Portable Generative Design for CAD Applications

José Lopes
Instituto Superior Técnico

António Leitão
Instituto Superior Técnico

ABSTRACT

Most CAD applications provide programming languages for automation and generative design. However, programs written in these languages are not portable because they execute only in the family of CAD applications for which they were originally written. Consequently, users are locked-in to one family of CAD applications and they cannot reuse programs written for other families.

In this paper, we propose a solution to this problem: Rosetta, a programming environment that is compatible with several CAD applications. Rosetta is composed of (1) an abstraction layer that allows portable and transparent access to several different CAD applications; (2) back-ends that translate the abstraction layer into different CAD applications; (3) front-end programming languages in which users write the generative design programs; and (4) an intermediate programming language that encompasses the language constructs essential for geometric modeling and that is used as a compilation target for the front-ends.

Rosetta allows users to explore different front-ends and back-ends, in order to find a combination that is most suitable for the problem at hand. As a result, users have access to different programming languages, namely, visual and textual, which can be used interchangeably to write generative design programs, without breaking portability. Furthermore, Rosetta ensures that a single program can be used to create identical geometric models 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 the programs and of the underlying ideas.

1 Introduction

Portability is a software requirement of the utmost importance because it is a key factor for software reuse. Software reuse is a common programming practice that increases not only the productivity of the software development process, but also the quality of the software itself. Moreover, software reuse is the key survival factor for small and medium-sized programmer communities.

For example, the Perl programming language 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 cover practically every task they might think of. CPAN relies on the contributions of the Perl community and it is one of the reasons for the success of Perl. The AutoLISP community is another example: there are thousands of AutoLISP scripts available on the Internet that AutoCAD users can use to speed up their design tasks.

In addition, portability and reusability are qualities that allow software to adapt to new environments. For example, consider the set of programs written in RhinoScript, the programming language of Rhinoceros 3D. These programs might become useless if Rhinoceros 3D replaces RhinoScript with another programming language. In this scenario, users have two options: either discard the RhinoScript programs or rewrite them in a supported programming language. Unfortunately, both options are inadequate.

Even though there are many CAD applications, the fact remains that programs written in the programming language of a particular CAD application are not portable across the majority of others. This strong coupling between a program and a CAD application is not only dangerous to the survival of the program itself, it also damages the software process productivity, the software quality and the communities.

In order to address this problem, we propose Rosetta, a programming environment that (1) is specifically tailored for generative design, (2) integrates with different CAD applications, and (3) accommodates different programming languages.

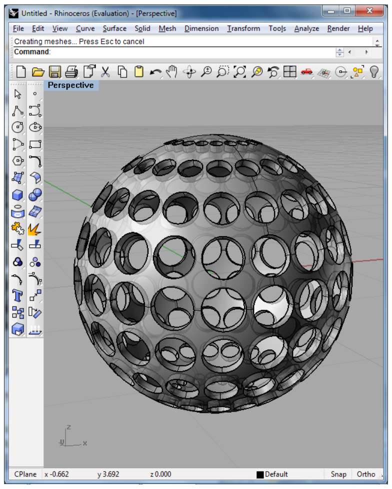
In the rest of this paper, we explain the implementation of Rosetta and we illustrate its use with examples from generative design. Finally, we evaluate Rosetta, we compare it with similar approaches and we present the conclusions.

2 Rosetta

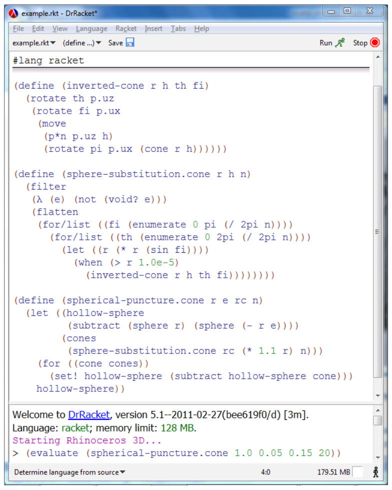
Programs written in the programming languages provided by CAD applications suffer from portability issues. In order to overcome this difficulty, we have been developing Rosetta, a flexible and extensible programming environment that connects front-end programming languages (front-ends) and back-end CAD applications (back-ends). The front-ends represent the programming languages in which users write the generative design programs. The back-ends represent the modeling target, in which the geometric models that result from the generative design programs are created. **(Figure 1)** illustrates Rosetta with the Racket programming language and the Rhinoceros 3D back-end creating a perforated hollow sphere.

