A Browser-based Programming Environment for Generative Design
[Extended Abstract]

Pedro Alfaiate, pedro.alfaiate@tecnico.ulisboa.pt

Instituto Superior Técnico, Universidade de Lisboa

May 2017

Abstract

Generative Design (GD) allows architects to explore design using a programming-based approach. Current GD environments are based on existing Computer-Aided Design (CAD) applications, such as AutoCAD or Rhinoceros 3D, which, due to their complexity, are slow and fail to give architects the feedback they need to explore GD. In addition, current GD environments are limited by the fact that they need to be installed and, therefore, are not easily accessible from any computer. Architects would benefit from a GD Integrated Development Environment (IDE) in the web that is accessible without installation and that is more interactive than existing GD environments.

This thesis proposes a GD IDE based on web technologies. Its main component is a web page, containing a program editing interface that allows the architect to make programs and view results in 3D. To make the editing experience more intuitive, it runs programs whenever they are changed, allows numeric literals to be adjusted by clicking and dragging, and highlights the relationship between program and results. The IDE also includes a secondary application for exporting results to CAD applications installed in the architect's computer.

With this approach, we were able to implement a GD environment that is accessible from any computer, offers an interactive editing environment, and integrates easily into the architect's workflow. In addition, in what concerns program running times, it has good performance that can be one order of magnitude faster than current GD IDEs.

Keywords: Generative Design; Web technologies; Integrated Development Environments; Architecture

1. Introduction

Through the years, computers have been taking more ground in the field of architecture. In the beginning, they were only used for creating technical drawings using Computer Aided Design (CAD) software but, later, they began to also integrate 3D modeling capabilities. However, modeling a complex building is still a time-consuming activity that requires several repetitive tasks that are not trivial to accomplish using just the functionalities that a 3D modeling software provides.

The recognition of this problem has led to the emergence of programming languages for 3D modeling software. By using programming, the architect is able to use a program to describe his design intention and let the computer do the modeling task for him. On top of that, after writing the program, the process of generating and applying changes to the model is much faster than the manual equivalent. Like so, the architect gets to do more in less time.

Having a faster and more flexible process for building 3D models allows the architect to explore more variations of a design, i.e., to explore a broader design space, since it is no longer too expensive and time consuming to remodel. At first, he can just explore changes to parameters, but, if he wants, he can also change the algorithm. By making these changes, he can create a large amount of variations of the generated models. Designing by using programming to generate parts of the design is what is called Generative Design (GD). GD allows computers to be used as a new medium for artistic expression[1] that can be used by architects, as shown in [2]. Furthermore, as stated in [3], using GD as a new stage of the design process promotes a simpler handling of changes coming from uncertain design intents and emergent requirements,as problem understanding improves or as needs change. However, in order to create the programs that will generate his designs, the architect needs to have a programming environment that lets him write the programs and see

their results.

To create GD programs, the architect needs to use a programming language, its runtime environment (like the CAD where the models are generated), and an Integrated Development Environment (IDE). The programming language defines what is a program, that is, what concepts can be part of it and how they can be combined to perform tasks that can be understood and executed by a computer. The IDE provides tools – editors, compilers, debuggers, among others – that let the architect create programs.

In order to help architects with the creation of GD programs, visual programming IDEs, such as Grasshopper[1] and Dynamo,[2] provide tools that are capable of showing all the nodes/functions that can be added, and the relationship between the code being developed and the results. They are, therefore, more popular among novices, since they do not require as much training to be used when compared to textual programming IDEs. However, as stated in [4], visual programming languages do not scale well when used in big GD projects when compared to textual programming languages. Textual programming languages have mechanisms that make complexity more manageable, so programs can stay smaller. Still, textual programming IDEs lack ways to show the connection between program and results that visual programming IDEs have. Some work on this has been done in Rosetta, as shown in [5], where it is possible to see which objects each part of the program generated, and the other way around, that is, which parts of the program were responsible for creating an object.

In spite of this, up until now GD IDEs have been primarily desktop applications, which brings some problems. Firstly, every computer has its own installation of the IDE, meaning that users may not be using the most up to date version of the IDE. Secondly, desktop applications are more easily pirated, since they run solely on the computer they are installed in. Moreover, they also generally offer limited support when the architect needs to work remotely while still collaborating with his coworkers.

Web applications are less prone to these problems. This has led to the creation of many web application counterparts of common desktop applications. For example, office productivity tools like Microsoft Word, Excel and PowerPoint have seen the appearance of their web application counterparts when Microsoft Office 365 appeared. Furthermore, complete CAD web applications have also appeared. One example is OnShape,[3] a CAD application for Product Design/Engineering completely accessible from the web browser.

What remains to be done is making this functionality available to architects.

1.1. Goals

This thesis aims at increasing the architects' productivity when working on GD projects by giving them a programming environment in the web specifically for GD. This requires the implementation of a web application, that harnesses the performance and graphical capabilities available in modern web browsers, and allowing models to be exported to traditional CAD applications.

2. Background

In order to develop a programming environment for GD that works as a web application, we first need to study the currently available GD environments.

Not all presented solutions are used exclusively for GD. GD encompasses both 3D modeling and programming, so it makes sense to not only look at systems that explore both aspects, but also systems that explore just one of them. Furthermore, the presented solutions include both desktop-based and web-based applications to allow the understanding of what is currently possible in the cloud and how it compares to what is available on the desktop.

