Extending Processing to CAD Applications

Hugo Correia[1], António Leitão[2]
[1,2]INESC-ID, Técnico Lisboa
[1,2]{hugo.f.correia|antonio.menezes.leitao}@tecnico.ulisboa.pt

The Processing language was created to teach programming to the design, architecture, and electronic arts communities. Despite its success, Processing has limited applicability in the architectural realm, as no CAD (Computer-Aided Design) or BIM (Building Information Modeling) application supports Processing. As a result, architects that have learnt Processing are unable to use the language in the context of modern, script-based, architectural work. This work joins Processing with the world of CAD or BIM applications, creating a solution that allows architects to prototype new designs using Processing and generate results in a CAD or BIM application. To achieve this, we developed an implementation of Processing for the Rosetta programming environment, allowing Processing scripts to generate 2D and 3D models in a variety of CAD or BIM applications, such as AutoCAD, Rhinoceros3D, SketchUp, and Revit.

Keywords: *Processing Language, Processing Script, Generative Design, 3D Modeling*

INTRODUCTION

Many technological solutions have influenced areas, such as design and architecture, where they have explored different tools to aid the creation of new designs. With the help of powerful Computer-Aided Design (CAD) and Building Information Modeling (BIM) environments, these communities have been able to develop increasingly complex and innovative designs in a more productive manner. However, in some cases, a manual design process in a CAD tool is not sufficient, as trivial geometry that is constantly repeated in a design could be easily generated using automated mechanisms. For this reason, these systems quickly found the need to provide Application Program Interfaces (APIs) to their users; allowing them to explore programming in the context of a CAD or BIM application. For example, AutoCAD offers an API usable from C++, AutoLisp, and VBA; while Rhinoceros 3D supports RhinoScript, Python, and Grasshopper.

A novel approach that has been increasingly explored is Generative Design (GD), which takes advantage of programming solutions to enhance the design creation process (McCormack et al. 2004). Using GD, artists, designers, and architects can test different design combinations, fostering their capacity for innovation and creativity, and, at the same time, providing a low-cost solution to explore different design prototypes. Due to this phenomenon, several programming languages and Interactive Development Environments (IDEs) have been adapted, or specifically developed, to fulfil the needs of these artistic communities. A well-known example is Processing, created to provide a pedagogical and highly visual

approach to programming.

Processing is one of the most successful efforts to bring programming into the realm of design, art, and architecture. There are numerous examples of the use of Processing in digitally-generated paintings, tapestries, photographs, installations, choreographies, visualizations, simulations, sculptures, music, games, etc. Unfortunately, in architecture, Processing is much less used than other programming languages such as AutoLISP, VisualBasic, RhinoScript, Grasshopper, Ruby, or Python, as none of the professional CAD tools that are typically used by architects (e.g. AutoCAD, Rhinoceros 3D, ArchiCAD, etc.) include Processing as one of its scripting languages.

In this paper, we present a solution that allows Processing to be used in the context of a CAD or BIM application. Our solution makes the following contributions: (1) we allow Processing to be used in the context of a CAD or BIM environment, extending the language with 2D and 3D modeling primitives that are adequate for architectural work; (2) we allow Processing scripts to take advantage of modules and libraries in other languages developed for architectural work; finally, (3) we provide a development environment equipped with an interactive evaluator for Processing that is suitable for experimental design work. Our solution is based on an implementation of Processing for Rosetta, an IDE for GD that supports programming in a variety of languages, including AutoLISP, Racket, JavaScript, and Python, while allowing the generation of architectural models in a variety of CAD tools, including AutoCAD, Rhinoceros 3D, and SketchUp.

The following sections describe the Processing programming language and our Processing implementation developed for Rosetta. We explain our proposed extensions that make Processing suitable for architectural work in a CAD or BIM application, and evaluate examples that illustrate the use of Processing in the context of script-based architectural work.

THE PROCESSING LANGUAGE

The Processing language (Reas and Fry 2007) was developed at MIT Media Labs and was heavily inspired by the *Design by Numbers* project (Maeda 1999), with the goal to teach computer science to artists and designers with no previous programming experience. The language has grown over the years with the support of an academic community, which has written several educational materials demonstrating how programming can be used in the visual arts. Processing is based on the Java programming language, being statically typed and sharing Java's object-oriented capabilities. This design decision was due to Java being a mainstream language used by a large community of developers. Moreover, as Processing was developed to promote programming literacy in design and architecture, its syntax enables users to easily migrate to other languages that share Java's syntax, such as C, C++, C#, or JavaScript.

