2: *RosettaLang* documentation

Index

- .NET, 9, 58
3ds Max, 7, 17

ACIS, 71
ArchiCAD, 7, 17
AutoCAD, 2, 5, 6, 9, 17, 20, 23,
 27, 28, 31, 32, 34, 36, 38,
 41, 47, 57, 70
AutoLISP, 2, 5, 6, 23, 24, 31, 37,
 38, 41, 57, 61, 64, 65,
 69–71

BASIC, 7
Blender, 9, 17

C, 2, 6–9, 12, 69
C++, 2, 8, 9, 69
C#, 2, 16, 24, 69
CGA, 14, 16, 17
CGAL, 71
CityEngine, 16
Common LISP, 6
CPAN, 24

DrRacket, 33, 37

FL, 8
Fortran, 6

GDL, 2, 7, 23, 69

GenerativeComponents, 5, 14,
 16, 23, 58, 69
Grasshopper, 2, 5, 12, 14–16, 23,
 27–29, 41, 50, 51, 58, 59,
 61, 62, 64, 69
Haskell, 45
Hypergraph, 14, 16
Java, 2, 6, 8, 9, 24, 69
JavaScript, 31, 32, 37, 38, 57, 70
JIT, 33
Lego, 52, 53
LISP, 5–7
MAXScript, 7, 8
Maya, 16, 17
MEL, 16
OpenGL, 7, 9, 10, 34–37, 41, 42,
 70
Pascal, 6
Perl, 9, 24
PLaSM, 5, 8, 9, 17, 23, 45
POV-Ray, 10
Processing, 9
Python, 9, 10, 16, 20
Racket, 33, 37, 38, 43, 50, 59, 61

Rhinoceros3D, 2, 6, 9, 14, 17, 23,
 27, 28, 31, 34–36, 38, 41,
 57, 70
RhinoScript, 2, 5, 6, 23, 24, 31, 51,
 57, 69
Rosetta, 2, 3, 31–34, 36–39, 41, 43,
 45–51, 57, 58, 61, 64–67,
 70, 71, 77
RosettaFlow, 58–60, 71
RosettaLang, 77, 78
RosettaRacket, 34, 35, 37, 38, 43,
 65, 70

Scala, 50
Scheme, 9, 11, 12, 33, 61
SDL, 10

TikZ, 10, 11, 17, 21, 58, 71
TPL, 5, 12, 14, 16, 23, 58, 61, 62,
 69, 71
Turning Torso, 65, 71

VBScript, 6
VisualBasic, 7
VisualBasic .NET, 7
VisualLISP, 5
VisualScheme, 2, 11, 12, 45,
 61–64
VPL, 12, 14, 16, 23, 58, 61, 69, 71