2.1. Related Work

2.1.1 LightTable

LightTable[6] is a code editor for the Clojure programming language[7] and is an example of a desktop application that uses web technologies.

Although not related to GD, LightTable proposed some interesting features and concepts for IDEs. One of these is the use of the drafting table as metaphor.[4] Instead of displaying the contents of entire files, LightTable divides the code into meaningful units and displays them as small editors spread over the table's surface. Figure 1 shows an example of this metaphor, which has some resemblance to node-based/visual programming environments, since the programmer still has to think of how to arrange what is on the table.

Other feature deals with making navigation in Clojure code bases easier. In Clojure, functions are defined inside namespaces and all Clojure definitions (like functions, variables, macros) are stored in text files. Navigating among definitions and the namespace structure should not get in

[1] https://www.rhino3d.com/ (last accessed on 10/05/2017)
[2] http://dynamobim.com/ (last accessed on 10/05/2017)
[3] https://www.onshape.org (last accessed on 10/05/2017)
[4] http://www.chris-granger.com/2012/04/12/light-table-a-new-ide-concept/ (last accessed on 10/05/2017)

Figure 1: LightTable's drafting table. A game is being run inside it while some of its code is displayed in separate editors.

the way of editing code. To make editing easier, LightTable provides a *namespace browser* that allows to find functions and a *code document* where functions can be added for editing, without moving them out of their namespace or displaying entire files where they are defined.

Another interesting feature of LightTable is its ability to show data flow in a function call. Since the main purpose of a function is to transform its input data into its output data, it helps to see what happens to the data on each step of the function. To achieve this, LightTable overlays variable values and return values, respectively, on each variable occurrence and expression of the function.

This last feature has the problem of not being capable of showing the data-flow for more than one run of a part of the code. If a function is called more than once, then it will be more useful to show the data-flow of all the calls.

2.1.2 IPython

IPython[8] is a programming environment directed towards providing better, more straightforward, scientific computing.

IPython can be used as a command shell by using its notebook User Interface (UI), the IPython notebook. As seen in Figure 2, IPython's notebooks can contain not only source code but also results and rich text. These notebooks are displayed using an interactive web page.

The style of producing notebooks in IPython is one that mixes programming, writing and exploring. Interestingly, this style is also part of a designer's processes. Like a scientist, the designer also has to do exploration of ideas (design ideas in his case), reach conclusions (finished designs) and share his work with others (fellow designers, clients, friends, blog readers). Unfortunately, although IPython notebooks are natural tools for exploration, they do not provide domain specific functionality for architecture.

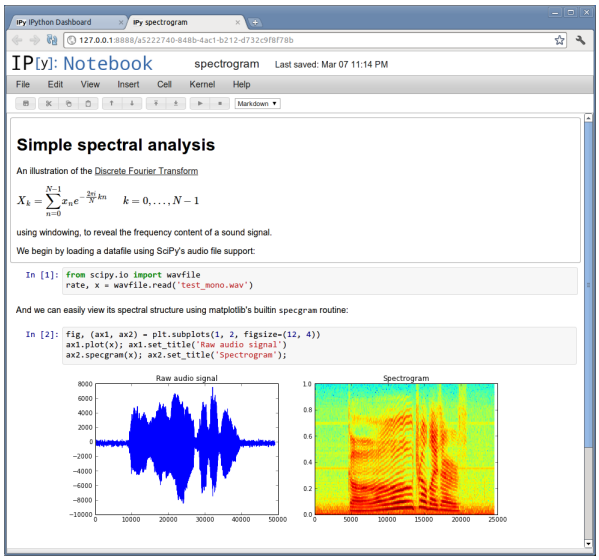


Figure 2: An IPython notebook with rich text, mathematical notation, source code and results from executing such code.

To allow more flexibility between the core computing functionality and the UI, IPython was decomposed into execution kernels, a communication protocol, and several front-ends. Execution kernels are responsible for running the code of notebooks, and front-ends implement the UI, as is the case of IPython's notebooks. The communication protocol defines how communication between execution kernels and front-ends is done. With this, it is possible to implement new execution kernels for running code from a programming language not yet implemented or an alternative front-end different from IPython notebooks. Moreover, the communication protocol also allows execution kernels and front-ends to run on different computers[8].

2.1.3 OpenJSCAD

OpenJSCAD[5] is a project aiming to implement OpenSCAD[6] using web technologies. OpenJSCAD supports most of OpenSCAD's functionality. Like OpenSCAD, it focuses on creating 3D models for 3D printing. Similarly to the GD approach, to actually model in OpenJSCAD, the user has to write a program in either JavaScript or OpenSCAD's language.

OpenJSCAD provides two user interfaces, one Command Line Interface (CLI) and one Graphical User Interface (GUI), the latter implemented as a web page.[7] Moreover, the first can be used for batch processing, while the second integrates an editor to edit a program and a 3D view for viewing the corresponding results. This separation enables

[5] http://openjscad.org/ (last accessed 10/05/2017)
[6] http://www.openscad.org/ (last accessed 10/05/2017)
[7] https://github.com/Spiritdude/OpenJSCAD.org (last accessed on 10/05/2017)

the use of OpenJSCAD for programming, without requiring the programmer to install anything more than a web browser, which is almost always already installed.

