Processing Architecture

António Leitão, Inês Caetano and Hugo Correia

INESC-ID/Instituto Superior Técnico, Universidade de Lisboa, Portugal
Av. Rovisco Pais, 1
1049-001 Lisboa, Portugal
antonio.menezes.leitao@tecnico.ulisboa.pt

Abstract

Programming promotes creative freedom but might require considerable effort to learn. The Processing language was created to simplify this learning process. Due to its graphical capabilities, the language has become very popular among the electronic arts and design communities. Unfortunately, this popularity could not be extended to the architecture community, which relies on traditional heavyweight CAD and BIM applications that cannot be programmed using Processing. As a result, it becomes difficult for architects to take advantage of Processing. To solve this problem, we propose an implementation of Processing that runs in the context of the most used CAD tools in architecture. Our implementation allows Processing to generate 2D or 3D models that are directly usable for architectural work. To this end, we also propose extensions to the language, including three-dimensional modelling primitives that dramatically simplify the effort needed for developing large and complex architectural models with Processing.

1. INTRODUCTION

Programming was originally considered a very specialist activity and not part of a design education [1]. Nowadays, many designers are aware of its potential and want to take advantage of it for many different purposes, including automating repetitive tasks, form finding and optimization [2]. Unfortunately, learning a programming language and the associated programming techniques is far from a trivial task and, thus, requires carefully designed languages and programming environments. In this paper, we focus on one of these languages: Processing.

Processing is an open source programming language created by Casey Reas and Benjamin Fry at MIT Media Lab [3]. Processing was developed for the electronic arts and design communities with the aim of teaching programming skills, thus improving their ability to produce creative designs.

Processing is, arguably, the most successful effort 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. In the architecture field, Processing is used in several different research areas, including urban design [4] and shape studies [5] but it is less used than other programming languages such as AutoLISP, VisualBasic, RhinoScript, Grasshopper, Ruby, or Python. The fundamental reason for this phenomena seems to rely on the needs of the architectural practice [6], particularly, on the fact that none of the professional Computer-Aided Design (CAD) and Building Information Modeling (BIM) tools typically used by architects (e.g. AutoCAD, Rhinoceros 3D, ArchiCAD, etc) include Processing as one of its scripting languages. This is an unfortunate situation, as the attractive pedagogical capabilities of Processing become a trap for the architect that, after learning the language, discovers that Processing does not natively provide many of the operations needed for architectural work, such as extruding, lofting, etc. On the other hand, although Processing can export the generated shapes (polygons, boxes, spheres, etc.) to DXF, a format understood by many CAD applications, there is a considerable loss of information. For example, a shape created with the *sphere* function will be exported as a large arrangement of triangles, rather than as a single object.

In order to overcome these obstacles, the architect needs to learn additional Processing libraries or he is forced to learn a completely different programming language that is supported by his preferred CAD application.

In this paper, we present a solution to this problem. Specifically, we make the following contributions: (1) we extend Processing with two- and three-dimensional modelling primitives adequate for architectural work; (2) we extend Processing with the ability to use modules and libraries that were developed in other programming languages; (3) we provide an interactive evaluator for Processing that is suitable for experimental design work, and, finally, (4) we connect Processing with the most used CAD tools in Architecture, allowing the generated models to be directly used for architectural work. Our solution is based on an implementation of Processing for Rosetta, an Interactive Development Environment (IDE) for Generative Design (GD) that supports programming in a variety of languages, including AutoLISP, Racket, JavaScript, and Python, and that allows the generation of architectural models in a variety of CAD tools, including AutoCAD, Rhinoceros 3D, and Sketchup.

In the following sections we introduce the Processing programming language and the specific implementation developed for Rosetta. We then explain our proposed extensions for supporting architectural work in Processing and we evaluate their application.