Notwithstanding, as Java is a fully featured multipurpose programming language, it requires users to grasp a considerable amount of knowledge which can be irrelevant for users that want to develop simple scripts for visual purposes. As a result, several simplifying features were introduced in Processing that allow users to quickly test their design ideas without requiring extensive knowledge of the Java language. For instance, to execute code, Java requires users to define a public *class* and a public *main* method. Processing simplifies this by removing these requirements, allowing users to write simple scripts (i.e. simple sequences of statements) that produce designs without the boilerplate code that is needed in Java.

Fundamentally, Processing enables users to gradually introduce more complex tools to their programming toolbox. Users start by learning to develop simple scripts, quickly visualizing their designs. After some time, as there is so much one can achieve using simple scripts, they shift into using higher levels of abstraction, such as functions and classes, which enable them to create more complex designs in an easier way. Finally, they can shift into a full Java style mode of programming taking advan-

tage of its object-oriented features, allowing the use of libraries and capabilities that the Java environment provides.

On top of being a programming language, Processing offers the Processing Development Environment (PDE), an IDE for the Processing language. In the PDE (illustrated in figure 1), users can develop their scripts using a simple and straightforward environment, which is equipped with a tabbed editor and IDE services, such as syntax highlighting and code formatting. Moreover, Processing users can create custom libraries and tools that extend the PDE with additional functionality, such as networking, PDF rendering support, color pickers, sketch archivers, etc.

The main advantage of Processing is its ability to help users quickly test and visualize their creations, incrementally evolving them. This is possible due to Processing's rendering system, which is based on OpenGL, thus allowing designers to rapidly render complex and computationally intensive designs that would take the typical CAD system much longer to produce. Furthermore, Processing offers users a set of modeling tools that are specially tailored for visual artists, namely 2D primitives, and a set of built-in classes specifically created for design. These primitives demonstrate the advantages of using Processing as a tool for testing and developing design prototypes.

Unfortunately, Processing fails to provide a connection with CAD or BIM applications. The Processing community is therefore limited to Processing's rendering system, that is based on OpenGL, lacking the CAD modeling features which are used by architects in their day-to-day work. Our solution joins both worlds, allowing architects and designers to prototype new designs using Processing; while generating the results in a CAD or BIM environment. Therefore, a wide community of artistic programmers can explore the modeling capabilities of a CAD environment using the Processing language. Moreover, by connecting Processing to CAD applications, architects that already use CAD in their day-to-day work can take advantage of a pedagogic programming language in a CAD application and explore its wide range of examples and educational materials.

EXTENDING PROCESSING TO CAD APPLICATIONS

To make Processing more useful for the large community of designers and architects that have learned the language, we extended Rosetta to support the Processing language. Rosetta (Lopes and Leitão 2011) is an IDE for GD, implemented using the Racket lan-