OpenJSCAD makes its functionality available as functions, as well as methods on its objects, which makes writing programs more flexible. Both functions and methods return new objects and do not have side-effects. This allows the programmer to use the functional programming paradigm and makes it easier to understand programs, as there are less side-effects that could change their behavior.

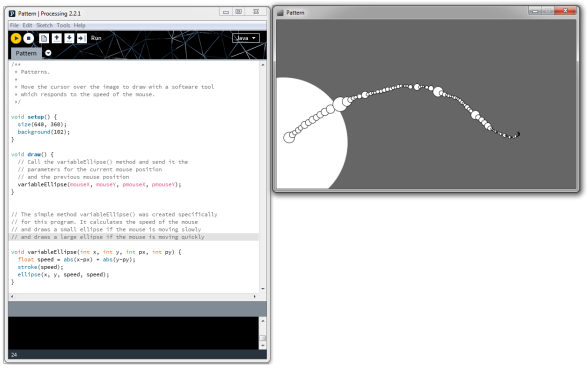


Figure 3: On the left: The PDE displaying an example *sketch* while it is being run. On the right: The drawing window to which the *sketch's* instructions are applied.

2.1.4 Processing

Processing[9] is a programming language and a development environment aimed at "promoting software literacy in the visual arts and visual literacy within technology".[8]

Processing enables everyone to write programs that both draw to the screen and react to input from the user, like moving the mouse or pressing a key on the keyboard. It makes this possible by implementing most of the functionality that is commonly required, like initializing the drawing surface, so the programmer only has to implement the functionality specific to the result he wants to achieve. The code in Listing 2.1, for instance, is what is needed to setup a drawing canvas, its background color, and continuously draw a line from the mouse position to a point on the canvas.

To use Processing's programming language, one needs to use its IDE, the Processing Development Environment (PDE). As shown in Figure 3, the Processing Development Environment (PDE) includes a text editor with syntax highlighting and runs Processing programs.

```
//Hello mouse.
void setup() {
   size(400, 400);
   stroke(255);
   background(192, 64, 0);
}

void draw() {
   line(150, 25, mouseX, mouseY);
}
```

Listing 2.1: A simple Processing sketch

Processing is sometimes used by architects for exploring design ideas. However, since most of its functionality revolves around graphics for the visual arts, its use is normally restricted to 2D designs and cannot be used as a full-fledged tool for GD.

Several projects have stemmed from Processing to extend its functionality to different programming languages. These include Processing.py[9], that extends the PDE to support Python, and p5.js[10], that provides JavaScript libraries to create interactive web pages.

2.1.5 DesignScript

DesignScript[10] is a programming language that was designed to suit the needs of architecture related design.

DesignScript uses concepts from multiple programming paradigms like object-oriented, functional, and associative programming[10]. Entities have properties that can be either data or functions, like in object-oriented languages; functions' most important role is to take some input and produce some output without producing side-effects, like in functional languages; and dependencies among variables are retained, like in associative languages.

Being a domain-specific language for architecture, DesignScript provides functions for 3D modeling, such as creating concrete 3D objects, like cubes and spheres, as well as abstract geometry, like planes and points, that is used as scaffolding.

DesignScript also supports lists of values and lets them be used in place of single parameters in calls to functions as it is common to start with one value and then scale up to many. This lets architects use lists more easily, as they do not have to use loops to extract values and pass them to functions.

Having modeling primitives and combining several programming paradigms allows the architect to draw from knowledge about architecture modeling,

[8] Quoting https://www.processing.org, 9/Nov/2015.

[9] http://py.processing.org/ (last accessed on 10/05/2017)

[10] http://p5js.org/ (last accessed on 10/05/2017)

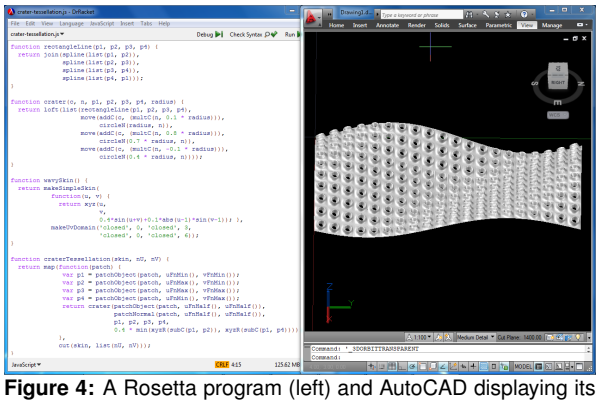


Figure 4: A Rosetta program (left) and AutoCAD displaying its results (right). The program is written in Javascript.

while empowering him to express the processes in which those primitives are used.

DesignScript is used in several environments, all of which are desktop applications. These include a textual editor in Autodesk AutoCAD, a dedicated graph editor called DesignScript Studio and Dynamo, which integrates with Autodesk Revit. Both DesignScript Studio and Dynamo use graph based program editing.

2.1.6 Rosetta

Rosetta[11, 12], shown in Figure 4, is an environment for GD.

Rosetta's motivation is to allow architects to write portable GD programs that generate equivalent models in any CAD applications they use. Rosetta allows architects to choose the front-end programming language and the back-end where the primitive operations will be performed[11].[11] With this, architects can experiment with GD without having to switch CAD, and they can also share programs with others using different CADs.