2. PROCESSING

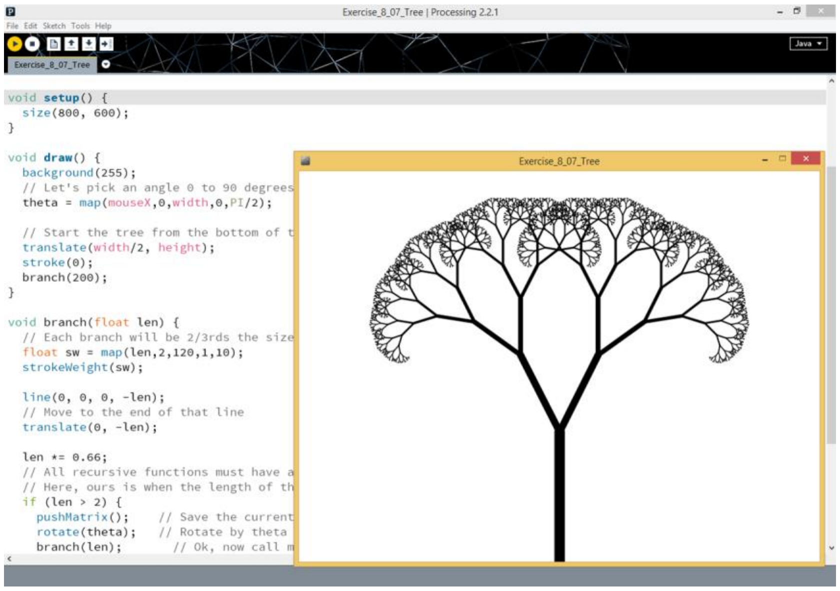
Processing was heavily inspired by the Design by Numbers [7] project, with the explicit goal of being a medium to teach computer science to artists and designers with no previous programming experience. The language has grown over the years with the support of an academic community, which has produced vast amounts of teaching material demonstrating the use of programming in the visual arts. Also, an online community was created around the language, allowing users to share their work. The existence of an online community, good documentation, and a wide range of publicly available examples has been a positive factor for the language's growth over the years.

Processing is based on the Java 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 computer 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 general-purpose programming language, it requires users to grasp a considerable amount of knowledge which can be irrelevant for those who just want to develop simple scripts for visual arts. As a result, several simplifying features were introduced in Processing that enable users to test their design ideas without requiring extensive knowledge of Java. For instance, in order to execute code, Java requires users to define a public class and a public static 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 allow them to create more complex designs in an easier way. Finally, they can shift into a full Java style mode of programming, using object-oriented programming features, and supporting the use of the libraries and the capabilities that are available in the Java language.

To ease the learning effort, Processing is equipped with the Processing Development Environment (PDE), illustrated in Figure 1. This IDE enables designers to develop their programs 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, among others.


Exercise_8_07_Tree | Processing 2.2.1
File Edit Sketch Tools Help
Java
Exercise_8_07_Tree
Exercise_8_07_Tree

```
void setup() {
  size(800, 600);
}

void draw() {
  background(255);
  // Let's pick an angle 0 to 90 degrees
  theta = map(mouseX,0,width,0,PI/2);

  // Start the tree from the bottom of t
  translate(width/2, height);
  stroke(0);
  branch(200);
}

void branch(float len) {
  // Each branch will be 2/3rds the size
  float sw = map(len,2,120,1,10);
  strokeWeight(sw);

  line(0, 0, 0, -len);
  // Move to the end of that line
  translate(0, -len);

  len *= 0.66;
  // All recursive functions must have a
  // Here, ours is when the length of th
  if (len > 2) {
    pushMatrix();    // Save the current
    rotate(theta);   // Rotate by theta
    branch(len);       // Ok, now call m
```

Figure 1: The Processing Development Environment.

