Programming Languages For Generative Design: A Comparative Study

António Leitão, Luís Santos, and José Lopes

Programming Languages For Generative Design: A Comparative Study

António Leitão, Luís Santos, and José Lopes

Abstract

In the field of Generative Design (GD), Visual Programming Languages (VPLs), such as Grasshopper, are becoming increasingly popular compared to the traditional Textual Programming Languages (TPLs) provided by CAD applications, such as RhinoScript. This reaction is explained by the relative obsolescence of these TPLs and the faster learning curve of VPLs. However, modern TPLs offer a variety of linguistic features designed to overcome the limitations of traditional TPLs, making them hypothetical competitors to VPLs. In this paper, we reconsider the role of TPLs in the design process and we present a comparative study of VPLs and modern TPLs. Our findings show that modern TPLs can be more productive than VPLs, especially, for large-scale and complex design tasks. Finally, we identify some problems of modern TPLs related to portability and sharing of programs and we propose a solution.

1. INTRODUCTION

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 Alberti's De re aedificatoria [1].

Computers popularized and extended the notion of coding in architecture [2] 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 [3]. 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 [4].

Computational design methods allow automation of the design process and extension of the standard features of CAD applications [5], thus transcending their limitations [6]. Therefore, CAD software shifts from a representation tool to a medium for algorithmic computation, from which architecture can emerge. To apply computational methods, one must first translate the thought process into a computer program by means of a Programming Language (PL).

This paper discusses the most used PLs for Generative Design (GD), dividing them in two groups: Visual Programming Languages (VPLs) and Textual Programming Languages (TPLs). VPLs and TPLs are defined, compared and analyzed in terms of their advantages and fitness to the GD domain. We consider GenerativeComponents for MicroStation, Hypergraph for Maya, and Grasshopper for Rhinoceros3D as representative of VPLs. We consider RhinoScript, Haskell, Python, and Scheme as representatives of TPLs.

Given that comparing state-of-the-art VPLs, such as Grasshopper, with old general purpose TPLs, such as RhinoScript, is inadequate, we choose Grasshopper and VisualScheme [7] for a comparative study. VisualScheme is a programming environment for AutoCAD that uses Scheme for pedagogical reasons. In spite of its qualities, Scheme is relatively unknown in the GD community, which motivated us to develop Rosetta, a descendant of VisualScheme that allows the use of additional languages, such as AutoLISP and JavaScript, and additional CAD tools, such as Rhinoceros3D.

1.1. Overview

Sections 2, 3 and 4 compare TPLs and VPLs from different perspectives: (1) comparative examples showing different programming approaches to GD and theoretical differences between TPLs and VPLs; (2) a practical experiment using Grasshopper and VisualScheme users; and (3) an evaluation of the linguistic dimensions.

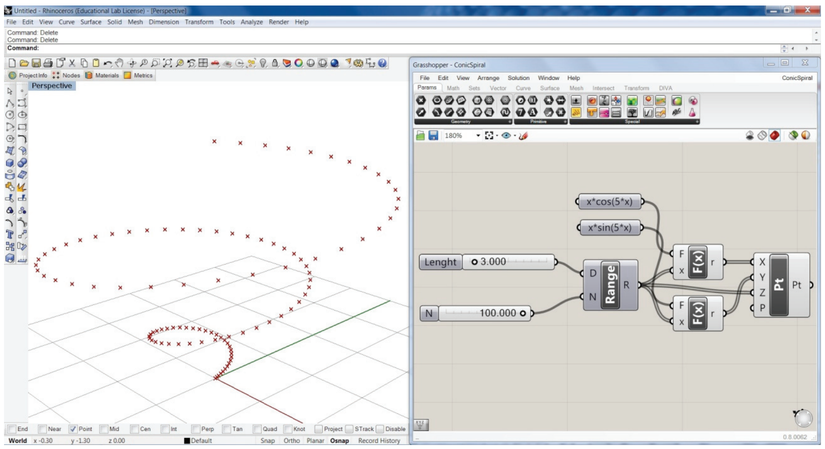
2. PROGRAMMING LANGUAGES

A programming language is more than just a means for instructing a computer to perform tasks [8]: it is a formal medium for expressing ideas. Therefore, languages should match the human thinking process, including the ability to combine simple ideas and abstract complex ones. Languages conforming to these principles provide (1) primitive elements, (2) combination mechanisms, and (3) abstraction mechanisms.

2.1. Visual and Textual Programming Languages

In a VPL, programs consist of iconic elements that can be interactively manipulated according to some spatial grammar [9].

Figure 1 shows a Grasshopper program that computes a sequence of points of a conical spiral.



◀ Figure 1: Grasshopper program for computing the points of a conical spiral with corresponding output in Rhinoceros3D.

In a TPL, programs are a linear sequence of characters. The major difference between VPLs and TPLs is the number of dimensions: TPLs are one-dimensional while VPLs are, at least, two-dimensional.

The following RhinoScript program computes a sequence of points identical to the previous Grasshopper example:

```
Function ConicSpiralPts(Length,N)
  Dim points()
  ReDim points(N-1)
  Dim t
  Dim i
  For i=0 To N-1
    t=i*Length/N
    points(i)=Pt(t*Cos(5*t),t*Sin(5*t),t)
  Next
  ConicSpiralPts=points
End Function
```

Several studies comparing VPLs and TPLs show that there is no conclusive evidence regarding their relative advantages [10]. However, it is generally admitted that VPLs are more productive and motivating for beginners. On the other hand, TPLs are considerably more productive for dealing with large-scale and complex problems and, in fact, most languages are TPLs and most programs are textual.

Nevertheless, traditional TPLs require mastering a large set of concepts that, in many cases, are just limitations of the language. For example, to understand just the first three lines of the previous RhinoScript example the reader must know (1) function syntax, (2) zero-based index arrays, (3) array declaration, and (4) redimension of non-statically sized arrays. Additional knowledge is required to understand the complete example. On the other hand, the Grasshopper example only contains the elements that are relevant to the design task, namely, input sliders, range components, functions that map over sequences of values, and wires establishing dataflow between components.

This example shows several advantages of a modern VPL over an old TPL: (1) less background knowledge; (2) presentation of all language elements in the Interactive Development Environment (IDE); and (3) immediate visual feedback, facilitating defect detection and adjustment of input parameters, and allowing incremental/interactive development.

Most TPLs have additional drawbacks: (1) the absence of a (good) IDE requires users to either remember the functionality or read extensive documentation; and (2) an iterative write-compile-execute cycle results in non-interactive development.

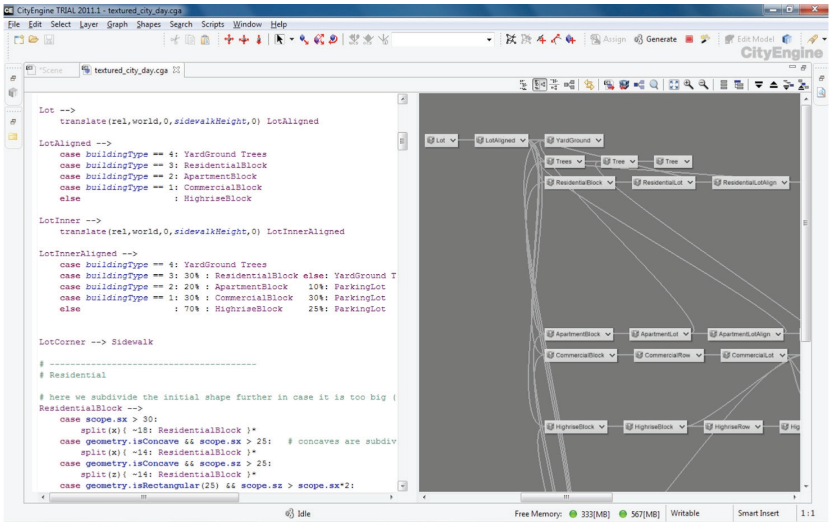
Nevertheless, VPLs also have problems: (1) VPL programs scale poorly with the complexity of the design task [11], for example, as programs grow it becomes increasingly difficult to understand what they do; and (2) the absence of (sophisticated) abstraction mechanisms forces users to rely extensively on copy/paste, introducing redundancy. In turn, redundancy leads to maintenance problems because modifications in duplicated components must be manually propagated to all instances. These problems might explain the small size and throwaway nature of the majority of visual programs when compared to the size and longevity of textual programs.

2.2. Hybrid Programming Languages

In addition to the predefined components, Grasshopper provides custom components for advanced tasks, which are textually scripted by the user, making it an example of a hybrid VPL/TPL. This section presents three additional VPL/TPL hybrids: GenerativeComponents [12], CityEngine CGA [13] and Maya Hypergraph [14].