The loose coupling between the front-ends and back-ends gives users the flexibility to explore different combinations. For example, one user can write a generative design program in AutoLISP and choose Rhinoceros 3D as modeling target, while another user can choose JavaScript as front-end and AutoCAD as back-end. On the other hand, Rosetta is extensible in the sense that it allows new front-ends and back-ends to be plugged-in to the programming environment, without breaking the portability with existing programs.

In order to accommodate both flexibility and extensibility in the same architecture, we defined an abstraction layer, which represents the Application Programming


Untitled - Rhinoceros (Evaluation) - [Perspective]
File Edit View Curve Surface Solid Mesh Dimension Transform Tools Analyze Render Help
Perspective
CPlane x -0.662 y 3.692 z 0.000 Default Snap Ortho

Creating meshes... Press Esc to cancel
Command:


example.rkt - DrRacket*
File Edit View Language Racket Insert Tabs Help
example.rkt (define ...) Save Run Stop

```
#lang racket

(define (inverted-cone r h th fi)
  (rotate th p.uz
   (rotate fi p.ux
    (move
     (p*n p.uz h)
     (rotate pi p.ux (cone r h))))))

(define (sphere-substitution.cone r h n)
  (filter
   (λ (e) (not (void? e)))
   (flatten
    (for/list ((fi (enumerate 0 pi (/ 2pi n))))
      (for/list ((th (enumerate 0 2pi (/ 2pi n))))
        (let ((r (* r (sin fi))))
          (when (> r 1.0e-5)
            (inverted-cone r h th fi))))))))

(define (spherical-puncture.cone r e rc n)
  (let ((hollow-sphere
         (subtract (sphere r) (sphere (- r e))))
        (cones
         (sphere-substitution.cone rc (* 1.1 r) n)))
    (for ((cone cones))
      (set! hollow-sphere (subtract hollow-sphere cone)))
    hollow-sphere))
```

```
Welcome to DrRacket, version 5.1--2011-02-27(bee619f0/d) [3m].
Language: racket; memory limit: 128 MB.
Starting Rhinoceros 3D...
> (evaluate (spherical-puncture.cone 1.0 0.05 0.15 20))
```

Determine language from source 4:0 179.51 MB

Fig. 1

Figure 1. *Perforated Hollow Sphere program in Rosetta using the Racket programming language and the Rhinoceros 3D back-end*

Interface (API) of the programming environment, and an intermediate programming language (IPL), which provides the core language constructs essential for geometric modeling. When combined, the abstraction layer and the IPL establish a communication protocol between the front-ends and the back-ends.

The rest of this section is dedicated to exploring the aforementioned components of the architecture in further detail, namely, the abstraction layer, the IPL, and the front-ends/back-ends.

2.1 ABSTRACTION LAYER

The abstraction layer encompasses the functionality common to CAD applications, which includes (1) shape constructors, the procedures to create geometric shapes, namely, circles and boxes; and (2) transformations, the procedures that apply geometric transformations, including, translations, lofts, extrusions, and sweeps.

In order to understand the relevant shape constructors and transformations for generative design, several CAD applications were surveyed and the results were generalized in order to accommodate the most used CAD applications.

2.2 INTERMEDIATE PROGRAMMING LANGUAGE

The IPL is the compilation target of the front-ends, but it can also be used directly to write generative design programs. This language is an extension to Racket, a dialect of the Scheme programming language (Kelsey 1998). The IPL is supported by DrRacket, a pedagogic programming environment suitable for the development of programming languages (Findler 2002; Flatt 1999).