One of the main advantages of Processing is its ability to help designers quickly test and visualize their ideas. This is possible due to Processing's rendering system, which is based on OpenGL, allowing the designer to rapidly render complex and computationally intensive designs that would take much longer to produce in typical CAD systems. This makes Processing an excellent tool for testing and developing design prototypes. Unfortunately, when prototypes are developed for architectural work, there comes a time when they need to be transformed into an actual product. It is at this moment that Processing's shortcomings become obvious, as Processing fails to provide a connection with the CAD systems typically used in architecture.

3. RELATED WORK

Despite its limitations, Processing has been used in architecture. To that end, several users developed and proposed extensions to the Processing system that mitigate some of its limitations. These extensions address the generation of three-dimensional shapes, and the transfer of shapes to the traditional CAD tools used in Architecture, Engineering and Construction (AEC).

As a first example, Shapes 3D [8] is a library that extends the set of shapes that can be generated from Processing. This set includes ellipsoids, toroids, helixes, cones, and tubes, among others, as well as shape-forming operations such as extrusions and sweepings.

Toxiclibs [9] is another library for Processing that supports the creation of 3D models through programming and includes the ability to export the created shapes to an STL file, thus allowing it to be printed in 3D printers.

ANAR+ [10] is a geometry library for Processing that was developed with the explicit goal of being a programming interface for designers that want to investigate form exploration strategies based on parametric variations.

Finally, OBJExport [11] is a library that exports meshes with triangle and quadrangle shaped faces from Processing to OBJ or X3D files that can then be imported into some CAD applications.

All the previous libraries demonstrate that it is possible to make Processing more useful for architectural work. However, all of them have an important drawback: they do not support the interactivity of programming directly in a CAD application, as users have to re-export the generated shapes each time the program is run and then re-import them in the CAD application. Moreover, as these export/import steps transform the generated shapes into meshes of triangles, there is a considerable loss of information in the process.

In order to solve these problems, a more integrated approach might be preferable. In this approach, instead of extending Processing with additional shape-generating operations and import/export functionality, we integrate Processing in the CAD environment, exposing the CAD tool functionality to the Processing language and eliminating the need for migration steps. This approach has been used in the past, e.g., for integrating the VBScript or Python programming languages with commercial CAD tools, such as Rhinoceros 3D or AutoCAD, but has not yet been attempted with the Processing language. There is, however, a serious drawback in this approach: it makes the programs developed in such implementations non-portable as, due to the use of CAD-specific operations, they will only be able to run with that particular CAD tool.

4. IMPLEMENTING PROCESSING FOR ROSETTA

Rosetta [12] is an IDE for Generative Design, implemented using the Racket language [13] and taking advantage of the pedagogical capabilities of the DrRacket programming environment [14]. 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 all different supported languages (currently, AutoLisp, JavaScript, Scheme, Racket, and Python) and generate identical models in all supported CAD applications (currently, AutoCAD, Rhinoceros 3D, Sketchup, and Revit).

To make Rosetta more useful for the large community of designers and architects that have learned Processing, we extended Rosetta to support the Processing language. This means that it becomes possible for architects that have learnt Processing to use the language in the context of modern, script-based, architectural work. To this end, we also proposed and implemented 3D modelling extensions to the Processing language that makes it more suitable for the needs of the architect.

The implementation is based on a traditional compiler pipeline: tokenization, parsing, static analysis (including lexical scope analysis and type-checking) and, finally, code generation. Our approach was to develop Processing as a new Racket language module [15, 16], extending Rosetta with Processing and integrating Processing with DrRacket.

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.

5. PROCESSING FOR ARCHITECTURAL WORK

Besides supporting the traditional syntax and semantics of the Processing language, our implementation extends Processing in three different directions: interactive evaluation, 3D modelling, and professional CAD.

Interactive evaluation allows the designer to evaluate small fragments of Processing programs in a Read-Eval-Print-Loop (REPL). This is an important feature for the incremental development of programs, as it allows quick experimentation and validation of the script being developed. This is particularly important for designers that want to visualize the evolution of their design as the script is being written.