GenerativeComponents (GC) is a parametric and associative design system that provides three forms of user interaction: (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. The recommended learning path for GC is made of three steps: (1) design using the interactive Graphical User Interface (GUI); (2) write simple scripts using the formula bar and GCScript; and (3) develop complex programs through the C# programming language, a TPL mainly used for large-scale software production. This shows that, similarly to Grasshopper, for complex design tasks, users need to become proficient in a TPL. It is generally admitted, though, that this need comes later in Grasshopper than in GC, which might explain why Grasshopper is considered easier to learn and use.

CityEngine is a modeling application for buildings and cities. It features a Python Scripting Interface and a dedicated programming language for procedural modeling: the CGA (Computer Graphics Architecture). This language 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).



```
Lot -->
    translate(rel,world,0,sidewalkHeight,0) LotAligned

LotAligned -->
    case buildingType == 4: YardGround Trees
    case buildingType == 3: ResidentialBlock
    case buildingType == 2: ApartmentBlock
    case buildingType == 1: CommercialBlock
    else                  : HighriseBlock

LotInner -->
    translate(rel,world,0,sidewalkHeight,0) LotInnerAligned

LotInnerAligned -->
    case buildingType == 4: YardGround Trees
    case buildingType == 3: 30% : ResidentialBlock else: YardGround T
    case buildingType == 2: 20% : ApartmentBlock    10%: ParkingLot
    case buildingType == 1: 30% : CommercialBlock   30%: ParkingLot
    else                  : 70% : HighriseBlock     25%: ParkingLot


LotCorner --> Sidewalk

# ----------------------------------------
# Residential

# here we subdivide the initial shape further in case it is too big (
ResidentialBlock -->
    case scope.sx > 30:
        split(x){ ~18: ResidentialBlock }*
    case geometry.isConcave && scope.sx > 25:   # concaves are subdiv
        split(x){ ~14: ResidentialBlock }*
    case geometry.isConcave && scope.sz > 25:
        split(z){ ~14: ResidentialBlock }*
    case geometry.isRectangular(25) && scope.sz > scope.sx*2:
```

◀ Figure 2: CityEngine CGA editor showing both textual and visual representations of a shape grammar that generates a city.

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

Maya is a popular 3D modeling application with a built-in scripting editor. Programs can be written in MEL and Python. When users interact with Maya, via a script or the GUI, a history of interactions is constructed. This history can then be visualized and manipulated via editors such as Outliner and Hypergraph. 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 GC and CityEngine CGA, the relationship between textual scripts and the visual editors is unidirectional: changes in these editors are not reflected in the textual program. Despite the visual editing features of Hypergraph, a TPL is still required for creating more complex algorithms.

Although there are many VPL alternatives for GD, Grasshopper is the most used one. This can be explained in part by the simplicity and attractiveness of its programming model and GUI. Moreover, there is a general perception among designers that VPLs are more productive than TPLs. We claim that this perception is a natural response to two problems: (1) traditional TPLs lack domain-specific concepts and (2) they make it difficult for the user to define them.

2.3. Modern Textual Programming Languages

Modern TPLs provide several abstractions and provide mechanisms for users to define new ones tailored to specific domains, drastically simplifying program development.

For example, consider list comprehensions, a syntactic abstraction influenced by the set-builder notation used in mathematics. The following Haskell program shows a definition of the conical spiral using list comprehensions:

```
ConicSpiralPts length n =
  [(t*(cos 5*t), t*(sin 5*t), t) | t<-range length n]
  where range d n = [i*d/n | i<-[0..n]]
```

A comparison between this example and the RhinoScript example in Section 2.1 clearly shows the amount of background knowledge that is required in each case and provides anecdotal evidence that modern TPLs can be significantly easier to understand.

Several other modern TPLs provide similar concepts. For example, in Python we can rewrite the same definition as:

```
def frange (l, n):
    return [float(i)*l/n for i in range(n+1)]

def conic_spiral_pts (length, n):
    return [[t*cos(5*t), t*sin(5*t), t]
            for t in frange(length, n)]
```

Compared to Haskell or Python, RhinoScript seems archaic, to say the least. However, it should be noted that while Haskell and Python were developed in 1990 and 1991, respectively, RhinoScript is a descendant of a long line of BASIC dialects that started much earlier, in 1964. Although there are now several differences between modern dialects of BASIC (e.g., VisualBasic and VBScript) and their ancestors, the language could not evolve as freely as possible because it was necessary to provide an easy migration path to users of older dialects. This problem occurs with several PLs for GD, including GDL for ArchiCAD and MEL for Maya. The end result is that traditional TPLs, such as Rhinoscript, can never be favorably compared with state-of-the-art VPLs, such as Grasshopper.

In this paper, we argue that it is possible and, in fact, advantageous to use TPLs instead of VPLs as long as we restrict ourselves to modern languages that target the GD domain. We support our argument using examples of modern programming techniques. We choose Scheme for its pedagogical qualities but identical examples could be provided in Haskell, Python, Javascript and many modern TPLs.

2.4. VisualScheme

VisualScheme is a research project integrated in the architecture curriculum that explores the advantages of VPLs and TPLs for GD, including domain-specific constructs, immediate feedback, visual widgets, and CAD integration. Following a series of pedagogical studies [15-17], VisualScheme relies on Scheme as a teaching tool for an audience without a background in Computer Science. The rest of this section describes several features of VisualScheme.

The first example computes the points of a conical spiral:

```
(define (conic-spiral-points length n φ)
  (for/list ([i (in-range 0 n)])
    (let ([t (/ (* i length) n]))
      (cyl t (* φ t) t))))
```

There are two noteworthy differences between this example and the Haskell and Python examples: (1) the fully parenthesized prefix notation typical of Lisp dialects; and (2) the use of the cylindrical coordinate system (via the *cyl* function), which reduces the need for trigonometric expressions.

VisualScheme implements not only the traditional Cartesian coordinate system but also the polar, cylindrical, and spherical systems.

This example follows the same approach used in the Grasshopper example in Section 2.1. However, a more parametric approach can be used:

```
(define (conic-spiral-pts r φ z Δr Δφ Δz n)
  (if (= n 0)
    (list)
    (cons (cyl r φ z)
          (conic-spiral-pts (+ z Δz)
                            (+ r Δr)
                            (+ j Δφ)
                            Δr Δφ Δz
                            (- n 1)))))
```

This example illustrates a recursive definition (i.e., a function defined in terms of itself) using the starting radius (*r*), angle (φ), and height (z), the respective increments (Δr, $\Delta\varphi$, Δz), and the number of points (*n*).

Note that this example cannot be directly encoded in Grasshopper using the standard components. Instead, custom components must be scripted using a TPL, thus contradicting the visual paradigm. In this particular case, it is possible to find different encodings that produce the same result, but in the general case this can be a serious limitation.

As a final example, consider the following definition that provides the coefficients α and β that affect the rate at which the spiral grows in the radial and vertical directions:

```
(define (conic-spiral-pts α β length n)
  (map (λ (t) (cyl (* α t) t (* β t)))
       (range length n)))
```

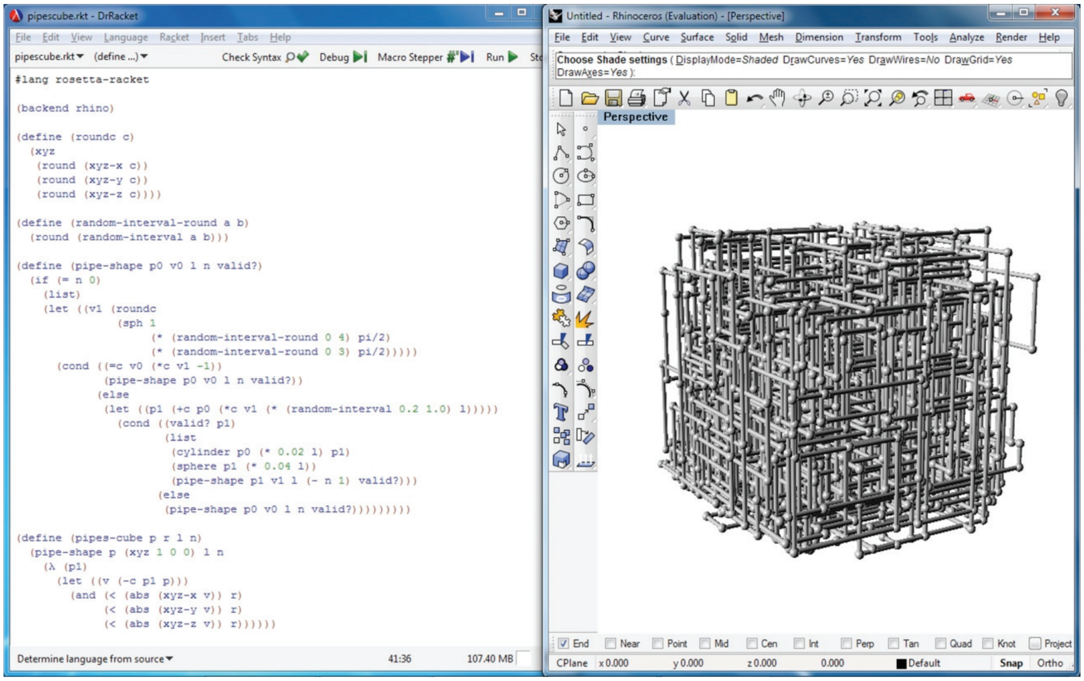
This definition uses the higher-order function *map*, which computes the mathematical image of a function over a domain. This function receives an anonymous function as argument (via λ), i.e., a function that is not explicitly named. Due to their expressive power, higher-order and anonymous functions are common in modern TPLs.