Additionally, Rosetta also allows programs written in one programming language to use not only parts from programs written in the same language, but also from programs written in other languages. This makes it possible for architects to share programs written in different programming languages, and it also allows them to choose the programming language that best fits each part of the program they are working on.

2.1.7 OnShape

OnShape[12] is a cloud-based CAD application that can be accessed using either a web browser or its mobile app. As opposed to the systems we talked about earlier, OnShape is a CAD for mechanical modeling and, as such, its users are familiar with CADs like SolidWorks[13] and Autodesk Inventor.[14]

OnShape has version control of project documents (drawings, 3D models, text documents) and supports real time collaboration. Like most cloud applications, OnShape promotes collaboration on development teams where team members work remotely. In order to do this, it allows users to work on the same copy of files, as opposed to each working with his own copy, and therefore changes are synchronized automatically. As users can edit the same copy of a file at the same time, OnShape shows each user what the others are doing to enhance real time collaboration.

OnShape also includes a dedicated programming language, FeatureScript[15], for defining features for its models. OnShape includes an IDE for FeatureScript, but the language does not aim to replace OnShape's modeling approach based on interactively manipulating the model's feature list, which makes it impossible to use for GD.

2.2. Comparison

In this section, we have presented environments that support activities related to programming, modeling and GD. We looked at this diverse set of environments to get the broader picture of domain-specific programming environments, 3D modeling/CAD applications, and some of their cloud counterparts.

The environments supporting GD directly — namely Rosetta, DesignScript, OpenJSCAD, and, to some extent, Processing — follow either visual editing, which is easier to learn and use, or textual editing, which, although less intuitive, supports languages with mechanisms to keep the complexity down in bigger problems.

When it comes to programming paradigms, the presented environments support imperative, functional, and associative programming. More specifically, visual environments support the associative paradigm, while textual environments support the imperative paradigm.

Turning our attention to web applications, we can see that although IPython does present notebooks as intuitive ways to program, it does not have the 3D modeling primitives required for GD. On the other hand, OnShape does provide a complete set of 3D modeling primitives, however, it does not allow the use of programming as the main driver of

[11] Some front-ends supported by Rosetta are AutoLisp (one of AutoCAD's programming languages), Javascript, Racket and Python; some of the supported back-ends include CADs like Autodesk AutoCAD, Autodesk Revit, Sketchup, and Rhinoceros 3D and also graphics libraries like TikZ and OpenGL.

[12] https://www.onshape.com/ (last accessed on 10/05/2017)

[13] http://www.solidworks.com/ (last accessed on 10/05/2017)

[14] http://www.autodesk.com/products/inventor/overview (last accessed on 10/05/2017)

[15] https://www.onshape.com/featurescript (last accessed on 10/05/2017)

modeling tasks. Finally, OpenJSCAD supports 3D modeling primitives and programming is its only way to do it, however, it still lacks the traceability that can be found in other IDEs, like Rosetta and DesignScript, and also lacks the interoperability provided by Rosetta.

Looking at collaboration, OnShape is the only environment that supports it directly, allowing for real-time collaboration, thus enabling its users to work on the same 3D model simultaneously. For the rest of the environments, collaboration is the user's responsibility.

In conclusion, although there are capable GD environments available to architects, they are only available as desktop applications. There is no capable GD environment on the web that is also capable of providing interoperability and intuitive editing features. With this in mind, we will be presenting our own solution in the next sections.

3. Solution

3.1. Overview

We set out to create an IDE for GD programming that is available as a web application. To do that, we identified several tasks that the programming environment needs to support in order to be useful for the architect. It needs to: (1) let the architect develop programs; (2) run programs; (3) display their results; (4) make it easy to understand programs; and (5) export results to the most used commercial CAD applications.

To accomplish these tasks, there are two separate components, as seen in Figure 5: (1) the web application, that supports the UI for creating programs; and (2) the remote CAD application, that offers an Application Programming Interface (API) for running programs in CAD applications. The first four tasks can be handled by the web application alone since there is no need to interact with the CAD applications running in the user's computer. The forth task requires both the web application and the remote CAD service.

We made a test implementation of this architecture which we will call Luna Moth from here on. In the rest of the section, we describe these two components and how each of the tasks is implemented in Luna Moth.

We start by looking at the web application.

3.2. Web application

To handle the tasks of letting the architect develop programs, running programs, supporting program understanding, and displaying results, Luna Moth's UI has the layout shown in Figure 6. Taking up most of the screen space are a text editor (C) and a 3D view (D) that allow the architect to edit a program and view its results. On top of the text editor and the 3D view are controls for the running process (B), including whether to run automatically and whether to collect data to display traceability. On the left are hidden panels for actions like selecting a program and exporting to CAD applications (A).

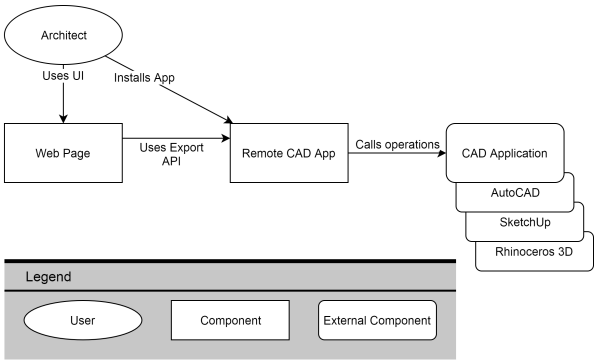


Figure 5: Architecture of the solution.