Extensions for 3D modelling are essential for improving the use of Processing in the context of professional CAD tools. The original Processing language only provides very basic primitives for 3D modelling, 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 not only the definition of the set of vertices that describe the shape but also the definition of the connections between the vertices (e.g., using lines, triangles, or quadrangular strips). Unfortunately, this process of shape modelling is too low-level and very unnatural for designers. Therefore, as the current Processing environment is rather poor in these modelling operations, we augmented it with a large set of primitive shapes (e.g., cylinder, cone and cone frustum, regular pyramid, torus, etc), boolean operations (i.e., union, intersection, and difference), and other shape-forming operations (e.g., extrusion, sweeping, and lofting). These primitive shapes and operations significantly reduce the effort needed for the generation of complex designs.

Finally, our implementation of Processing is capable of generating designs in a variety of CAD tools that are typically used for architectural work, such as AutoCAD, Rhinoceros, Sketchup, or Revit. One important advantage of our approach is that the generated designs do not suffer from the typical problems that affect designs imported from different applications, namely, the transformation of shapes into meshes of triangles or into non-editable forms.

6. 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. All of the examples were developed by an architect experienced in generative design that only recently started to use the Processing language.

For a first example, 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);
  }
}
```

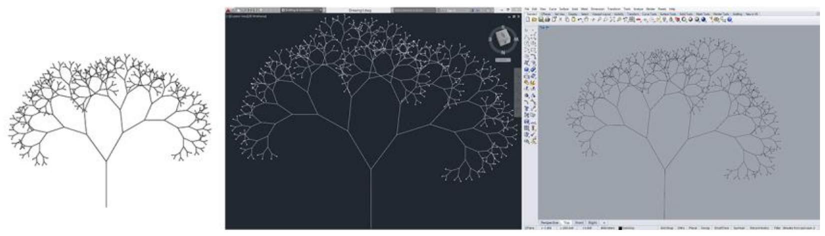


Figure 2: Two-dimensional fractal tree generated in Processing, AutoCAD, and Rhinoceros 3D.

This example generates the typical fractal tree that gradually reduces the length of each tree branch, using Processing's *line* and *ellipse* primitives to create the branches and leaves. Using our Processing implementation, besides seeing the generated tree in Processing's dedicated visualizer, we can generate the same drawing in different CAD applications. Figure 2 illustrates this behavior, showing the use of *tree* in the original Processing environment, as well as in AutoCAD and Rhinoceros 3D. The only change made to the original program was the addition of the *backend* function that allows the architect to specify the CAD environment to use for his design. Note that the generated drawings have slight variations, because of the random function which is used to produce shorter tree branches. Although these 2D illustrations have limited applicability for architectural work, they can be a starting point to explore other designs. For instance, consider the following variation:

```
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),
        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;
    tree3D(x2, y2, z2, l2, a2*1/4, da, r2);
    tree3D(x2, y2, z2, l2, a2*3/4, da, r2);
    tree3D(x2, y2, z2, l2, a2*5/4, da, r2);
    tree3D(x2, y2, z2, l2, a2*7/4, da, r2);
  }
}
```

The previous code demonstrates how easily we can migrate from a simple 2D drawing to a more complex and architecturally useful creation, visible in Figure 3.

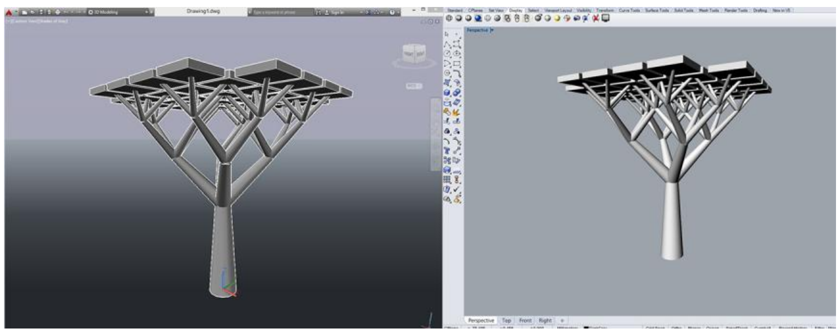


Figure 3: Three-dimensional column tree generated in AutoCAD and Rhinoceros 3D.

In this case, the branches were modeled using the 3D *coneFrustum* primitive, while the endings were modeled using a *box*, thus creating a tree-inspired column. Note that the *coneFrustum* primitive is an example of a primitive that our implementation brings to Processing's modeling set. Also note that *coneFrustum* and *box* were adapted to support Rosetta's coordinate abstractions (in this case, the *xyz* primitive). These abstractions allow users to develop their models more efficiently by taking advantage of different coordinate systems, namely cartesian (*xyz*), polar (*pol*), and cylindrical (*cyl*), which can be used and combined interchangeably. The original Processing system provides some abstractions, such as *PVector* to encapsulate vectors, yet it does not have an appropriate abstraction for coordinates. Usually, users solve this issue by passing points around in an array, or by creating custom abstractions, or even by passing each coordinate value individually, resulting in long function headers and additional verbosity in the program. As a concrete example, consider Figure 4, which shows a model of the Mediopadana station, from Santiago Calatrava. The model was generated by a Processing script running in Rosetta and using Rhinoceros 3D as backend. The undulation of the walls and ceiling was generated by computing vectors of coordinates that follow sinusoidal curves.

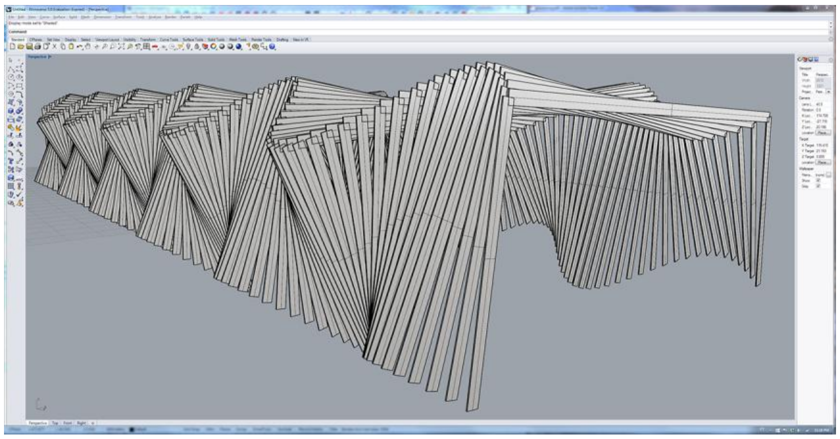


Figure 4: A model of the Mediopadana station, generated from a Processing script running in Rosetta and visualized in Rhinoceros 3D.

Boolean operations, such as *union, intersection*, and *subtraction* are another useful extension available in Rosetta that improves Processing usefulness for architectural work, as it allows architects to use Constructive Solid Geometry (CSG) techniques [17]. As an example, the following Processing script generates the pattern visible in Figure 5, on the left, and its application on a larger scale generates an entire wall made according to that pattern, visible on the right.