Despite the identified advantages there are three important drawbacks in VisualScheme: (1) users have to spend time learning a TPL (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 PLs for GD, VisualScheme forces the use of a particular CAD package, contributing to a known problem of current CAAD-education [18]: students become experts in a single CAD application.

2.5. Rosetta

Rosetta [19] is a programming environment designed to address the problems of VisualScheme. To this end, Rosetta allows the use of different languages (front-ends) and interoperates with different CAD applications (back-ends): (1) it is possible to write programs not only in Scheme but also in AutoLISP, JavaScript, and Racket; and (2) programs can generate output in AutoCAD and Rhinoceros3D. We plan to implement additional front-ends, such as, Processing and Python, and back-ends, such as, Revit, MicroStation and ArchiCAD.

With Rosetta, users leverage the effort spent learning a specific language because they can use that language with different CAD tools (Figure 3 and Figure 4).

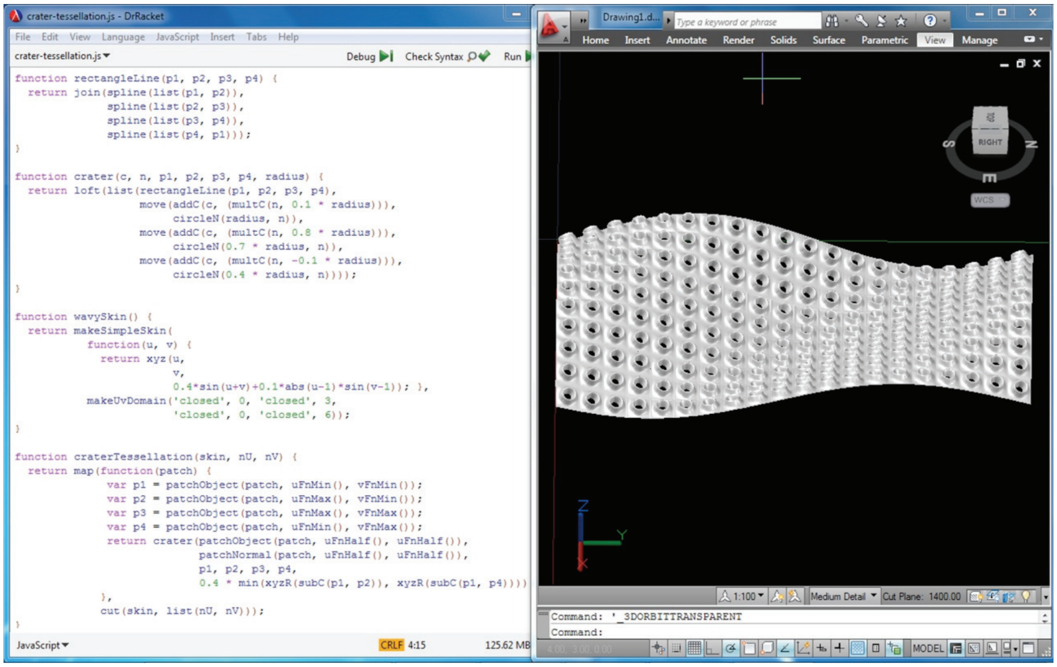

pipescube.rkt - DrRacket
File Edit View Language Racket Insert Tabs Help
pipescube.rkt (define ...)
Check Syntax
Debug
Macro Stepper
Run
Determine language from source
41:36
107.40 MB
Untitled - Rhinoceros (Evaluation) - [Perspective]
File Edit View Curve Surface Solid Mesh Dimension Transform Tools Analyze Render Help
Perspective
End Near Point Mid Cen Int Perp Tan Quad Knot Project
CPlane x 0.000 y 0.000 z 0.000 0.000 Default Snap Ortho

```
#lang rosetta-racket

(backend rhino)

(define (roundc c)
  (xyz
   (round (xyz-x c))
   (round (xyz-y c))
   (round (xyz-z c))))

(define (random-interval-round a b)
  (round (random-interval a b)))

(define (pipe-shape p0 v0 l n valid?)
  (if (= n 0)
    (list)
    (let ((v1 (roundc
                (sph 1
                     (* (random-interval-round 0 4) pi/2)
                     (* (random-interval-round 0 3) pi/2)))))
      (cond ((=c v0 (*c v1 -1))
             (pipe-shape p0 v0 l n valid?))
            (else
             (let ((p1 (+c p0 (*c v1 (* (random-interval 0.2 1.0) l)))))
               (cond ((valid? p1)
                      (list
                       (cylinder p0 (* 0.02 l) p1)
                       (sphere p1 (* 0.04 l))
                       (pipe-shape p1 v1 l (- n 1) valid?)))
                     (else
                      (pipe-shape p0 v0 l n valid?)))))))))

(define (pipes-cube p r l n)
  (pipe-shape p (xyz 1 0 0) l n
    (λ (p1)
      (let ((v (-c p1 p)))
        (and (< (abs (xyz-x v)) r)
             (< (abs (xyz-y v)) r)
             (< (abs (xyz-z v)) r))))))
```

```
Choose Shade settings ( DisplayMode=Shaded DrawCurves=Yes DrawWires=No DrawGrid=Yes
DrawAxes=Yes ):
```

▲ Figure 3. Left: Rosetta showing a Racket program that creates a pipe structure that follows a random orthogonal walk bounded by a virtual box. Right: The output of the program in Rhinoceros3D.

Figure 5 illustrates the program sharing capabilities of Rosetta: a Racket library (on the right) defines an L-System library that is imported into a JavaScript program (on the left) that uses the provided functions to produce the Arabesque in Figure 6.