Because the IPL is used as a compilation target for the front-ends, the main focus of this language is to provide the core functionality essential for generative design. Therefore, certain linguistic features, such as presentation (i.e., syntax), are not addressed by this language. In fact, because the presentation is such an important aspect of a language and there are several possible syntaxes, including visual and textual ones, the only way to accommodate all possibilities is to defer the choice of the presentation to the front-ends, while maintaining an abstract syntax in the IPL. The need for an abstract syntax is one of the reasons for supporting the choice of Racket as a starting point for the IPL.

By using the IPL, either directly or as a compilation target, generative design programs can be executed, generating the appropriate geometric models in the CAD application of choice. As part of its execution, the program generates internally a scene graph which represents the structure of the geometric models to be drawn, including their relationships and transformations (Döllner 2000). Before feeding the scene graph into the CAD application, several transformations and optimizations are performed to adapt the scene graph to the requirements of that particular CAD application.

2.3 FRONT-ENDS / BACK-ENDS

The front-ends are the programming languages in which users write the generative design programs. Users accustomed to the Scheme language will feel at home using directly the IPL, but those who prefer something different might appreciate the different languages that can be used on the front-end, both textual and visual. After surveying the programming language landscape through the generative design prism, we selected a representative subset: AutoLISP, Processing, and JavaScript. In order to accommodate both textual and visual presentations, we also considered Grasshopper and GenerativeComponents. We plan to provide emulation for these languages and we already have preliminary versions for three different front-ends, namely, RosettaFlow, Rosetta AutoLISP and Rosetta JavaScript. The rest of this section is dedicated to explaining these front-ends.

RosettaFlow is a visual programming language inspired in Grasshopper and Generative Components. Similarly to Grasshopper, RosettaFlow represents a generative design program as a data flow diagram, in which an operation is represented by a box and data flow is represented by a connector between boxes. This application allows the user to describe the generative design program visually and automatically generate the code in the IPL. The generated code is then executed in Rosetta and the geometric models are created in the CAD application of choice.

Rosetta AutoLISP and Rosetta JavaScript are textual front-ends that faithfully implement the syntax and semantics of AutoLISP and JavaScript, respectively, using normalized modeling primitives. The main purpose of these front-ends is to attract the large community of designers that learned and used these languages in the past and to simplify their transition to Rosetta.

Where back-ends are concerned, we have implemented two, namely, for AutoCAD and Rhinoceros 3D, and we plan to implement more. Given that the back-ends are highly dependent both on the API provided by the CAD application and on the communication framework used to access that API, it is not expected that each of the currently implemented back-ends provides the same coverage over the corresponding CAD application.

3 Evaluation / Results

In this section, we perform the evaluation of Rosetta and its main components, resorting to examples of generative design programs.

3.1 ROSETTA PROGRAMMING ENVIRONMENT

Rosetta has been successfully used for the development of several different generative design programs written in Racket, AutoLISP, JavaScript, and RosettaFlow. In this paper we illustrate only a few examples.

The DrRacket programming environment, in which Rosetta is based, has been instrumental in this development effort by providing the editor with the typical features required for programming, including syntax highlighting and text formatting, allowing the debugger to quickly identify the cause and the location of programming errors, and the listener has been a fundamental tool for incremental development and interactive testing. This allows us to quickly experiment with generative design programs with different combinations of parameters.

3.2 ABSTRACTION LAYER

As mentioned before, the shape constructors and transformations defined in the abstraction layer result from a generalization of the functionality common to the most used CAD applications. Using the API provided in the abstraction layer to write programs is the key factor for virtualizing the different back-ends, resulting in portable generative design programs. Nevertheless, the definition of this abstraction layer prevents the use of any CAD-specific functionality, making it impossible to write portable programs that take advantage of that functionality.