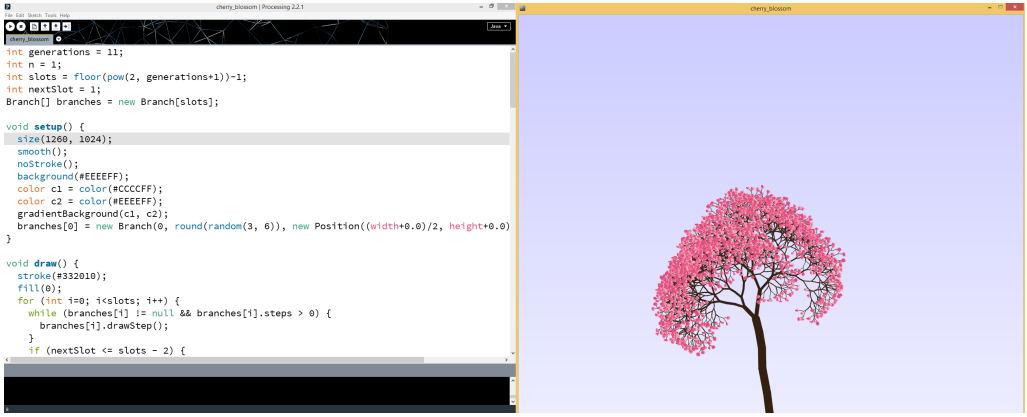


Figure 1
The Processing Development Environment

guage (Flatt 2012) and taking advantage of the pedagogical capabilities of the DrRacket programming environment (Findler et al. 2002). Compared to other development environments, such as Grasshopper, the main advantage of Rosetta is the emphasis on choice and portability; scripts can be written using one of the different supported languages (currently, AutoLisp, JavaScript, Scheme, Racket, and Python) and generate identical models in any of the supported CAD applications (currently, AutoCAD, Rhinoceros 3D, SketchUp, and Revit). This means that, by connecting Processing to Rosetta, it becomes possible for architects that have learned the language to use it in the context of modern, script-based, architectural work.

Our implementation is based on a traditional compiler pipeline composed by: tokenization, parsing, static analysis (including lexical scope analysis and type-checking) and, finally, code generation phases. Our approach was to develop Processing as a new Racket language module (Tobin-Hochstadt et al. 2011), extending Rosetta with Processing and integrating Processing with DrRacket - Racket's pedagogic IDE.

Firstly, the compilation process starts by reading Processing source code and transforming it into tokens. Subsequently, these tokens are given to the parser that generates an intermediated representation of the original Processing source code in the form of an Abstract Syntax Tree (AST). This was accomplished by developing lexical and syntactical specifications that adhere to Processing's syntax rules.

Secondly, the generated AST is analyzed, taking into consideration the language definitions of both Processing and Racket, particularly, its scoping and typing rules. We start by making a scope analysis of the AST, by identifying when definitions (i.e. variables, functions, classes) are created, adding them to a custom scoping mechanism that saves the type declarations provided in the definitions. After this process, we check the types of each node of the AST, until the full AST is traversed. Each node is tested for type correctness and, when necessary, its type is promoted. In the event that types do not match, a type error is produced, informing the user of the location of the type error.

Finally, after the AST is fully analyzed and type-checked, semantically equivalent Racket code is generated and loaded into Racket's virtual machine, where it is executed.

PROCESSING FOR ARCHITECTURAL WORK

Besides supporting the traditional syntax and semantics of the Processing language, our implementation extends Processing to: 1) support a wider set of modeling primitives, 2) access libraries that are written in other programming languages, and 3) take advantage of an IDE tailored for the Processing language that offers interactive evaluation.

Extensions for 3D modeling are essential for Processing to be used in the context of professional CAD tools. The original Processing language only provides very basic primitives for 3D modeling, namely, a *box*, and a *sphere* (with variable resolution). Users that need to model other complex shapes have to explicitly create them using *beginShape* and *endShape* primitive operations. These operations require the user to explicitly define the set of vertexes that describe each surface of the shape and how they are connected (e.g. using lines, triangles, or quadrangular strips).

Unfortunately, this modeling process is very unnatural for designers and architects. Therefore, as the current Processing environment is rather poor in these modeling operations, we augmented it with a large set of primitive shapes (e.g., cylinder, cone and cone frustum, regular pyramid, torus, etc), boolean operations (e.g., union, intersection, and difference), and many more (e.g., extrusion, sweeping, and lofting). These primitive shapes and operations significantly reduce the effort needed to generate complex designs.

Moreover, although Processing provides some abstractions, such as *PVector* to abstract the use of vectors, it does not have an appropriate abstraction

for coordinates. Usually, users solve this issue by passing points around in a array or even by passing each coordinate point information individually, resulting in long function headers and additional verbosity in the program. On the other hand, Rosetta has custom mechanisms to abstract coordinate systems, namely cartesian (*xyz*), polar (*pol*), and cylindrical (*cyl*) which can be used and combined interchangeably. As a result, these abstractions (*xyz*, *pol*, and *cyl*) are made available in our system, so that users can take advantage of them in their designs, enabling them to use an intuitive and simple mechanism to manipulate and build their models.