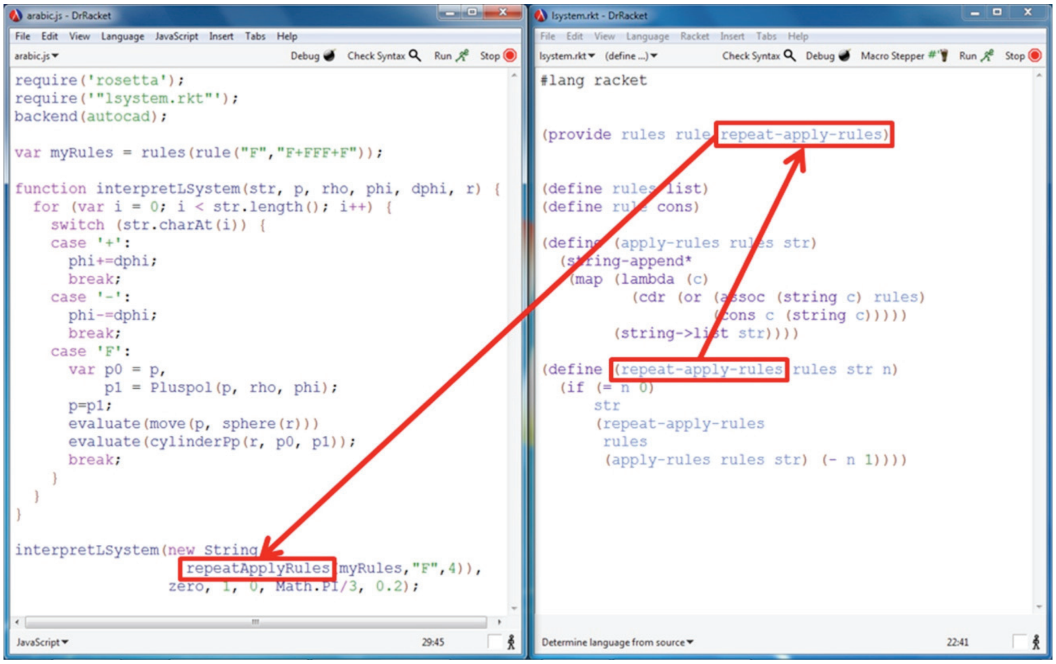

crater-tessellation.js - DrRacket
File Edit View Language JavaScript Insert Tabs Help
crater-tessellation.js
Debug
Check Syntax
Run

```
function rectangleLine(p1, p2, p3, p4) {
  return join(spline(list(p1, p2)),
              spline(list(p2, p3)),
              spline(list(p3, p4)),
              spline(list(p4, p1)));
}

function crater(c, n, p1, p2, p3, p4, radius) {
  return loft(list(rectangleLine(p1, p2, p3, p4),
                   move(addC(c, (multC(n, 0.1 * radius))),
                        circleN(radius, n)),
                   move(addC(c, (multC(n, 0.8 * radius))),
                        circleN(0.7 * radius, n)),
                   move(addC(c, (multC(n, -0.1 * radius))),
                        circleN(0.4 * radius, n))));
}

function wavySkin() {
  return makeSimpleSkin(
           function(u, v) {
             return xyz(u,
                        v,
                        0.4*sin(u+v)+0.1*abs(u-1)*sin(v-1)); },
           makeUvDomain('closed', 0, 'closed', 3,
                        'closed', 0, 'closed', 6));
}

function craterTessellation(skin, nU, nV) {
  return map(function(patch) {
               var p1 = patchObject(patch, uFnMin(), vFnMin());
               var p2 = patchObject(patch, uFnMax(), vFnMin());
               var p3 = patchObject(patch, uFnMax(), vFnMax());
               var p4 = patchObject(patch, uFnMin(), vFnMax());
               return crater(patchObject(patch, uFnHalf(), uFnHalf()),
                             patchNormal(patch, uFnHalf(), uFnHalf()),
                             p1, p2, p3, p4,
                             0.4 * min(xyzR(subC(p1, p2)), xyzR(subC(p1, p4))))
             },
             cut(skin, list(nU, nV)));
}
```

```
Command: '_3DORBITTRANSPARENT
Command:
```

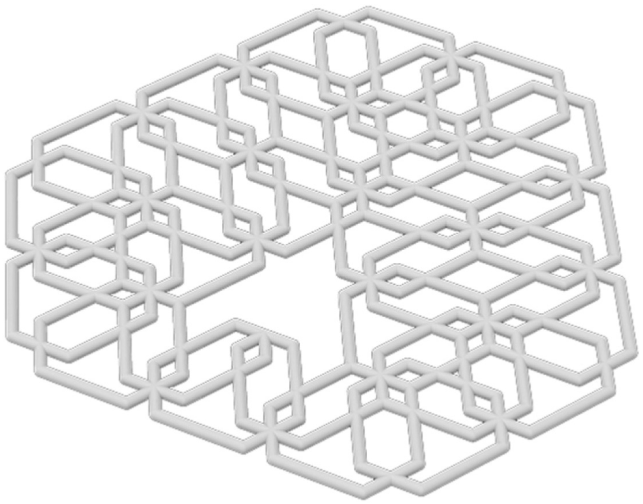
▲ Figure 4. Left: JavaScript program that creates a surface tessellation. Right: The output of the program in AutoCAD.

▼ Figure 5. Left: JavaScript program that uses an L-System (*repeat-apply-rules*) to produce Figure 6. Right: A Racket program that defines the L-System.



```
require('rosetta');
require('"lsystem.rkt"');
backend(autocad);

var myRules = rules(rule("F","F+FFF+F"));

function interpretLSystem(str, p, rho, phi, dphi, r) {
  for (var i = 0; i < str.length(); i++) {
    switch (str.charAt(i)) {
    case '+':
      phi+=dphi;
      break;
    case '-':
      phi-=dphi;
      break;
    case 'F':
      var p0 = p,
          p1 = Pluspol(p, rho, phi);
      p=p1;
      evaluate(move(p, sphere(r)))
      evaluate(cylinderPp(r, p0, p1));
      break;
    }
  }
}

interpretLSystem(new String(
                 repeatApplyRules(myRules,"F",4)),
                 zero, 1, 0, Math.PI/3, 0.2);
```

```
#lang racket

(provide rules rule repeat-apply-rules)

(define rules list)
(define rule cons)

(define (apply-rules rules str)
  (string-append*
   (map (lambda (c)
          (cdr (or (assoc (string c) rules)
                   (cons c (string c)))))
        (string->list str))))

(define (repeat-apply-rules rules str n)
  (if (= n 0)
      str
      (repeat-apply-rules
       rules
       (apply-rules rules str) (- n 1))))
```



◀ Figure 6: Arabesque produced by the L-System presented in Figure 5.

The next section presents a practical experiment comparing VPLs and TPLs. Although Rosetta would have allowed us to run the experiment using any of the supported TPLs, for simplicity reasons we used VisualScheme.

3. EXPERIMENT

The objectives of this experiment were: (1) compare properties of VPL and TPL programs, such as, parametric degree and modifiability; (2) assess whether textual programming experience has an influence on visual program design; and (3) measure the time needed for implementing VPL and TPL solutions.

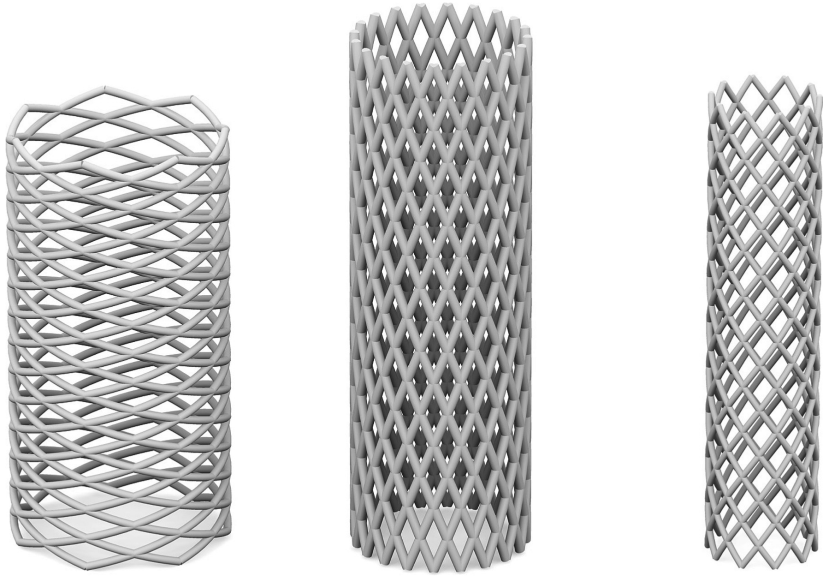
Six designers were invited to this experiment and divided in two groups: group A consisted of three Grasshopper users with no knowledge of TPLs - subjects 1, 2 and 3; and group B consisted of three Grasshopper users with experience in TPLs - subjects 4, 5 and 6. Table 1 summarizes subject expertise in VPLs and TPLs.

Group	Subjects	TPL Expertise (Any language)	VPL Expertise (Grasshopper)
	1	None	Beginner
A	2	None	Intermediate
	3	None	Advanced
	4	Intermediate	Advanced
B	5	Intermediate	Intermediate
	6	Advanced	Beginner

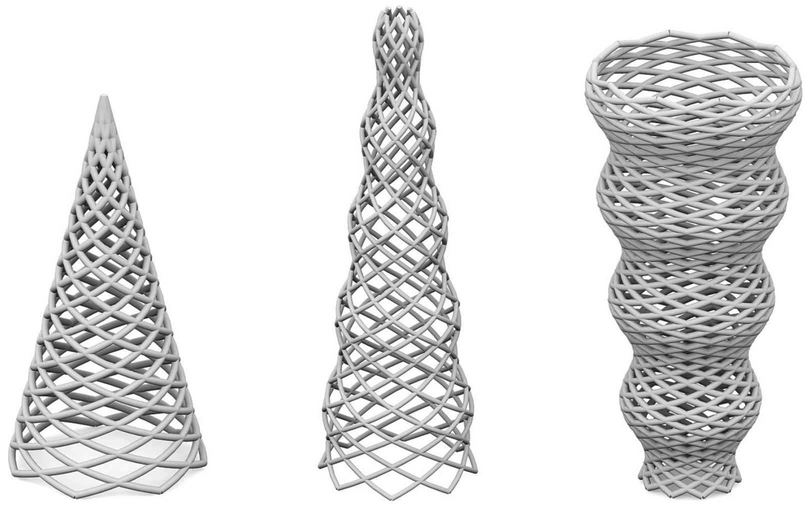
◀ Table 1: Subject expertise in TPLs and VPLs.