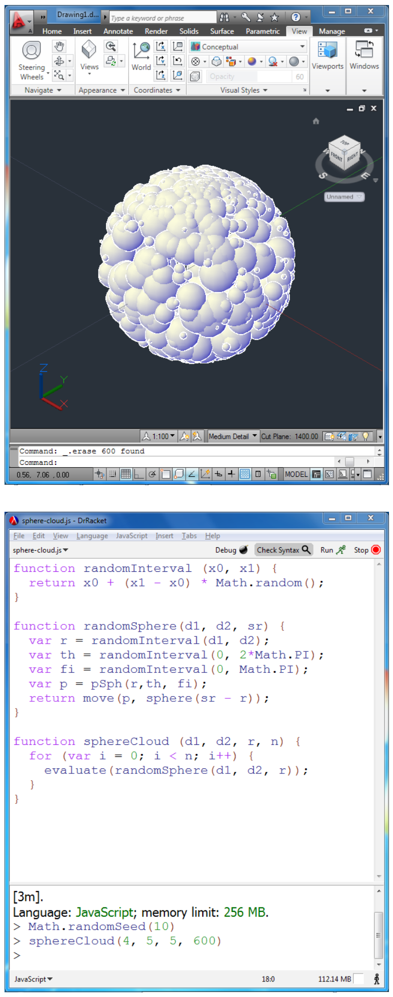
With this situation in mind, we provide a choice: users can either write portable programs using the API defined in the abstraction layer or they can take advantage of CAD-specific functionality. The latter choice may result in programs that are not portable, however, it makes Rosetta available to a broader audience of Designers.

3.3 INTERMEDIATE PROGRAMMING LANGUAGE

Since the front-ends use the IPL as a compilation target, it is possible to write programs that contain modules written in different programming languages. For the user of a module it is not relevant to know in which language it was written but only how to use the exported functionality. This allows a significant freedom both to the module writer as well as to the module user.

3.4 SEMANTIC DIFFERENCES IN BACK-ENDS

Even though there is an abstraction layer that allows transparent access to several CAD applications, these applications still present semantic differences. For



```
Command: _.erase 600 found
Command:
```

```
function randomInterval (x0, x1) {
  return x0 + (x1 - x0) * Math.random();
}

function randomSphere(d1, d2, sr) {
  var r = randomInterval(d1, d2);
  var th = randomInterval(0, 2*Math.PI);
  var fi = randomInterval(0, Math.PI);
  var p = pSph(r,th, fi);
  return move(p, sphere(sr - r));
}

function sphereCloud (d1, d2, r, n) {
  for (var i = 0; i < n; i++) {
    evaluate(randomSphere(d1, d2, r));
  }
}
```

```
[3m].
Language: JavaScript; memory limit: 256 MB.
> Math.randomSeed(10)
> sphereCloud(4, 5, 5, 600)
>
```

Fig. 2

Figure 2. *Sphere Cloud program in Rosetta using the JavaScript front-end and the AutoCAD back-end*

example, a solid sphere in AutoCAD is the space enclosed by the sphere surface, whereas, the same sphere in Rhinoceros 3D is simply 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 Rhinoceros 3D, the same operation results in an error because the two spherical surfaces do not intersect. In order to solve these problems, each back-end understands the limitations of the corresponding CAD application and provides ways around those limitations. As an example, reconsider the shape in **Figure 1**. This object can be produced by subtracting cones from a hollow sphere. However, when using the Rhinoceros 3D back-end, 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 final shape.

3.5 AUTOCAD AND RHINOCEROS 3D BACK-ENDS

As an example of the use of the AutoCAD back-end, consider the Sphere Cloud program and the Shelter program.The Sphere Cloud program creates a number of randomly positioned spheres inside a spherical region. **Figure 2** illustrates the interaction between the JavaScript front-end and the AutoCAD back-end. The main function, "sphere-cloud", is fully parameterized, such that it is possible to specify the number of spheres inside the spherical region, as well as the minimum and maximum radii for the spheres and the radius for the spherical region.

The Shelter program subtracts a sphere from a set of tubes placed in an orthogonal grid layout. **Figure 3** illustrates the execution of this program.

Because Rosetta focuses on the portability of its programs, we show that the same programs that generated the Sphere Cloud and the Shelter models in AutoCAD generate the exact same models in Rhinoceros 3D **(Figures 4, 5)**. It should be noted that **Figure 2** represents the Shelter program in JavaScript, while **Figure 4** represents the semantically equivalent program in Racket.