Our work also has the advantage of allowing users to explore libraries written in another language. This ability not only allows designers and architects to use previously created libraries for GD (are already used with Rosetta); but also allows us to access libraries and frameworks of any language that Rosetta supports (e.g. Racket, JavaScript, and Python), offering an opportunity for users to explore other approaches to programming.

Finally, an important feature that is essential for Processing users is the existence of IDE for the language. Therefore, as Rosetta is integrated in DE (i.e. DrRacket), we adapted it to support the Processing language. As illustrate in figure 2, our system is very similar to the PDE, sharing features such as syntax highlighting, a tabbed editor, and code formatting. Additionally, we provide an interactive evaluation mechanism which is unavailable in the original Processing environment, allowing users to evaluate small fragments of Processing programs in a Read-Eval-Print-Loop (REPL). This is an important feature for incremental development of programs, as it allows quick experimentation and validation of scripts that are being developed; which is particularly important for designers and architects that want to visualize the evolution of their design as the script is being written.

EVALUATION

The main reason for the development of our Processing implementation was to promote its use in the context of architectural problems. In this section, we present a few examples that demonstrate this use.

Many of Processing's 2D primitives are directly available in Rosetta, therefore primitives such as *line*, *ellipse*, *arc*, *rect*, *quad*, and *triangle* were easily implemented, requiring only small adjustments. For example, Processing's *triangle* primitive is implemented using Rosetta's *polygon* primitive (which receives a list of points, creating a polygonal shape), that in this case receives the triangle's three points.

As previously mentioned, Processing provides a rather poor set of 3D modeling primitives, therefore we augmented Processing's primitive set with addi-

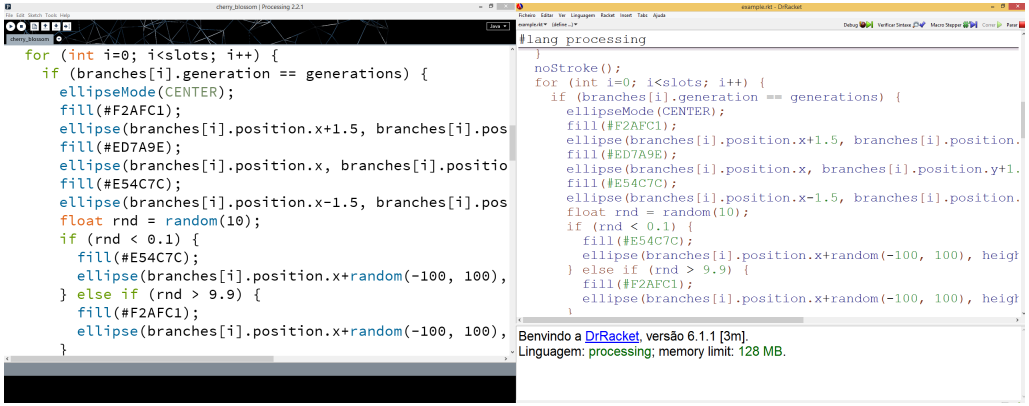


Figure 2
The Processing Development Environment and DrRacket using Processing

tional of 3D modeling primitives provided by Rosetta, such as *cylinder*, *cone*, *pyramid*, or *torus*. Furthermore, Processing offers users a set of transformation primitives such as *rotate*, *scale*, and *translate* to transform and manipulate their shapes. Using Rosetta, we can expand this set of transformations with primitives such as *union*, *intersection*, *subtraction*, *loft*, or *sweep*, which are heavily used in architectural work. For instance, figure 3 illustrates the use of the *union*, *intersection*, and *subtraction* primitives with a *box* and a *sphere*.

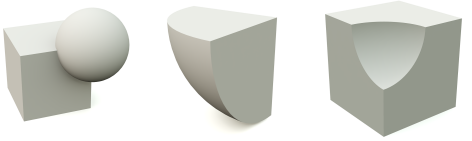


Figure 3
union, intersection, and subtraction of a sphere with box

Furthermore, to illustrate the capabilities of our implementation, consider the following Processing code:

```
float da = PI/6, db = PI/5;

void tree(float x, float y,
          float l, float a){
  float x2 = x - l*cos(a),
        y2 = y - l*sin(a);
  line(x,y,x2,y2);
  if (len < 10) {
    ellipse(x2,y2,0.6,0.6);
  } else {
    tree(x2,y2,random(.7,.8)*l,a+da);
    tree(x2,y2,random(.7,.8)*l,a-db);
  }
}
```

This example generates a fractal tree that gradually reduces the length of each tree branch, using Processing's *line* and *ellipse* primitives to create its branches and leaves. Note that the rate by which the length of each branch is controlled by a randomly generated number.