The challenge consisted of two phases: (1) implementation of a parametric algorithm to create a tower made of cylindrical spirals (Figure 7); and (2) modification of the original algorithms to create a variation of the cylindrical spirals (Figure 8). Phase 2 was presented to the subjects only after finishing phase 1. All subjects were asked to measure the time spent to complete the challenge.

► Figure 7: Cylindrical spiral towers.



► Figure 8: Conical and sinusoidal spiral towers.



3.1. Phase 1 - Grasshopper solutions

Discrepancies in individual times are expected because solutions differ in accuracy and parametric degree. Moreover, unpredictable mistakes during implementation may have a strong impact on results because the total number of subjects is small. Finally, each user spends a different amount of time in component placement, wire connection, data visualization, and comments. For these reasons we decided to round the times to the nearest 5 minute interval and to show group times instead of individual ones. Table 2 shows the minimum, average and maximum times (in minutes) for groups A and B to complete phase 1 of the challenge.

Groups	Minimum	Average	Maximum
A	50	94	140
B	25	37	50

◀ **Table 2: Minimum, average and maximum times (in minutes) for groups A and B to complete phase 1 of the challenge.**

From the analysis of Table 2, it is clear that the times of group B are considerably smaller than those of group A, which suggests that experience in TPLs reduces the time needed to design a visual program. The rest of this section analyzes the most relevant solutions.

Figure 9 shows a solution from a Grasshopper beginner with no knowledge of TPLs. The user approached the problem in a visual fashion, creating stacks of rotated circles, dividing them in equal parts, and using the division points in a sequential manner to create the spiral curves. Although some copy/paste was used, the resulting program is clear and concise.

▼ Figure 9: Grasshopper solution of the cylindrical spiral tower based on a set of geometrical transformations.

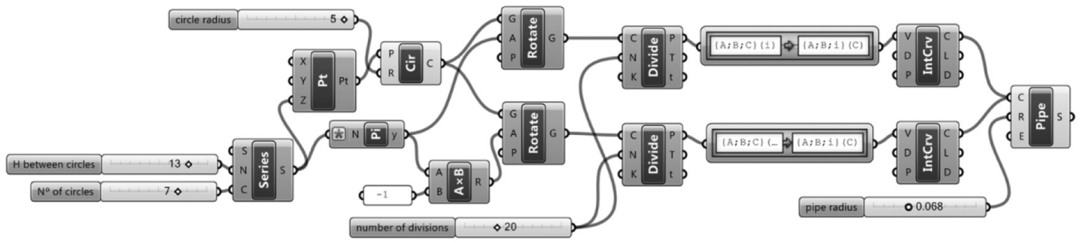
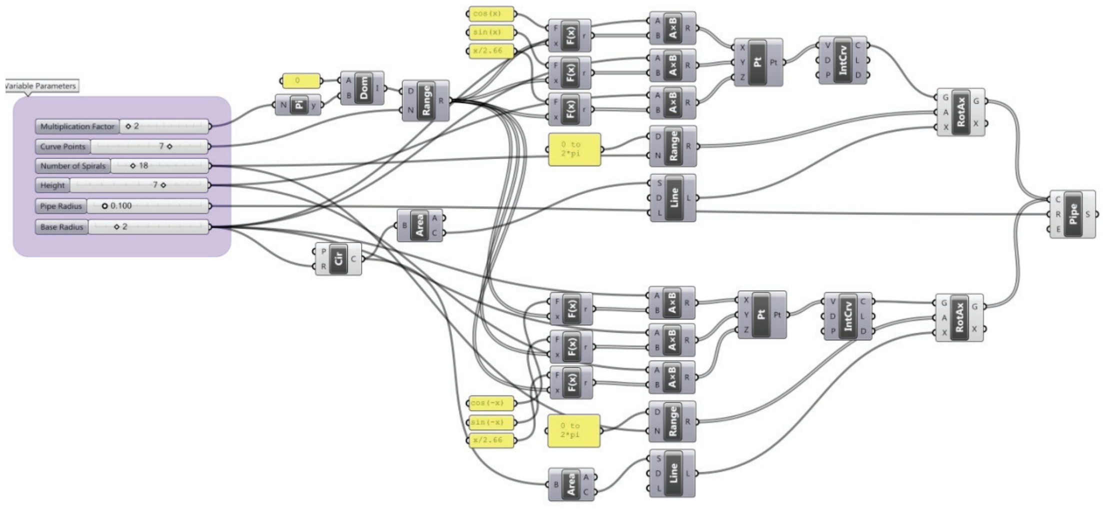


Figure 10 shows the solution from an expert in Grasshopper with no knowledge of TPLs. This solution shows a more mathematical approach where the spiral points are computed using Cartesian coordinates. However, there is redundancy resultant of extensive copy/paste and the amalgam of wire connections makes the code difficult to understand, leading to future maintenance problems [20].

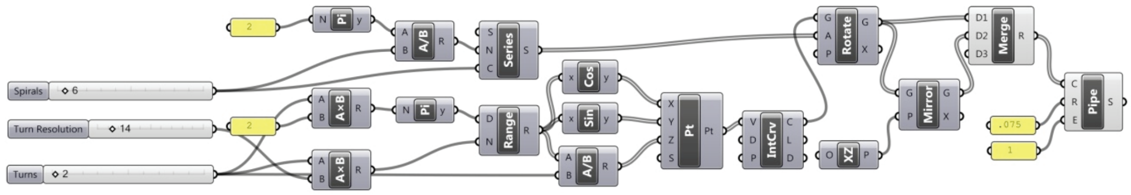
Figure 11 shows the solution from an advanced Grasshopper user with some experience in TPLs.

This solution uses the same approach as the previous one. Even though this solution is more concise and easier to read, it is also less parametric, because the height and radius are not provided as parameters.

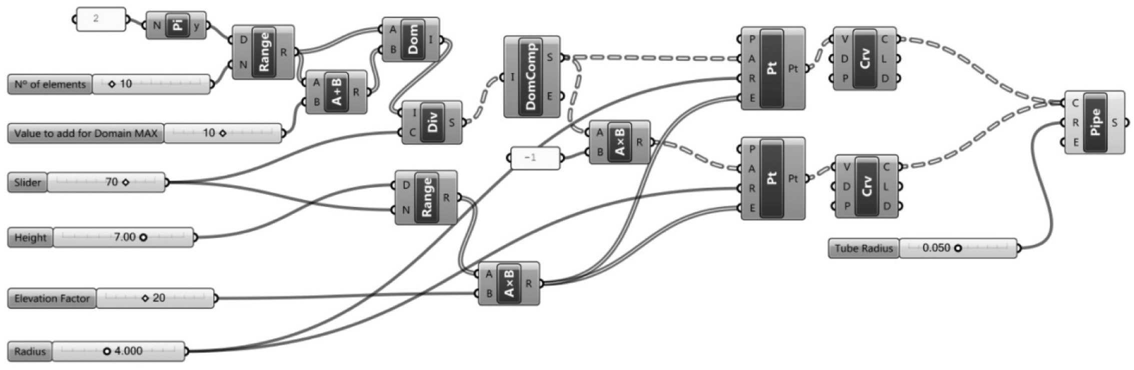


▲ Figure 10: Grasshopper solution of the cylindrical spiral tower based on the conical spiral Cartesian definition.

▼ Figure 11: Grasshopper solution of the cylindrical spiral tower based on a less parametric conical spiral Cartesian definition.



Finally, Figure 12 presents the most abstract and parametric definition. The program is not only more compact but also easier to read and to maintain. This program was developed by a beginner in Grasshopper but expert in TPLs.



▲ Figure 12: Grasshopper solution of the cylindrical spiral tower using cylindrical coordinates.

3.2. Phase 1 - VisualScheme solution

This section describes a VisualScheme solution developed in 20 minutes by an advanced TPL user. The tower was modeled as two sets of spirals, with opposite turning directions, described by cylindrical coordinates. To achieve a parametric solution, each spiral was defined as three linear variations of the cylindrical components, i.e., the radius, the angle and the height.

VisualScheme does not provide a predefined concept of linear variation. Therefore, the user is responsible for defining it. A linear variation in the range *[a,b]* was defined as a function over the domain *[0,1]*:

```
(define (linear a b)
  (λ (t)
    (+ a (* t (- b a)))))
```

This approach uses higher-order functions, allowing many different variations to be easily implemented. The *map* and *range* functions helped computing the actual values defined in the variation:

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

The conic spiral is then the mapping of cylindrical coordinates over three linear variations of the radius, angle, and height:

```
(define (spiral-points r0 r1 φ0 φ1 h n)
  (map cyl
       (variation (linear r0 r1) n)
       (variation (linear φ0 φ1) n)
       (variation (linear 0 h) n)))
```

To create one set of spirals, the concept of linear variation was reused to provide a sequence of starting angles between *0* and *2π*, according to the intended number of spirals *s* and to the number of turns *t* that each spiral should follow:

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

In order to create a mesh of opposing spirals, two calls to the previous function were combined, the second one with a symmetrical number of turns:

```
(define (spirals-mesh r0 r1 h d f s t n)
  (append (spirals r0 r1 h d f s t n)
          (spirals r0 r1 h d f s (- t) n)))
```

Comparing the solutions, it is clear that, although not as aesthetically pleasing, this program is more analytic and modular than the Grasshopper programs. 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 every component is necessarily connected to another.