3.6 ROSETTA AUTOLISP

AutoLISP is one of the most used programming languages in generative design. However, this language has a few shortcomings which we have overcome in our implementation for the Rosetta programming environment. For example, one of the most frequent mistakes is to accidentally misspell the name of some variable or function, but AutoLISP treats the use of undefined names as automatically bound to a default value, meaning that it will silently accept the mistake. Usually, 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 to the user the cause of the error even before running the program.

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

3.7 ROSETTAFLOW

Our implementation of RosettaFlow is still in the early stages of development. The purpose of this prototype is to demonstrate the capabilities of Rosetta in incorporating not only textual but also visual programming languages. At the moment, RosettaFlow presents a very simple interface and a small set of operations, but we plan to extend it with more functionality. Nevertheless, RosettaFlow is fully functional and integrated with Rosetta. As an example, consider **Figure 6** which illustrates the schematic representation of the Sphere Cloud program. Due to the size of the visual program, only a fragment is illustrated.

One important feature of RosettaFlow programs is that they are compiled to the IPL and executed in DrRacket. At the moment, the execution workflow of RosettaFlow requires manual intervention to compile and execute the generated program, but we intend to automate this process.

4 Related work

In this section, we present other programming environments that served as inspiration to Rosetta.

4.1 VISUALSCHEME

VisualScheme is a CAD programming language (Leitão 2010) that envisages a pedagogical approach to programming and its integration in the architecture curriculum. Following a series of pedagogical studies (Chen 1992; Berman 1994; Felleisen 2002, 2004; Marceau 2011), VisualScheme relies on the Scheme programming language as a teaching tool for an audience without a background in Computer Science.

Rosetta is a direct descendent of VisualScheme, maintaining the same pedagogical concerns, but with a different approach. We still defend the use of Scheme for the general audience of Designers that do not have a background in Computer Science. However, we include additional programming languages for those who have programming experience, so that they can choose the one that is most suitable for their program. As a result, CAD applications no longer impose the provided programming language to their users. Other benefits include the automatic compatibility of the chosen language with the programming environment and the portability of the programs written in it.

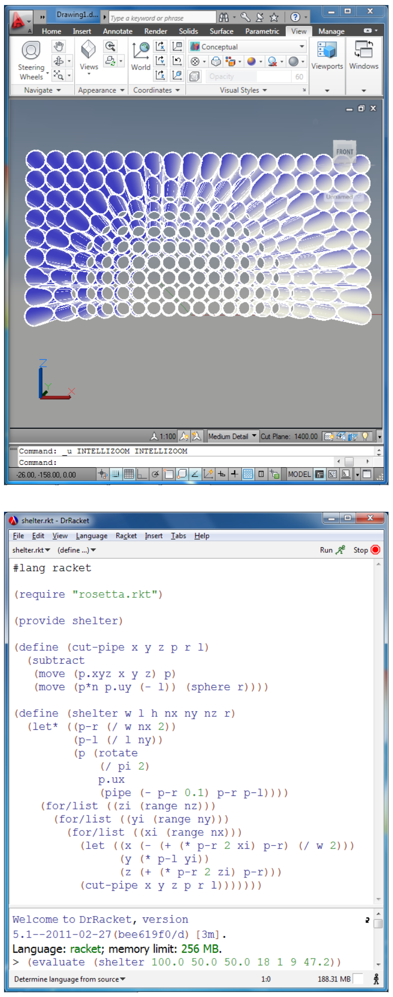
4.2 GRASSHOPPER AND GENERATIVE COMPONENTS

Grasshopper and Generative Components are visual approaches to generative design. These languages have a visual syntax and users create programs using boxes, which represent operations, and connectors, which represent data flow. These visual programming languages make it easier to avoid errors associated with writing code in textual form. However, complex programs tend to be very large graphs that are hard to read and understand, and even harder to maintain.

Grasshopper presents not only with an appealing interface but also with a rich set of geometric functions which operate transparently with values and lists of values. For example, the translation operation can be used both on a single sphere and on a list of spheres. This transparent mapping of operations over multiple values allows for separating the chain of operations from the input values, such that the same program that is used for transforming a single shape can also be used for multiple shapes.