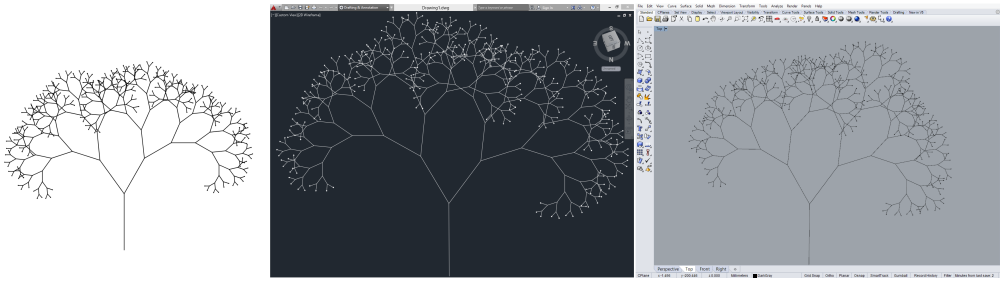
Our goal was to use the same Processing code, and, with it, generate the same drawing in different CAD applications. Figure 4 illustrates this behaviour, showing the use of *tree* in the original Processing environment, AutoCAD and Rhinoceros 3D. The only change made to the original Processing invocation of *tree* was to specify which CAD back-end to use. This is accomplished by executing the *backend* function, that allows the architect to specify the CAD environment to use in his design. Note that the generated drawings have slight variations, because of the *random* function which is used to produce shorter tree branches.

Notwithstanding, these 2D illustrations have limited applicability for architectural work. They can however be a base idea to explore in other designs. For instance, consider the following Processing code:

```
float da = PI/4;
void tree3D(float x, float y, float z,
       float l, float a, float r){

  float x2 = x + l*cos(a)*sin(da),
        y2 = y + l*sin(a)*cos(da),
        z2 = z + l*cos(da),
  float r2 = r * 0.55;

  coneFrustum(xyz(x,y,z),r,
              xyz(x2,y2,z2),r2);

  if (l < 7) {
    box(xyz(x2-3,y2-3,z2-0.5),6,6,1);
  }  else {
    float l2 = l*0.7, a2 = a + PI;
    tree(x2,y2,z2,l2,a2*1/4,da,r2);
    tree(x2,y2,z2,l2,a2*3/4,da,r2);
    tree(x2,y2,z2,l2,a2*5/4,da,r2);
    tree(x2,y2,z2,l2,a2*7/4,da,r2);
  }
}
```

The previous code, shows an example of how we can quickly migrate from a simple 2D drawing to a more complex and architecturally useful creation. In this case, the branches were modeled using the *coneFrustum* primitive, while the endings were modeled using a *box*, thus creating a tree inspired column. Note that the *coneFrustum* primitive is another example of a primitive that our implementation brings to Processing's modeling set. Also note that *coneFrustum* and

Figure 4
2D fractal tree generated in Processing, AutoCAD, and Rhinoceros 3D



box where adapted to support Rosetta's coordinates abstractions (in this case, the *xyz* primitive). This allows users to take advantage of a set of coordinate abstractions which allows them to manipulate their models efficiently.