```
void setup(){
  backend(autocad);
}

void draw(){
  Object arc = emptyShape();
  Object hole = emptyShape();
  for(int I = 0; i < matrix.length-1; i++) {
    Object[] row0, row1;
    row0 = matrix[i];
    row1 = matrix[i+1];
    for(int j = 0; j < row0.length-1; j++) {
      Object[] p0,p1,p2,p3;
      p0 = row0[j];
      p1 = row1[j];
      p2 = row1[j+1];
      p3 = row0[j+1];
      float radius = distance(p0,p1)/2;
      Object p = quadCenter(p0,p1,p2,p3);
      Object n = quadNormal(p0,p1,p2,p3);
      arc = union(arc,cylinder(p,radius,addC(p,mulC(n,0.6))));
      hole = union(hole,cylinder(p,0.95*radius,addC(p,mulC(n,0.6))));
    }
  }
  Object rings = subtraction(arc,hole);
}
```

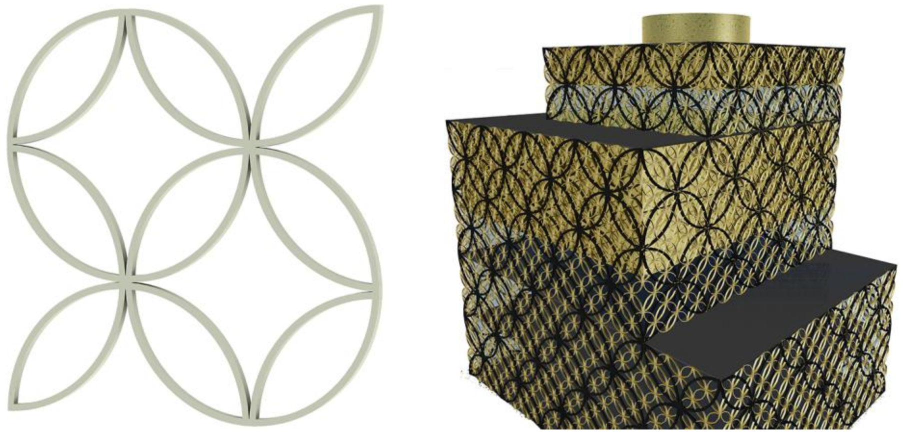


Figure 5: Left: A three-dimensional pattern made by a Processing script that uses unions and subtractions of cylinders. Right: A model of the Library of Birmingham (designed by Mecanoo) generated in AutoCAD by a Processing script running in Rosetta that uses the pattern on the left.

Besides boolean operations, our Processing implementation supports many other shape-forming operations that do not exist in the original Processing language, including *extrusion*, *loft*, *sweep*, *thicken*, etc. These operations dramatically simplify the amount of Processing code that needs to be written by the user. As a concrete example, Figure 6 shows an idealized building facade generated in AutoCAD where each parametric elements is produced using loft operations.

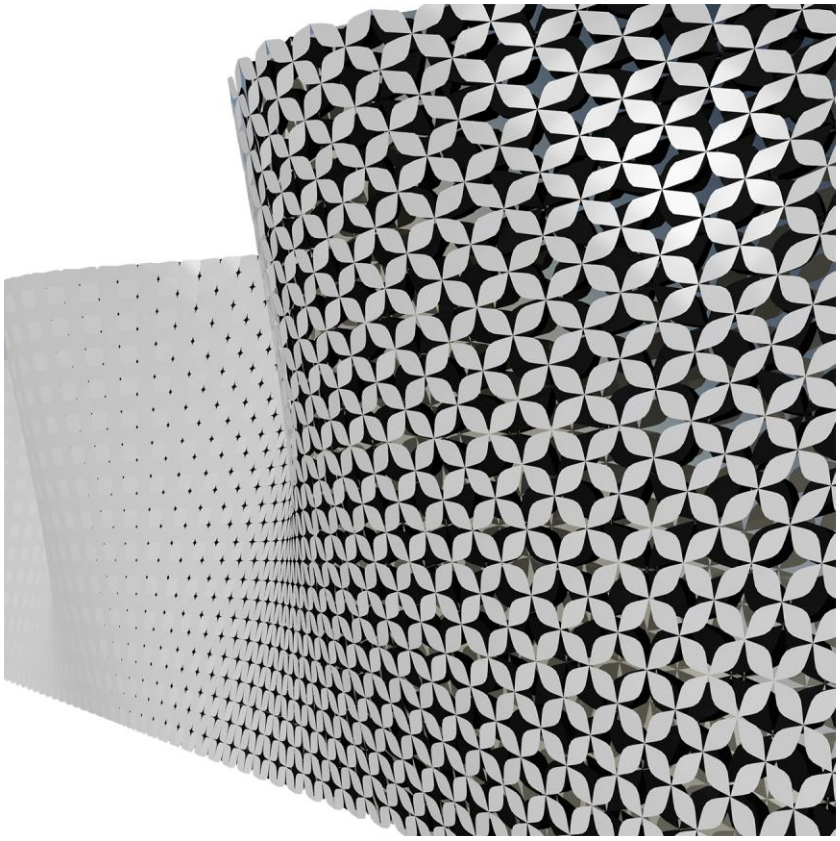


Figure 6: A building facade made by a Processing script using loft operations, generated in AutoCAD.

Finally, we demonstrate an example of our Processing implementation using libraries that are written in another language (shown in Figure 7):