3.3. Phase 2 - Grasshopper solutions

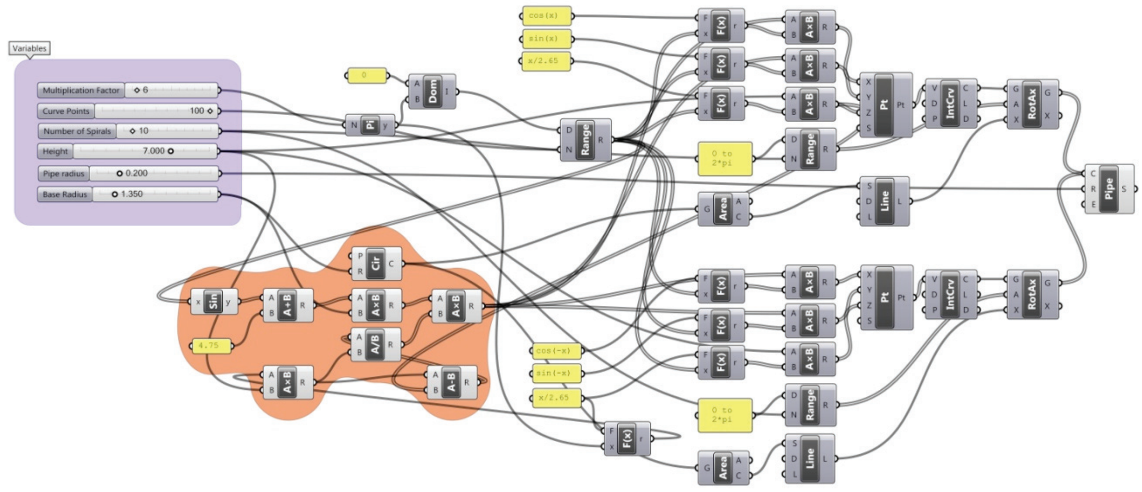
In GD, programs must be easily adaptable to changing requirements. To measure adaptability in VPLs and TPLs, subjects were asked to adapt the previous solutions to create the conical and sinusoidal towers illustrated in Figure 8. Table 3 shows the minimum, average and maximum times (in minutes) for groups A and B to complete phase 2 of the challenge in Grasshopper.

► Table 3: Minimum, average and maximum times (in minutes) for groups A and B to complete phase 2 of the challenge.

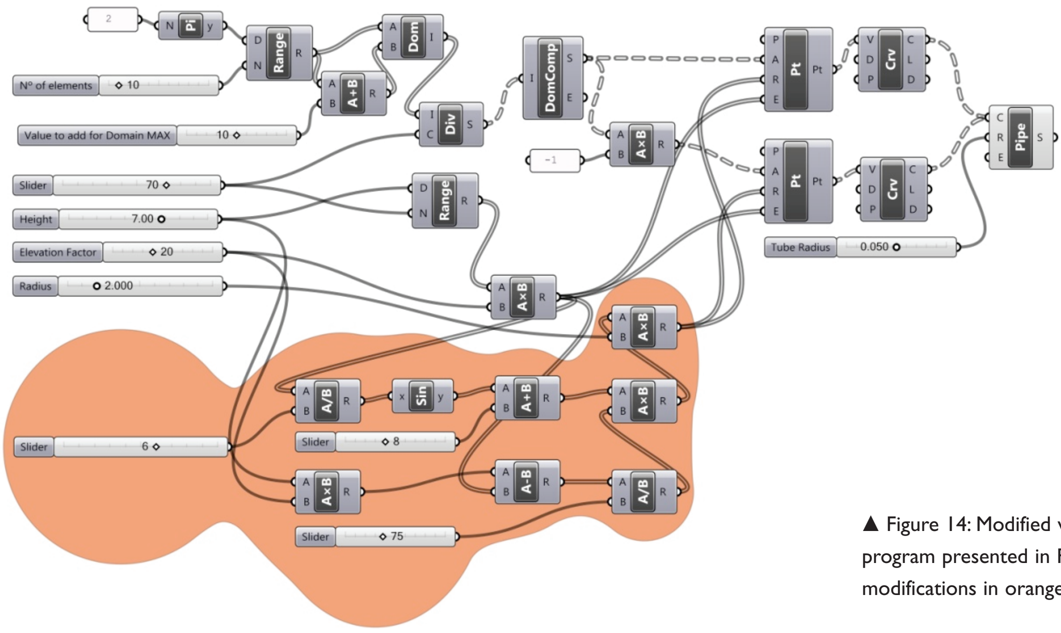
Groups	Minimum	Average	Maximum
A	20	37	60
B	20	23	25

Despite the differences in expertise within the groups, Table 3 shows that group A needed more time on average than group B.

Figure 13 shows the modified version of the solution presented in Figure 10, where conical and sinusoidal variations were incorporated.



▲ Figure 13: Modified version of the program presented in Figure 10, with changes marked in orange.

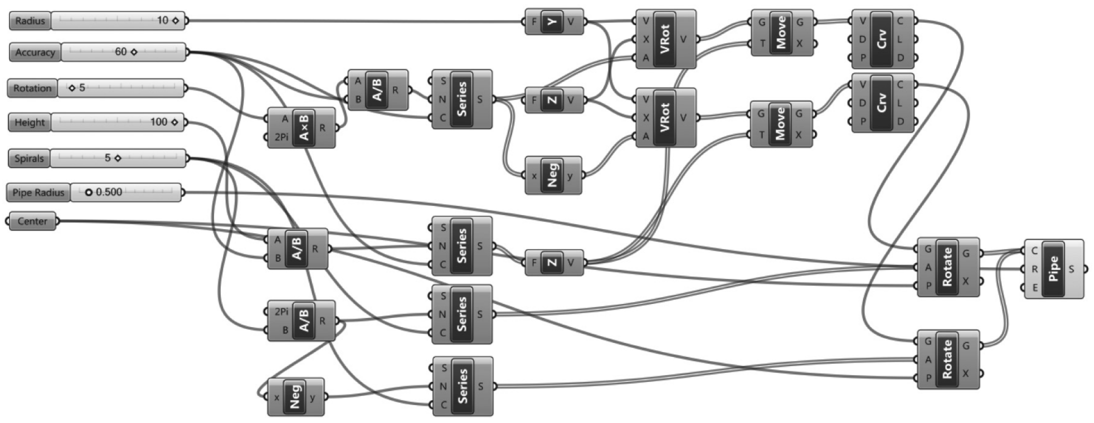


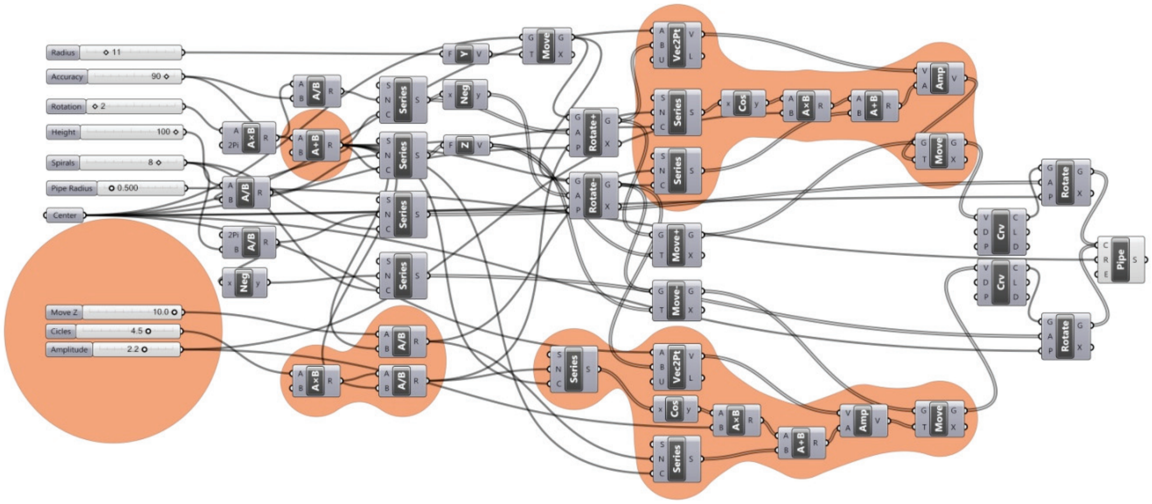
▲ Figure 14: Modified version of the program presented in Figure 12 with modifications in orange.