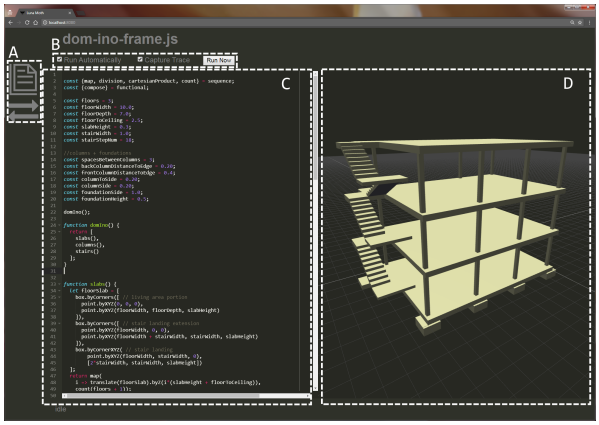


Figure 6: Layout of Luna Moth's UI.

3.2.1 Program Comprehension

In GD, the architect interacts with the program that creates the model instead of creating it directly in a CAD application. This allows him to automate tasks that normally take too much time and, therefore, also allows him to explore a design space with much more complexity. However, since he is not directly interacting with the model, he needs to understand the relationship between the program and its results in order to know how to change the program to get the model he wants. The process of understanding the program and how it relates to its results is called program comprehension[13]. Luna Moth enables him to do this by providing immediate feedback to changes and by showing traceability.

Adjusting Literals When a value, such as a number, is typed directly into a program's source code, it is called a literal value, or simply literal. When using GD, architects find themselves repeat-

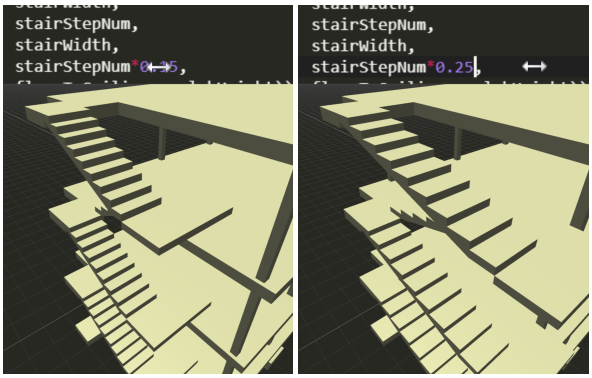


Figure 7: Example of literal adjustment.

edly adjusting these literals to tweak the generated model. This is usually done by rewriting parts of the literal's text with higher or lower digits and, then, re-running the program to see the effect.

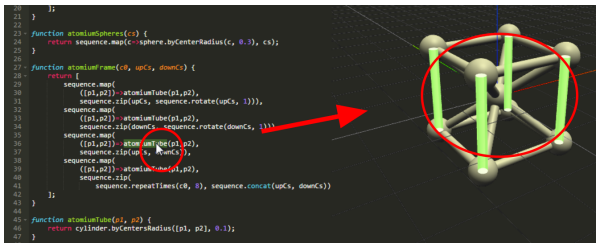
However, editing literals this way often leads to errors, since it is easy to mistype characters. The architect can increase the literal's value by an order of magnitude if, by mistake, he inserts one character without removing another. Furthermore, when increasing the literal in steps, different hand movements are required when the increment results in a carry over, e.g. when going from ninety-nine to one hundred, yet again increasing both the likelihood of mistyping and the time it takes to make the changes.

These mistakes get amplified when the programming environment provides real-time feedback and, like so, begins rerunning the program before the error is corrected, thus leading to reduced UI responsiveness. The adjustment can be friendlier if done in a clearer way instead of retyping the literal, Luna Moth enables the adjustment by simply "clicking and dragging", a movement that is similar to a "slider".

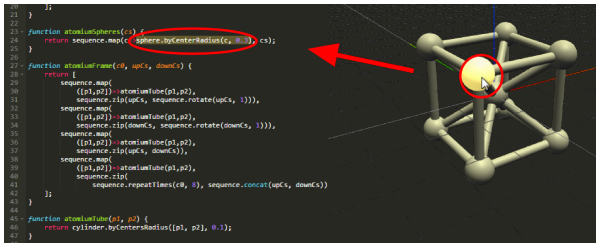
Traceability Reducing the time between making a change and seeing its effect is one way to help the architect to understand a program.

Another way to understand the program is to understand the relationship between parts of the program and parts of the results. If the programming environment takes care of tracking the relation between the program and results, the architect only needs to ask for the relation instead of having to track it by himself, freeing his mind to think about the problem.

There needs to be a way to give the architect quick access to the relation being tracked by the environment. To do this, Luna Moth highlights certain parts of the program and its results when the pointer is on either one. More specifically, when the pointer is over a part of the program, it highlights all the results that it produced; when the pointer is



(a) From code to results.



(b) From results to code.

Figure 8: Two examples of the traceability mechanism. The first from program to results and the second from results to program.

over a part of the results, it highlights the part of the program where it was created. Figure 8 shows an example for both of this cases.