```
require "elliptic-torus.rkt";
backend(rhinoceros);

Object center = xyz (0,0,0);
float aMin = QUARTER_PI, aMax = 7 * QUARTER_PI;
ellipticTorus(center, 0.005, 0.03, 0.5, aMin, aMax, 0, TWO_PI);
```

To produce this example, our Processing code *require*s *elliptic-torus.rkt*, a library written in the Racket language that is capable of generating a parametric elliptic torus where we specify, in Processing, the domain range, the thickness of the surface, the size of the surfaces' holes, etc. The generated model is visible in Rhinoceros 3D.

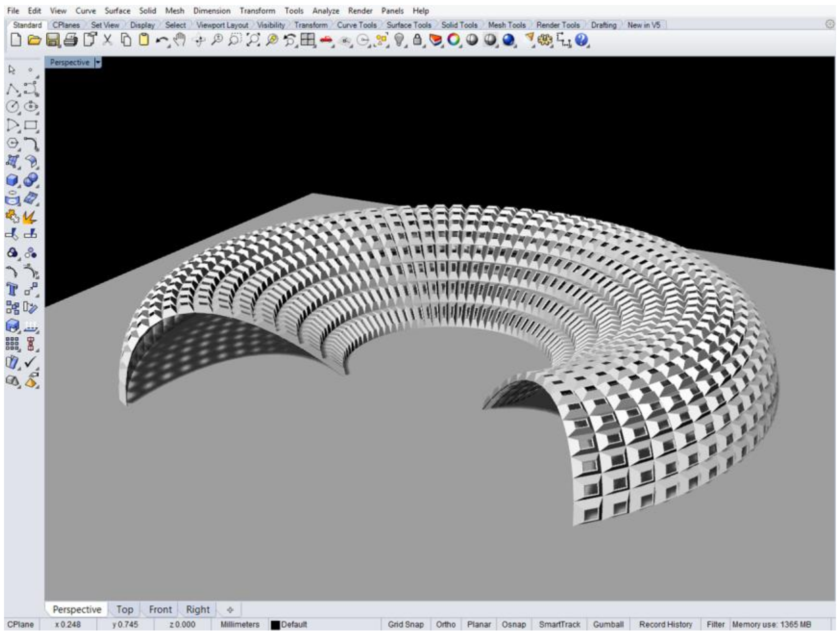

File Edit View Curve Surface Solid Mesh Dimension Transform Tools Analyze Render Panels Help
Standard CPlanes Set View Display Select Viewport Layout Visibility Transform Curve Tools Surface Tools Solid Tools Mesh Tools Render Tools Drafting New in V5
Perspective
Perspective Top Front Right
CPlane x 0.248 y 0.745 z 0.000 Millimeters Default Grid Snap Ortho Planar Osnap SmartTrack Gumball Record History Filter Memory use: 1365 MB

Figure 7: Elliptic torus generated using Rhinoceros 3D.

7. CONCLUSION

Processing empowers the creativity of its users. However, it has been difficult in the past to use Processing in combination with the traditional CAD tools used in Architecture, such as AutoCAD or Rhinoceros 3D. In this paper, we presented an implementation of Processing that solves this problem, increasing the creative freedom of architects while allowing them to work with their preferred professional tools.

By augmenting Processing with new design paradigms and abstractions, namely 3D modelling primitives (torus, cone, cylinder, etc) and transformations (union, subtraction, loft, etc), we make the language more attractive for the architecture community. We demonstrated that our implementation extends Processing with features that allow users to create new designs using a more expressive modelling approach. For instance, using our system, we can easily access and combine several modelling primitives, allowing the development of complex designs that would be much harder to implement in the original Processing environment.

Nowadays, the Processing language and design approach are 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.

Another essential feature of the language is the Processing Development Environment (PDE), as it reduces beginners' programming learning curve. Our implementation employs a similar solution, by adapting the pedagogical tool DrRacket for the development of Processing programs, offering a simplified editor and development environment that is almost identical to the PDE. However, we also provide an important additional feature, the Read-Eval-Print-Loop (REPL), which is unavailable in other Processing implementations, that allows users to quickly experiment their design ideas.