Figure 14 highlights the modifications made to the program in Figure 12 to accommodate the requested variations.

Figure 13 and Figure 14 show that both users were able to extend their initial definitions by introducing an isolated group of components. However, this was not consistently observed: Figure 15 and Figure16 show how a relatively simple solution for phase 1 becomes very difficult to understand after the changes needed for phase 2.

▼ Figure 15: Solution provided by an intermediated user of Grasshopper and TPLs to the phase 1 challenge.





▲ Figure 16: Modified version of the program presented in Figure 15 with modifications in orange.

Although Grasshopper allows quick changes in inputs, we observed that behavioral changes imply manual manipulation of dataflow wires from/to different components, a time consuming and error prone task. This means that even though the visual interface and dataflow paradigm are easier for novices, they can introduce readability and maintenance problems with increasingly complex design tasks.

3.4. Phase 2 - VisualScheme solution

Regarding VisualScheme, the original solution was already parametric enough to handle the conical variation. Therefore, it was only necessary to implement the changes needed for the sinusoidal variation, which were implemented in less than 10 minutes by reusing the previous concept of variation:

```
(define (sinusoidal d ω)
  (λ (t)
    (* d (sin (* 2 pi ω t)))))
```

To superimpose the sinusoidal variation with the linear one, we can calculate the function that adds the values computed by the respective functions for the same inputs:

```
(define (+fx f g)
  (λ (x)
    (+ (f x) (g x))))
```

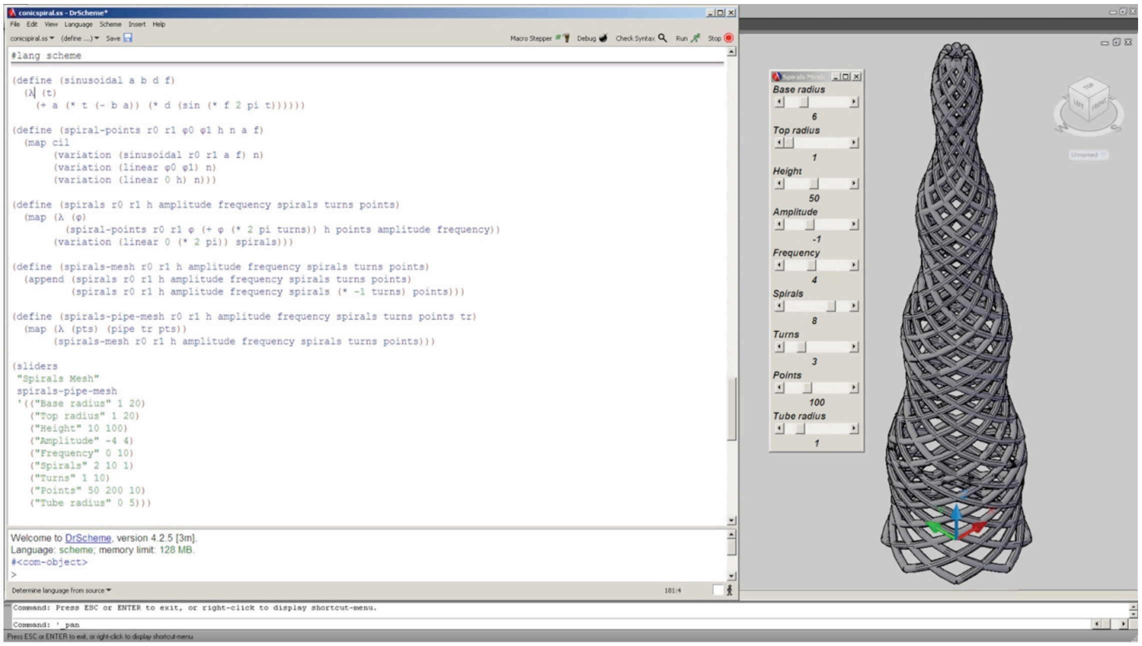
Finally, the spiral whose radius shows a *linear-plus-sinusoidal* variation along the height is:

```
(define (spiral-points r0 r1 φ0 φ1 h d ω n)
  (map cyl
       (variation (+fx (linear r0 r1)(sinusoidal d ω)) n)
       (variation (linear φ0 φ1) n)
       (variation (linear 0 h) n)))
```

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.

Figure 17 shows the VisualScheme IDE with the sinusoidal tower program generated in AutoCAD. Also visible is a panel of sliders for interactive manipulation of the function parameters.

▼ Figure 17: The VisualScheme IDE running on top of AutoCAD. The sliders window allows quick experimentation of the function parameters by regenerating the corresponding geometry in real time.


conicspiral.ss - DrScheme*
File Edit View Language Scheme Insert Help
conicspiral.ss (define ...) Save
Macro Stepper Debug Check Syntax Run Stop
Spirals Mesh

```
#lang scheme

(define (sinusoidal a b d f)
  (λ (t)
    (+ a (* t (- b a)) (* d (sin (* f 2 pi t))))))

(define (spiral-points r0 r1 φ0 φ1 h n a f)
  (map cil
       (variation (sinusoidal r0 r1 a f) n)
       (variation (linear φ0 φ1) n)
       (variation (linear 0 h) n)))

(define (spirals r0 r1 h amplitude frequency spirals turns points)
  (map (λ (φ)
          (spiral-points r0 r1 φ (+ φ (* 2 pi turns)) h points amplitude frequency))
       (variation (linear 0 (* 2 pi)) spirals)))

(define (spirals-mesh r0 r1 h amplitude frequency spirals turns points)
  (append (spirals r0 r1 h amplitude frequency spirals turns points)
          (spirals r0 r1 h amplitude frequency spirals (* -1 turns) points)))

(define (spirals-pipe-mesh r0 r1 h amplitude frequency spirals turns points tr)
  (map (λ (pts) (pipe tr pts))
       (spirals-mesh r0 r1 h amplitude frequency spirals turns points)))

(sliders
 "Spirals Mesh"
 spirals-pipe-mesh
 '(("Base radius" 1 20)
   ("Top radius" 1 20)
   ("Height" 10 100)
   ("Amplitude" -4 4)
   ("Frequency" 0 10)
   ("Spirals" 2 10 1)
   ("Turns" 1 10)
   ("Points" 50 200 10)
   ("Tube radius" 0 5)))
```

```
Welcome to DrScheme, version 4.2.5 [3m].
Language: scheme; memory limit: 128 MB.
#<com-object>
>
```

Determine language from source 181:4

```
Command: Press ESC or ENTER to exit, or right-click to display shortcut-menu.
Command: '_pan
```

Press ESC or ENTER to exit, or right-click to display shortcut-menu

Base radius 6
Top radius 1
Height 50
Amplitude -1
Frequency 4
Spirals 8
Turns 3
Points 100
Tube radius 1

4. EVALUATION

In the previous sections we presented different programming languages with a strong emphasis on Grasshopper as representative of VPLs and VisualScheme as representative of TPLs. We also discussed several advantages and disadvantages of each.

In this section, we start by comparing them according to the three fundamental dimensions of programming languages [8]: primitives, combinations, and abstractions.

In what regards primitives, Grasshopper is in a very good position: it 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, a significant advantage over VisualScheme that currently does not implement as many primitives as Grasshopper.

In what regards the combination mechanisms, Grasshopper relies on an extremely simple metaphor: primitives can be combined by connecting the output of a component to the input of another. The connections allow dataflow from primitive to primitive, until it reaches the end of the graph, usually, in primitives that create geometric models. Unfortunately, this metaphor is too restrictive, making it difficult to express some control structures, such as iteration or recursion, a complaint expressed by most TPL users involved in the experiment. In some cases, this is not a serious problem because most components implicitly map operations over sequences of values. However, it has been repeatedly reported [21-23] that it might be difficult or impossible to describe an algorithm without textually scripting a specialized component, thus contradicting the visual nature of the language. Moreover, Grasshopper programs scale poorly with the complexity of the design task, resulting in readability and maintainability problems.

In VisualScheme and Rosetta, the available combination mechanisms are provided by the underlying languages, which include expression composition and several control and data structures. Moreover, a variety of programming paradigms are provided, such as functional, imperative, and object-oriented, as opposed to Grasshopper which forces a single paradigm. Moreover, the set of paradigms can be extended. The net result is that VisualScheme and Rosetta are more expressive in the sense that they address a broader spectrum of design tasks.