Currently, Luna Moth only tracks the results produced by calls to functions. This was a compromise to balance performance, since keeping track of traceability is an additional task that needs to be performed while running the program.

3.2.2 Running Programs

One of the fundamental parts of the programming environment is that it runs programs. To run programs, the environment needs to support a programming language. This includes the syntax and semantics of the language and the primitive operations that are available to programs. After knowing these, the environment must implement a process to run the program and collect results so as to display them later.

Program Structure Before a program can be run, the programming language has to be defined. In Luna Moth, we decided that it would be best to use JavaScript, as it was thought for designers and also because of its current performance in modern web browsers.

In addition to the programming language, the environment also needs to get results from programs. Luna Moth uses the values of the top-level expression statements of the program as its results.

Running process When Luna Moth runs a program, it needs to do two things: (1) collect the results of the program; and (2) collect traceability

```
let s = sphere.byRadius(5);
s;
```

Before

```
let s = recordCallResult(
   sphere.byRadius(5),
   /*function call ID*/);
recordTopLevelResult(
   s,
   /*top-level expression ID*/);
```

After

Figure 9: A program before and after the transformation.

data. It does these two by instrumenting the program with additional calls to special functions that record the desired information.

To be able to perform the instrumentation, Luna Moth parses the program using Esprima, a JavaScript parser,[16] to get its Abstract Syntax Tree (AST).[17]

Afterwards, the AST is transformed by adding a recording call to (1) all top-level expression statements and (2) all function call expressions. Figure 9 exemplifies the transformation. To let the recording functions know what top-level expression or function call is producing a result, we also provide them with identifiers for those expressions.

After this step, Luna Moth creates a new function with the instrumented program as its body. The primitives and recording functions are provided as parameters to this new function.

Finally, the newly created function is called and, afterwards, the recorded information is made available to the rest of the system.

3.3. Remote CAD Service / Exporting to CAD

The last task that Luna Moth needs to support is exporting to the most used commercial CAD applications. As opposed to the other tasks, exporting to a CAD application requires the web page to communicate with other applications. Unfortunately, web pages are not allowed to communicate with applications running in the computer displaying them.

To make communication possible, Luna Moth includes an application, the *remote CAD app*, that the architect must install and run in his computer when he wants to export results. This application

[16] https://github.com/jquery/esprima (last accessed on 10/05/2017)

[17] The AST conforms to a community standard. It can be found at https://github.com/estree/estree (last accessed on 10/05/2017).

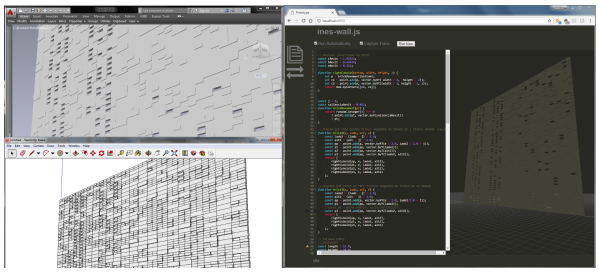


Figure 10: An example of the remote CAD service. The results of the program have been passed to both AutoCAD and SketchUp.

serves as a bridge between CAD applications installed in the computer and the environment running on the web browser. After being started, the application detects which CAD applications are installed in the computer and connects to the *environment directory* so it can be discovered by the web application. When the architect needs to export a program to his CAD application, the environment connects to the application in his computer through the *environment directory* and starts to send requests to it, in order to create shapes in the CAD application.

Figure 10 shows an example of the export to CAD functionality. The architect has created a program using Luna Moth. After selecting AutoCAD and SketchUp as export destinations, he starts the export process and, afterwards, the model resulting from running the program is available in both CAD applications.

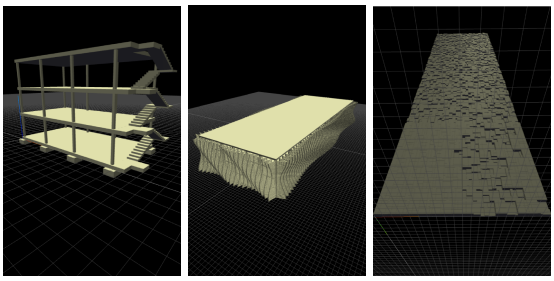
4. Evaluation

4.1. GD capability

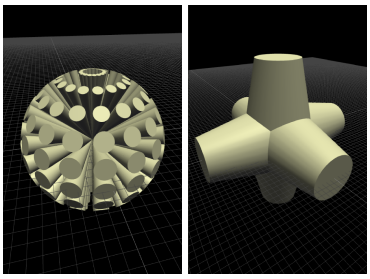
The goal of our work was to bring GD to the web browser so it can be used anywhere more easily. To do this, we implemented a set of primitives that produce several 3D concepts to be used as building blocks for modeling.

Now comes the time to test Luna Moth's ability to be used for GD, more specifically to produce diverse 3D models. In order to evaluate this, we developed several programs that reproduce examples that were previously done in other GD environments. The results of these programs are shown in Figure 11.

Still, the range of results is limited by the primitives currently implemented. For example, none of these programs make use of Constructive Solid Geometry (CSG) operations like union, intersection and difference as they are not implemented. Nonetheless, it is possible to support them since there are JavaScript libraries that already implement CSG, like the case of OpenJSCAD's. These operations should be addressed in future work. In addition, the support for randomness and program-



(a) Dom-ino **(b)** Carmo Facade **(c)** Sheung-wan hotel



(d) Sphere of cones **(e)** Orthogonal Cones

Figure 11: Renderings of results of the implemented example programs.

ming techniques like higher-order functions keeps the ability to create complex models.