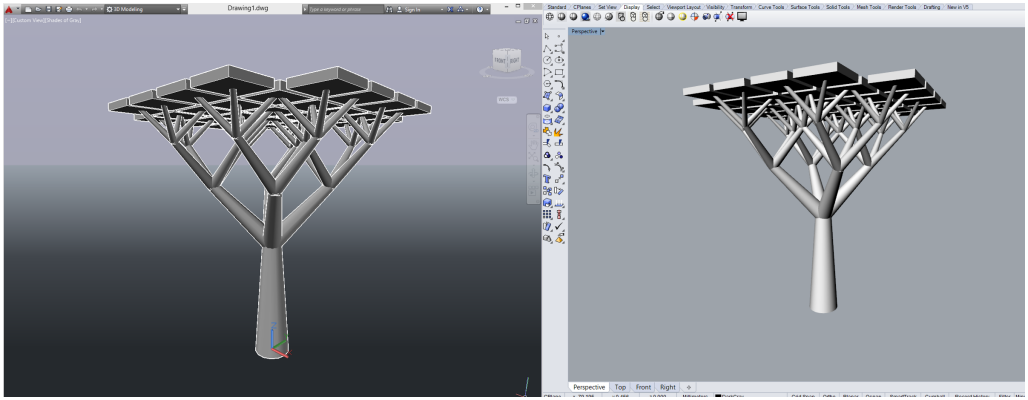
Again, using the *backend* primitive, we can generate these models in different CAD applications. Figure 5 shows the execution of *tree* (adapted for 3D) in AutoCAD and Rhinoceros 3D, illustrating that, with little effort, architects can run Processing scripts in different CAD or BIM application.

Finally, we demonstrate in figure 6, another example of our Processing implementation; but that uses previously defined libraries that are written in another language. To produce this example, our Processing code accesses a library written in the Racket language that is capable of generating highly parametric skyscrapers, allowing users to specify, in Processing, the height, radius, rotating factor, etc. of each skyscraper.

RELATED WORK

The Processing language has been used in other contexts outside the original Processing environment. Processing.js [1] is a JavaScript implementation of Processing, where developers can use Processing's approach to design 2D and 3D geometry in a HTML5 compatible browser. Similarly, P5.js [2] is a recent JavaScript library that allows users to program in the web using Processing's metaphors. Similarly, Ruby-Processing [3] implements Processing graphical primitives in the context of the language Ruby, while Processing.py [4] does the same for the Python language. One important difference is that Process-

Figure 5
3D column tree generated in AutoCAD and Rhinoceros 3D



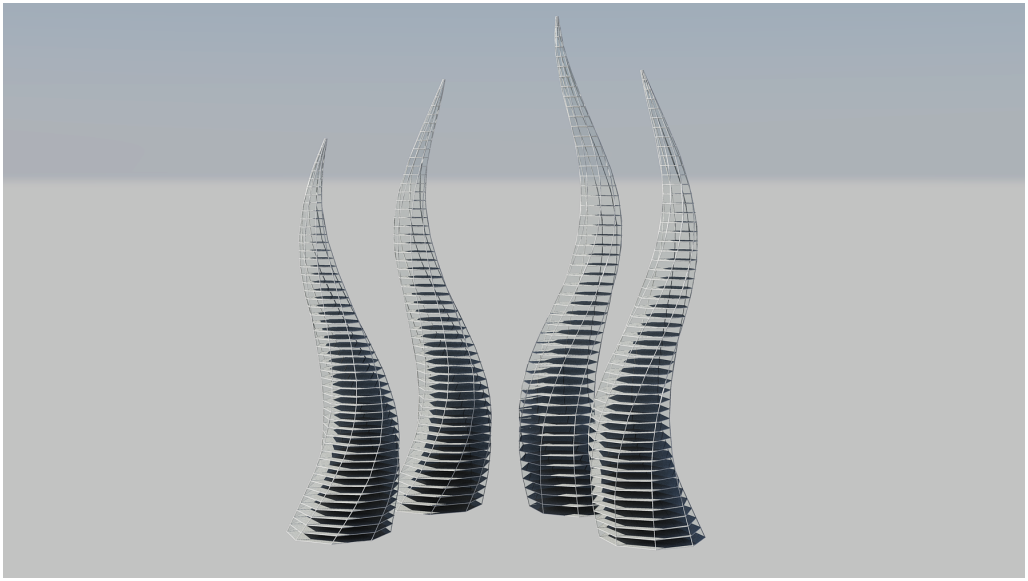


Figure 6
Skyscrapers rendered in AutoCAD using external libraries

ing.py is fully integrated within Processing's development environment while Ruby-Processing does not have a dedicated IDE. On the other hand, Ruby-processing offers sketch watching (code is automatically run when new changes are saved) and live coding (the sketch is updated during code changes).