Finally, abstraction: the fundamental mechanism for dealing with problem complexity. To this end, Grasshopper provides a special component, the cluster, which allows the user 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 operations and does not really represent an abstraction: each cluster is independent from its copies, thus preventing centralized definitions. In this regard, modern TPLs offer different forms of procedural, data, and control abstraction, thus being significantly more abstract than VPLs, such as Grasshopper. This might require a more analytical effort from TPL users, but it has the significant advantage that it greatly simplifies the solution and the resulting programs are usually easier to adapt to changing requirements.

Besides the linguistic dimensions, there is a learning dimension that must be considered. In this regard, they have a serious disadvantage because, in most cases, TPLs have a longer learning curve than VPLs. While this seems to suggest that learning Grasshopper is a better use of time, it is also important to note that complex programs tend to require much more time to develop using VPLs than TPLs. In the end, the time spent learning a TPL is quickly recovered once the complexity of the problem becomes sufficiently large. Finally, as it is clear from the presented experiment, experience in TPLs improved the productivity of VPL users, which suggests that GD curricula should include exposure to TPLs.

5. CONCLUSION

Nowadays, within the GD community, VPLs, such as Grasshopper, are becoming increasingly popular, which can be explained by the fact that they are state-of-the-art, domain-specific languages while textual alternatives, such as RhinoScript, are now considered obsolete languages.

Modern TPLs, such as Haskell, Python, or Scheme, were designed to be easier to learn, use and extend. When coupled with domain-specific primitives, they become better alternatives to current VPLs for GD. As an example, we considered VisualScheme, a research project that takes advantage of the pedagogical qualities of Scheme for teaching programming to Architecture students.

In order to evaluate the adequacy of VisualScheme for the GD domain, this paper presented an experiment comparing solutions to design problems solved in Grasshopper and in VisualScheme. We plan to make more extensive experiments in the near future but our preliminary findings show that the visual paradigm of Grasshopper does not scale well with the complexity of the design task, due to its shortcomings in abstraction and control mechanisms, and to the time-consuming metaphor of program construction based on the manipulation of wires and boxes.

Grasshopper is actually capable of overcoming its VPL limitations with textually scripted components. However, these components force the user to work at the TPL level, thus showing that even Grasshopper users might need to learn and use a TPL.

Learning a TPL takes more time and effort than learning a VPL, but this effort is quickly recovered when the complexity of the problems becomes sufficiently large. Unfortunately, there is always a risk that the language we are using now is not available in other CAD tools or is unknown by other members of the design team. To overcome these problems, we presented Rosetta, a descendant of VisualScheme that allows the simultaneous use of different programming languages and different CAD applications.

ACKNOWLEDGEMENTS

This work was supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds. The authors would like to acknowledge the contribution of Brimet Silva, Bruno Matos, Eduardo Castro e Costa, Paulo Fontaínha and Susana Martins for participating in the experiment presented in this paper.

References

1. Krüger, M., Duarte, J. P. and Coutinho, F., Decoding De re aedificatoria: Using Grammars to Trace Alberti's Influence on Portuguese Classical Architecture, *Nexus Network Journal*, 2011, 13(1), 171-182.
2. Rocker, I., When Code Matters, *AD - Architectural Design*, 2006, 76(4), 16-25.
3. Kolarevic, B., Eternity, Infinity and Virtuality in Architecture, in: Clayton, M. and Velasco, G., eds., *ACADIA 2000: Eternity, Infinity, and Virtuality in Architecture*, ACADIA-Association, Washington D.C., USA, 2000, 251-256.
4. Maeda, J., *Design by Numbers*, MIT Press, Cambridge, Massachusetts, USA, 1999.
5. Killian, A., Design innovation through constraint modeling, *International Journal of Architectural Computing*, 2006, 4(1), 87-105.
6. Terzidis, K., *Expressive Form: A conceptual approach to Computational Design,* Spon Press, London and New York, 2003.
7. Leitão, A., Cabecinhas, F. and Martins, S., Revisiting the Architecture Curriculum, in: Schmitt, G., Hovestadt, L. and Gool, L., eds., *ECAADe 2010 Conference: Future Cities: Proceedings of the 28th Conference on Education in Computer Aided Architectural Design in Europe*, Verlag der Fachvereine Hochschulverlag AG an der ETH Zurich, Zurich, Switzerland, 2010, 81–88.
8. Abelson, H. and Sussman, G., *Structure and interpretation of computer programs*, MIT Press, Cambridge, Massachusetts, USA, 1996.
9. Myers, B. A., Taxonomies of Visual Programming and Program Visualization, *Journal of Visual Languages and Computing*, 1990, 1(1), 97–123.
10. Menzies, T., Evaluation Issues for Visual Programming Languages, in: Chang, S. K., ed., *Handbook of Software Engineering and Knowledge Engineering, Vol.2 Emerging Tecnologies*, World Scientific Publishing Co. Pte. Ltd., London, UK, 2002, 93-101.
11. Park, K. and Holt, N., Parametric Design Process of a Complex Building In Practice Using Programmed Code As Master Model, *International Journal of Architectural Computing*, 2010, 8(3), 359-376.
12. Aish, R. and Woodbury, R., Multi-Level Interaction in Parametric Design, in: Butz, A., Krüger, A. and Olivier, P., eds., *SG 2005 Conference Proceedings: International Symposium on Smart Graphics*, Springer, Berlin, Heidelberg, Germany, 2005, 151-162.
13. Müller, P., Wonka, P., Haegler, S., Ulmer, A. and Gool, L., Procedural modeling of buildings, in: *ACM SIGGRAPH 2006 Papers (SIGGRAPH '06)*, ACM, New York, NY, USA, 614-623.
14. Wilkins, M. R. and Kazmier, C., *MEL Scripting for MAYA Animators*, Elsevier, Morgan Kauffmann Publishers, San Francisco, California, USA, 2005.
15. Dingle, A. and Zander, C., Assessing the ripple effect of CS1 language choice, *Journal of Computing Sciences in Colleges*, 2001, 16(2), 85-93.
16. Findler, C., Flanagan, F., Krishnamurthi, S., and Felleisen, M., DrScheme: A Programming Environment for Scheme, *Journal of Functional Programming*, 2002, 12(2), 159-182.
17. Felleisen, M., Findler, R., Flatt, M. and Krishnamurthi, S., The TeachScheme! Project: Computing and Programming for Every Student, *Computer Science Education*, 2004, 14(1), 55-77.
18. Pentillä, H., Architectural-IT and Educational Curriculums – A European Overview, *International Journal of Architectural Computing*, 2003, 1(1), 102-111.
19. Lopes, J. and Leitão, A., Portable Generative Design for CAD Applications, in: Taron, J., Parlac, V., Kolarevic, B. and Johnson, J., eds., *ACADIA 2011: Integration through Computation: Proceedings of the 31st annual conference of the Association for*

Computer Aided Design in Architecture (ACADIA), ACADIA-Association, Banff, Alberta, Canada, 2011, 196-203.

20. Davis, D., Burry, M. and Burry, J., Untangling Parametric Schemata: Enhancing Collaboration Through Modular Programming, in: Leclercq, P., Heylighen, A. and Martin, G., eds., *Designing Together - CAAD Futures 2011*, Les Editions de l'Université de Liège, Liège, Belgium, 2011, 55-78.
21. Stouffs, R. and Chang, W.-T., Representational programming for design analysis, in: Tizani, W., ed., *Computing in Civil and Building Engineering: Proceedings of the International Conference*, Nottingham University Press, Notthingham, UK, 2010, 351-359.
22. Chok, K., Progressive Spheres of Innovation: Efficiency, communication and collaboration, in: Taron, J., Parlac, V., Kolarevic, B. and Johnson, J., eds., *ACADIA 2011: Integration through Computation: Proceedings of the 31st annual conference of the Association for Computer Aided Design in Architecture (ACADIA)*, ACADIA-Association, Banff, Alberta, Canada, 2011, 234-241.
23. Miller, N., The Hangzhou Tennis Center - A Case Study in Integrated Parametric Design, in: Cheon, J., Hardy, S. and Hemsath, T., eds., *Proceedings of the 2011 Association for Computer Aided Design in Architecture (ACADIA) Regional Conference*, ACADIA-Association, Lincoln, Nebraska, USA, 2011.

António Leitão[1], Luís Santos[2], and José Lopes[1]

[1]Instituto Superior Técnico, Technical University of Lisbon/INESC-ID
Av. Rovisco Pais, 1
1049-001 Lisboa, Portugal

antonio.menezes.leitao@ist.utl.pt, jose.lopes@ist.utl.pt

[2]IHSIS - Institute for Humane Studies and Intelligent Sciences
Rua dos Freixos 9
2750-007 Cascais, Portugal

luis.sds82@gmail.com