Nevertheless, this linguistic feature may result in difficult semantics. For example, using the addition operation from Grasshopper, users can choose different data matching strategies for the input values, which include shortest list, longest list and cross reference. These strategies introduce confusing semantics and may lead to program defects. We are currently studying a way of introducing this linguistic feature in Rosetta while avoiding the need for data matching strategies.

While Grasshopper mainly uses the visual representation of the generative design program and the geometry displayed in the screen, GenerativeComponents (GC) uses different representations for the same program, including visual and textual ones (Menges 2010). These representations complement each other and are linked, such that changes in one representation are propagated to the others. Moreover, GC is used not only to create geometry but also to implement several other processes related to architecture and civil engineering, such as measurement, evaluation, configuration and fabrication. At the moment, the purpose of Rosetta is to develop generative design programs, so the focus will remain on producing geometric models, leaving other processes to be supported by the CAD applications used as back-ends. Furthermore, it is possible to have different representations of the same program in Rosetta. Examples of these representations include RosettaFlow and the IPL. These representations are also linked, such that it is possible to automatically generate the IPL code of a RosettaFlow program.



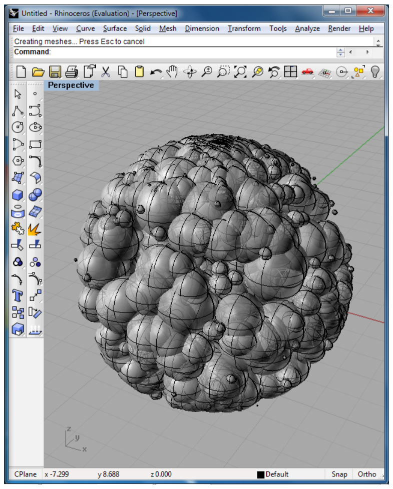
```
Command: _u INTELLIZOOM INTELLIZOOM
Command:
```

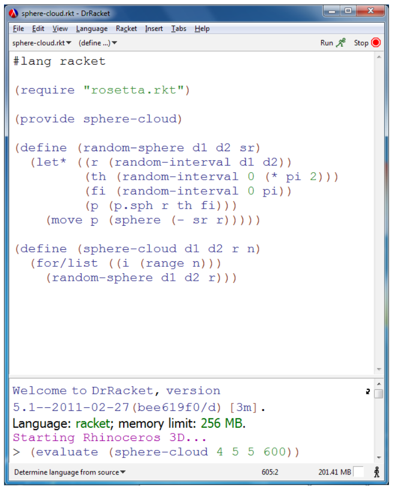
```
#lang racket

(require "rosetta.rkt")

(provide shelter)

(define (cut-pipe x y z p r l)
  (subtract
   (move (p.xyz x y z) p)
   (move (p*n p.uy (- l)) (sphere r))))

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

```
Welcome to DrRacket, version
5.1--2011-02-27(bee619f0/d) [3m].
Language: racket; memory limit: 256 MB.
> (evaluate (shelter 100.0 50.0 50.0 18 1 9 47.2))
```

Fig. 3

Figure 3. *Shelter program in Rosetta using the Racket programming language and the AutoCAD back-end*


Untitled - Rhinoceros (Evaluation) - [Perspective]
File Edit View Curve Surface Solid Mesh Dimension Transform Tools Analyze Render Help
Creating meshes... Press Esc to cancel
Command:
Perspective
CPlane x -7.299 y 8.688 z 0.000 Default Snap Ortho


sphere-cloud.rkt - DrRacket
File Edit View Language Racket Insert Tabs Help
sphere-cloud.rkt (define ...) Run Stop
Determine language from source 605:2 201.41 MB

```
#lang racket

(require "rosetta.rkt")

(provide sphere-cloud)

(define (random-sphere d1 d2 sr)
  (let* ((r (random-interval d1 d2))
         (th (random-interval 0 (* pi 2)))
         (fi (random-interval 0 pi))
         (p (p.sph r th fi)))
    (move p (sphere (- sr r)))))