Our Processing implementation is available in [18] and, although still in the testing phase, it 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 Processing language, such as classes and inheritance.

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

1. Burry, M. Scripting Cultures: Architectural design and programming. John Wiley & Sons, 2013.

2. Reas, C. and McWilliams, C., Form+Code: In Design, Art, and Architecture, Princeton Architecture Press, 2010.

3. Reas, C. and Fry, B., Processing: Programming for the Media Arts, AI & SOCIETY, 2006, 20(4), 526-538.

4. Leach, N., Digital cities, Architectural Design, 2009, 79(4), 6-13.

5. Ahlquist, S., and Menges, A., Physical Drivers: Synthesis of Evolutionary Developments and Force-Driven Design, Architectural Design, 2012, 82(2), 60-67.

6. Achten, H., Teaching advanced Architectural issues through principles of CAAD, in: Edited by Klercker, A., Ekholm, A. and Fridqvist, S., Education for Practice: Proceedings of the14th Conference on Education in Computer Aided Architectural Design in Europe (eCAADe), Lund, Sweden: Lund Institute of Technology, 1996, 7-16.

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

8. http://lagers.org.uk/s3d4p/index.html [17-10-2015].

9. http://toxiclibs.org/ [17-10-2015].

10. Labelle, G., Nembrini, J. and Huang, J., Geometric Programming Framework, ANAR+: Geometry library for Processing, in: Edited by Schmitt, G., Hovestadt, L., Van Gool, L., Bosché, F., Burkhard, R., Colemann, S., Halatsch, J., Hansmeyer, M., Konsorski-Lang, S., Kunze, A. et al., eCAADe 2010 Conference: Future Cities: Proceedings of the 28th Conference on Education in Computer Aided Architectural Design in Europe (eCAADe), ETH Zurich, 2010, 403-410.

11. http://n-e-r-v-o-u-s.com/tools/obj/ [17-10-2015].

12. Lopes, J. and Leitão, A., Portable Generative Design for CAD Applications, in: Edited by Taron, J., Parlac, V., Kolarevic, B. and Johnson, J., ACADIA 11: Integration Through Computation: Proceedings of the 31st annual conference of the Association for Computer Aided Design in Architecture (ACADIA), Banff, Alberta, 2011, 196-203.

13. Flatt, M., Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc, 2010.

14. Findler, R., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., and Felleisen, M., DrScheme: A programming environment for Scheme. Journal of Functional Programming, Mar. 2002, 12(2), 159–182.

15. Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., and Felleisen, M., Languages as Libraries. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11). ACM, New York, NY, USA, 2011, 132-141.

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

17. Requicha, A. Representations of Rigid Solid Objects. Springer, Berlin Heidelberg, 1980.

18. https://github.com/aptmcl/p2r [17-10-2015].