Both Processing.js and P5.js have attempted to bring Processing's power to the Web, while Ruby-Processing and Processing.py are clear examples of Processing reaching out to new languages. However, none of them address the needs of architects that want to work in the context of a typical professional architectural tool, such as AutoCAD, Rhinoceros 3D, or Revit.

Finally, OBJExport [5] is a library that export meshes from Processing to OBJ or X3D files. It can export color meshes with triangle and quadrangle shaped faces as an OBJ or X3D with a PNG texture map, that can then be imported into some CAD applications. However, using this approach, we lose the interactivity of programming directly in a CAD application, as users have to generate and import the OBJ file each time the Processing script is changed, creating a cumbersome workflow. Moreover, as shapes are transformed to meshes of triangles and points, there is a considerable loss of information, as the semantic notion of the shapes is lost.

CONCLUSIONS

Implementing Processing for traditional CAD applications benefits architects and designers by allowing them to take advantage of Processing's visual capabilities. Augmenting Processing with new design paradigms and abstractions, namely 3D modeling primitives (torus, cone, cylinder, etc.) and transformations (union, subtraction, loft, etc.), presents a strong reason for the architecture community to take advantage of our solution. Our implementation clearly empowers Processing's environment with modeling features that help Processing users create new designs using a more expressive modeling approach. Moreover, architects can easily ac-

cess and combine several modeling primitives, enabling the creation of new designs that would be much harder to achieve in the original Processing environment.

Nowadays, the Processing language and design approach is also being explored with other programming languages (e.g. Ruby, Python, and JavaScript). Our implementation also encompasses this feature, as it allows us to explore and combine Processing with any of the different languages that are provided by Rosetta, namely Scheme, Python, JavaScript, Racket, and AutoLISP.

The PDE is an essential feature of the Processing language, as it reduces beginners' programminglearning curve. Our implementation offers a similar solution, by adapting DrRacket to support our implementation, providing a simplified editor and development environment, which is almost identical to the PDE. Additionally, we provide an important additional feature, the REPL, which is unavailable to other Processing implementations, providing users with a mechanism to quickly and interactivity test out new ideas.

Although still in the testing phase, our Processing implementation already fulfils the basic needs of architects, namely the ability to write scripts and the visualization of the results in a professional CAD application. Afterwards, our goal is to build-upon our existing work, and progressively introduce more advanced mechanisms of the language, such as inheritance and interfaces in classes.

Having Processing in Racket allows us to connect with CAD applications, taking advantage of new 3D modelling primitives tailored for architectural work while using, at the same time, an IDE tailored for the Processing language. Moreover, we have the ability of accessing libraries that are written in any language of the Racket ecosystem. For all the reasons mentioned above, our system offers compelling motives for the architecture community to explore our system in their architectural endeavors.

ACKNOWLEDGEMENTS

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013, and by the Rosetta project under contract PTDC/ATP-AQI/5224/2012.

REFERENCES

Findler, RB, Clements, J, Flanagan, C, Flatt, M, Krishnamurthi, S, Steckler, P and Felleisen, M 2002, 'DrScheme: A programming environment for Scheme', *Journal of functional programming*, 12(02), pp. 159-182

Flatt, M 2012, 'Creating languages in Racket', *Communications of the ACM*, 55(1), pp. 48-56

Lopes, J and Leitão, A 2011 'Portable generative design for CAD applications', *Proceedings of the 31st annual conference of the Association for Computer Aided Design in Architecture*, pp. 196-203

Maeda, J 1999, *Design by Numbers*, MIT Press, Cambridge, MA, USA

McCormack, J, Dorin, A and Innocent, T 2004 'Generative design: a paradigm for design research', *Futureground Conference Proceedings*, Melbourne

Reas, C and Fry, B 2006, 'Processing: programming for the media arts', *AI & SOCIETY*, 20(4), pp. 526-538

Tobin-Hochstadt, S, St-Amour, V, Culpepper, R, Flatt, M and Felleisen, M 2011 'Languages as libraries', *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pp. 132-141

[1] http://processingjs.org/

[2] http://p5js.org/

[3] https://github.com/jashkenas/ruby-processing/wiki

[4] http://py.processing.org/

[5] http://n-e-r-v-o-u-s.com/tools/obj/