4.2. Performance

We evaluated our solution's performance from different perspectives: (1) how running performance in Luna Moth compares with other GD environments; (2) how running performance compares to export performance; (3) how traceability data collection affects running performance; (4) how export performance compares with exporting in other IDEs, like Rosetta and Grasshopper.

To make this evaluation possible, we measured times of several situations: (1) running in Luna Moth, with traceability enabled; (2) running in Luna Moth, with traceability disabled; (3) exporting from Luna Moth to AutoCAD; (4) running in Rosetta connected to AutoCAD; (5) running in OpenJSCAD; (6) running in Grasshopper, while generating previews in Rhinoceros; (7) running in Grasshopper, followed by baking geometry to Rhinoceros.

The next sections presents the evaluation from the previous perspectives.

Setup All tests were performed on a Microsoft Surface Pro 4 with in Intel Core i5 processor and 4GB of RAM. The web application and the remote CAD service were hosted at the same computer.

4.2.1 Running Performance

To see how the performance of our web-based GD environment compares to the existing environments, we compared the time each takes to generate identical models. First, we implemented a version of each program using each environment's programming language and, then, measured the time each took to complete. These times are shown in the chart on Figure 12. As can be seen, the relationship between running times varies from program to program, although Luna Moth is consistently faster than Rosetta (connected to AutoCAD), OpenJSCAD and Grasshopper (*Grasshopper + Rhino*). In some instances, we can see a difference of at least an order of magnitude.

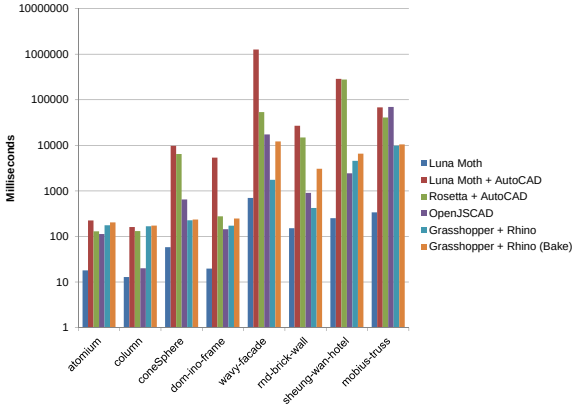


Figure 12: Comparison of running times for our IDE, its export process, Rosetta, OpenJSCAD, and Grasshopper. Note that the time is in logarithmic scale.

With this in mind, we can say that Luna Moth can provide better feedback than Rosetta.

4.2.2 Run vs Export

The next step was to evaluate how the execution time differs between the normal running process and the remote CAD running process. To measure the difference, we ran the previous programs using Luna Moth's export process and measured the time each took to complete. These times are also shown in Figure 12. As seen in the chart, comparing export times (*Luna Moth + AutoCAD*) to normal running times (*Luna Moth*), export times range from being around twelve times to eighteen hundred times the normal running times.

Given that the difference can grow to three orders of magnitude, it is highly preferable to get feedback using the web application instead of using the export process. The export process should be reserved to when the architect is satisfied with the obtained solution.

4.2.3 Export vs Other IDEs

Another comparison that we have to make is one between Luna Moth's export process and similar processes in other IDEs, namely, Rosetta and

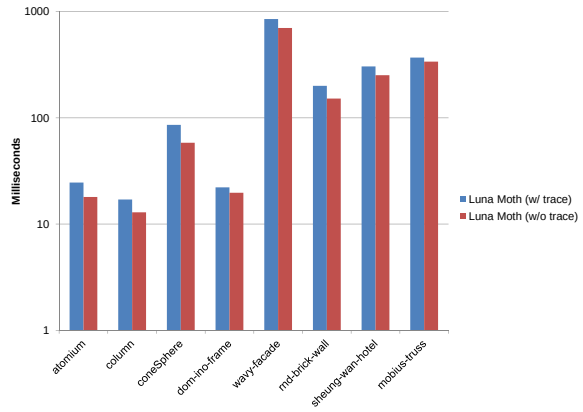


Figure 13: Running while collecting traceability data and while not collecting traceability data. Note that times are in logarithmic scale.

Grasshopper. For this comparison, we took the times from our export process, Rosetta connected to AutoCAD, and Grasshopper baking geometry to Rhinoceros.[18] These correspond, respectively, to *Luna Moth + AutoCAD*, *Rosetta + AutoCAD*, and *Grasshopper + Rhino (Bake)* in Figure 12. As can be seen, Luna Moth's export times range from ≈1x to ≈24x the times of running in Rosetta and range from ≈0.9x to ≈104x the times of running and baking in Grasshopper.

4.2.4 Traceability Performance

To measure the impact that traceability data collection has on program running time we also ran programs with data collection enabled and disabled and measured the time each took to execute. The chart in Figure 13 shows a comparison of their times.

We can observe that running with traceability impacts the running time, increasing it by 10–50% which, depending on the example, can be considered between a small impact and a large impact. Like so, traceability data collection may need to be disabled to increase feedback when running complex programs. However, the impact is worth taking when the user needs to debug the program.