(define (sphere-cloud d1 d2 r n)
  (for/list ((i (range n)))
    (random-sphere d1 d2 r)))
```

```
Welcome to DrRacket, version
5.1--2011-02-27(bee619f0/d) [3m].
Language: racket; memory limit: 256 MB.
Starting Rhinoceros 3D...
> (evaluate (sphere-cloud 4 5 5 600))
```

Fig. 4

4.3 PROCESSING

Processing (Reas 2007) is a programming language and environment specialized for the production of images, animations and interactions. Initially designed for sketching, it has grown to become a professional tool. The language is a simplified version of Java and is capable of generating applets for Java-enabled browsers or standalone applications that run on the major operating systems.

Rosetta differs from Processing in that it is not restricted to a single front-end language and was designed to integrate well with CAD applications. This is a direct consequence of the decision to target architectural work that, nowadays, relies critically on CAD applications. However, we plan to include a dedicated front-end that emulates the Processing language so that architects already used to Processing can easily migrate to Rosetta.

5 Conclusions

We have shown that portability in generative design programs is a very important requirement. Unfortunately, the current programming environments provided by CAD applications do not promote portable programs. In order to solve this problem, we propose Rosetta, a programming environment that integrates several programming languages, in which users write their programs with multiple CAD applications, in which the geometric models are created. As a result, users can explore different combinations of front-ends and back-ends in order to find the one that is most suitable for their needs. Moreover, they can use programs written by other designers, possibly written for a different CAD application and with a different language.

By providing a direct connection between a chosen language and a chosen CAD application, Rosetta dispenses with the error-prone, manual migration of models between different CAD applications, thus improving the designer's workflow. This workflow can be further simplified by the addition of specific back-ends (e.g., for performance analysis).

We are now currently evaluating Rosetta with a group of architecture students that volunteered as beta-testers. In future work, we plan to expand Rosetta in several ways: (1) the linguistic constructs of the IPL; (2) the supported front-ends and back-ends (e.g., Processing and Revit); and (3) RosettaFlow. It is our intention that our work creates an online community of programmers in the area of generative design in which they can share and develop software in a collaborative fashion.

Figure 4. *Sphere Cloud program in Rosetta using the Racket programming language and the Rhinoceros 3D back-end*

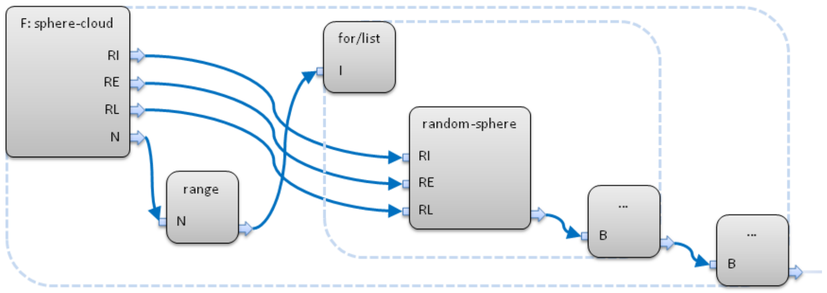


Fig. 6

References

Berman, A. M. 1994. Does Scheme enhance an introductory programming course?: some preliminary empirical results. In *ACM SIGPLAN Notices*, vol. 29, Issue 2, 44-48.