5. Conclusion

The current trend in architecture practice revolves around using programming as a way to automate repetitive tasks and to explore new design possibilities. These activities are often put under the umbrella of Generative Design (GD).

Tools used to support this design approach are mostly desktop applications but this is a leftover from a past that, in many other fields, is already being changed so that applications become web-based. This change has the significant advantage that applications can be much more easily updated and can be used with whatever machine it is available at that moment. Moreover, with teams becoming more dispersed geographically, the need arises for using a distributed solution in which team members can still collaborate. Web applications have the advantage of being platform-independent, requiring only a web browser to be used.

Architects need both 3D modeling and programming capabilities to have an environment where they can do GD. To have a better idea of the features necessary for such environment, Section 2 explored programming environments for various domains, including GD, and also 3D modeling applications.

Firstly, we compared these applications according to their application domain and the purpose they give to programming. Afterwards, we also compared them according to the editing and the programming paradigm, the persistence and the collaboration support and code editing features.

Following the introduction and comparison of those environments, we described the problem we would address and we proposed our solution in Section 3, which is composed of two components: (1) a web application, used by the architect to create programs; and (2) a desktop application that, when run, allows the web application to export results of those programs to CAD applications.

After defining the architecture, we created a test implementation called Luna Moth. To help the programming task, Luna Moth runs programs when they change, allows literals to be adjusted by clicking and dragging, and highlights the relationship between program and results in both directions.

In Section 4, we presented some examples of programs that were created using Luna Moth. Additionally, we tested Luna Moth's performance by measuring the program running and export times and comparing them to running times on Rosetta, OpenJSCAD and Grasshopper. Taking those measurements into consideration, we concluded that running programs in Luna Moth is faster than running these IDEs. However, exporting from Luna Moth is slower than running in Rosetta. We also measured the effect that keeping track of traceability data has on program running times. As a result, we concluded that collecting traceability data makes programs around 10–50% slower, which is a great achievement considering that every function call is being recorded.

In conclusion, with Luna Moth, architects are able to explore GD without needing to install and update the programming environment and avoiding

[18] Grasshopper only displays previews of results in Rhinoceros while programs are being developed. To export results to Rhinoceros, Grasshopper has a feature called "Bake Geometry".

the installation and use of heavy-weight CAD applications, which become slow for GD exploration.

Due to the achieved performance, we are convinced that web browsers can be seen as serious candidates for the next generations of design tools. We are also convinced that architects need tools dedicated to GD that avoid the complexity of current CAD applications to enhance the interactivity of the programming environment.

5.1. Future Work

Our solution can support the GD approach to architecture, however, there are still areas that can be improved.

There is room for improvement in the programming environment. Some future work includes improving the programming experience by performing static analysis and code completion, supporting addition of illustrations to programs as in [5] and IPython, supporting multiple programming languages, improving the debugging experience, improving the environments traceability, and supporting common modeling primitives. User testing should also begin to help steer the environment development according to the needs and difficulties of the users and, therefore, make sure it remains both useful and easy to use.

Apart from opportunities in the programming environment, there are also some opportunities for improvement in security and communication between components. On this side, future work includes sandboxing of user programs, implementation of remote persistence and collaboration, and improvement of the export process's performance.

References

[1] J. Maeda, *Design by Numbers*. Cambridge, MA, USA: MIT Press, 2001.

[2] K. Terzidis, *Expressive Form: A Conceptual Approach to Computational Design*. Taylor & Francis, 2003.

[3] A. Leitão, R. Fernandes, and L. Santos, "Pushing the Envelope: Stretching the Limits of Generative Design," *Blucher Design Proceedings*, vol. 1, no. 7, pp. 235–238, 2014.

[4] A. Leitão, L. Santos, and J. Lopes, "Programming languages for generative design: A comparative study," *International Journal of Architectural Computing*, vol. 10, no. 1, pp. 139–162, 2012.

[5] A. Leitao, J. Lopes, and L. Santos, "Illustrated Programming," in *ACADIA 14: Design Agency, Proceedings of the 34th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA)*, (Los Angeles), pp. 291–300, 2014.

[6] C. Granger, "Lighttable." retrieved from http://lighttable.com/, 2015. [Online; accessed 25-November-2015].

[7] R. Hickey, "The clojure programming language," in *Proceedings of the 2008 symposium on Dynamic languages*, p. 1, ACM, 2008.

[8] F. Pérez and B. E. Granger, "IPython: a system for interactive scientific computing," *Computing in Science and Engineering*, vol. 9, pp. 21–29, May 2007.

[9] C. Reas and B. Fry, *Processing: A Programming Handbook for Visual Designers and Artists*, vol. 54. Mit Press, 2007.

[10] R. Aish, "Designscript: origins, explanation, illustration," in *Computational Design Modelling*, pp. 1–8, Springer, 2012.

[11] J. Lopes, "Modern Programming for Generative Design," 2012.

[12] J. Lopes and A. Leitão, "Portable generative design for CAD applications," *Integration Through Computation - Proceedings of the 31st Annual Conference of the Association for Computer Aided Design in Architecture, ACADIA 2011*, pp. 196–203, 2011.

[13] S. Rugaber, "Program comprehension," *Encyclopedia of Computer Science and Technology*, vol. 35, no. 20, pp. 341–368, 1995.