Chen. N. M. 1992. High School Computing: The inside Story. In *The Computing Teacher*, vol. 19, no. 8, 51-52, International Society for Technology in Education.

Döllner, J., and K. Hinrichs. 2000. A Generalized Scene Graph. In *Vision, Modeling and Visualization* 2000, 247–254, Saarbrücken, Germany, Akademische Verlagsgesellschaft.

Felleisen, M., R. Findler, M. Flatt, and S. Krishnamurthi. 2002. The Structure and Interpretation of the Computer Science Curriculum. In *Journal of Functional Programming*, vol. 14, issue 4, 365-378.

Felleisen, M., R. Findler, M. Flatt, and S. Krishnamurthi. 2004. The TeachScheme! Project: Computing and Programming for Every Student. In *Computer Science Education*, vol. 14, issue 1, 55–77.

Findler, R., J. Clements, C. Flanagan, M Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. 2002. Dr Scheme: a programming environment for Scheme. In *Journal of Functional Programming*, vol. 12, issue 2, 159–182.

Flatt, M., R. Findler, S. Krishnamurthi, M. Felleisen. 1999. Programming Languages as Operating Systems (or Revenge of the Son of the Lisp Machine). In *ACM SIGPLAN International Conference on Functional Programming*, 138-147.

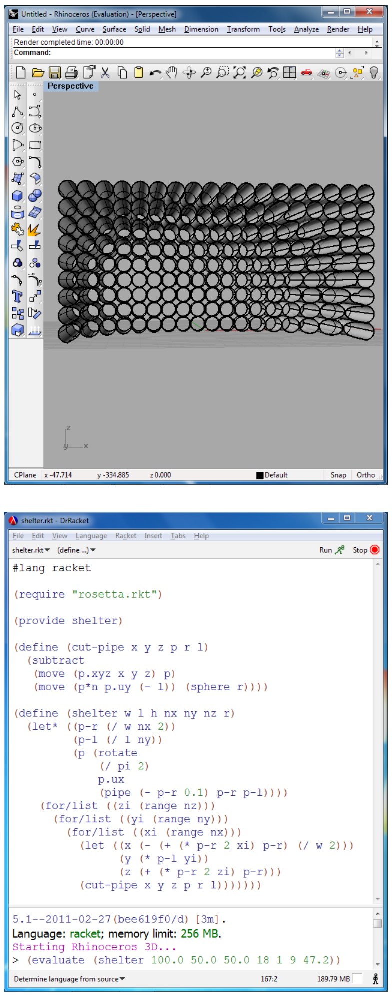
Kelsey, R., W. Clinger, and J. Rees. 1998. Revised 5 Report on the Algorithmic Language Scheme. In *ACM SIGPLAN Notices*, vol. 33, issue 9, September, 1998

Marceau, G., K. Fisler, and S. Krishnamurthi. 2011. Measuring the Effectiveness of Error Messages Designed for Novice Programmers. In *SIGCSE '11 Proceedings of the 42nd ACM technical symposium on Computer science education*, 499-504.

Leitão, A., F. Cabecinhas, and S. Martins. 2010. Revisiting the Architecture Curriculum: The Programming Perspective, In *28th eCAADe Conference Proceedings: Future Cities*, 81-88.

Menges, A. 2010. Instrumental Geometry. In *Corser, R.(ed.), Fabricating Architecture: Selected Readings in Digital Design and Manufacturing*, Princeton Architectural Press, 2010.

Reas, C., B. Fry, and J. Maeda. 2007. Processing: *A Programming Handbook for Visual Designers and Artists*. The MIT Press.


Untitled - Rhinoceros (Evaluation) - [Perspective]
File Edit View Curve Surface Solid Mesh Dimension Transform Tools Analyze Render Help
Render completed time: 00:00:00
Command:
Perspective
CPlane x -47.714 y -334.885 z 0.000 Default Snap Ortho
shelter.rkt - DrRacket
File Edit View Language Racket Insert Tabs Help
shelter.rkt (define ...)
Run
Stop
Determine language from source
167:2
189.79 MB

```
#lang racket

(require "rosetta.rkt")

(provide shelter)

(define (cut-pipe x y z p r l)
  (subtract
   (move (p.xyz x y z) p)
   (move (p*n p.uy (- l)) (sphere r))))

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

```
5.1--2011-02-27(bee619f0/d) [3m].
Language: racket; memory limit: 256 MB.
Starting Rhinoceros 3D...
> (evaluate (shelter 100.0 50.0 50.0 18 1 9 47.2))
```

Fig. 5

Figure 5. *Shelter program in Rosetta using the Racket programming language and the Rhinoceros 3D back-end*

Figure 6. *RosettaFlow schematic for the Sphere Cloud program*