A Browser-based Programming Environment for Generative Design

Pedro Alexandre Fonseca Alfaiate

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. António Paulo Teles de Menezes Correia Leitão

Examination Committee

Chairperson: Prof. Alberto Manuel Rodrigues da Silva
Supervisor: Prof. António Paulo Teles de Menezes Correia Leitão
Member of the Committee: Prof. Daniel Jorge Viegas Gonçalves

May 2017

Agradecimentos

Agradeço...

Ao Prof. Dr. António Leitão, por ter aceite orientar a minha tese e pelo acompanhamento ao longo deste processo.

Aos elementos do Grupo de Arquitetura e Computação, pelo feedback que deram na escrita e na preparação de apresentações, em especial à Inês Caetano, à Sofia Feist, à Renata Castelo Branco, à Maria Sammer, ao Bruno Ferreira, e ao Guilherme Barreto.

A toda a minha família, aos meus pais e avós, por me terem ajudado a crescer durante todos estes anos.

À Fundação para a Ciência e Tecnologia (FCT) e ao INESC-ID pela atribuição de uma bolsa de investigação no âmbito do contrato PTDC/ATP-AQI/5224/2012.

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

Resumo

Desenho Generativo (DG) permite aos arquitetos criar designs usando uma abordagem baseada na programação. Os ambientes de DG atuais baseiam-se nas aplicações de *Computer-Aided Design* (CAD) existentes, como o *AutoCAD* e o *Rhinoceros 3D*, que, devido à sua complexidade, são lentos e não dão o *feedback* necessário para que os arquitetos explorem DG. Além disso, os ambientes de DG atuais estão limitados pelo facto de terem de ser instalados e, consequentemente, não estarem acessíveis a partir de qualquer computador. Seria vantajoso para os arquitetos ter um Ambiente de Desenvolvimento Integrado (IDE em inglês) para DG na *web*, acessível em qualquer computador sem instalação e sendo mais interactivo que os ambientes para DG existentes.

Esta tese propõe um IDE para DG baseado em tecnologias *web*. A componente principal é uma página *web* que contem uma interface para edição de programas que, por sua vez, permite ao arquiteto criar programas e ver os resultados em 3D. Para tornar a experiência de edição mais intuitiva, o IDE também reexecuta os programas assim que são modificados, permite que os literais numéricos sejam ajustados clicando e arrastando, e realça a relação entre o programa e os resultados. O IDE também inclui uma aplicação secundária para permitir a exportação de resultados para aplicações CAD instaladas no computador do arquiteto.

Com esta abordagem, conseguimos implementar um ambiente para DG que está acessível a partir de qualquer computador o qual não só oferece uma interface de edição interactiva, como também se integra facilmente no fluxo de trabalho do arquiteto. Ainda, no que diz respeito a tempos de execução de programas, o ambiente tem um bom desempenho que consegue ser uma ordem de grandeza mais rápido que os IDEs para DG atuais.

Palavras-Chave: Desenho Generativo, Tecnologias Web, Ambientes de Desenvolvimento Integrado, Arquitetura

Contributions

During the development of this master thesis, two scientific contributions were made, namely:

- *Luna Moth: A Web-based Programming Environment for Generative Design*, accepted in the 35th International Conference on Education and Research in Computer Aided Architectural Design in Europe;
- *Luna Moth: Supporting Creativity in the Cloud*, submitted to the 37th Annual Conference of the Association for Computer Aided Design in Architecture;
- *Luna Moth*, a prototype IDE for the development of Generative Design programs.

Contents

List of Tables

List of Figures

List of Listings

Acronyms

AJAX Asynchronous JavaScript and XML. 35

API Application Programming Interface. 8, 18, 23, 30, 32, 34

AST Abstract Syntax Tree. 26, 28, 32

BOT Behavior Object Tag. 10

CAD Computer Aided Design. xv, 1–5, 15, 18, 19, 23, 25, 33, 34, 49–52, 54

CLI Command Line Interface. 12

CSG Constructive Solid Geometry. 45, 54

DSP Digital Signal Processor. 8, 20

GD Generative Design. xv, 2–5, 7, 8, 11, 13, 15, 17–19, 21–23, 25, 30, 32, 37, 46, 49, 51–53

GUI Graphical User Interface. 12

IDE Integrated Development Environment. xvi, 2–5, 7, 8, 13, 18, 23, 34, 46, 48, 49, 52, 53

JIT Just-In-Time. 46

PDE Processing Development Environment. xv, 13, 14

REPL Read Eval Print Loop. 7, 21

SDF Signed Distance Function. 19

UI User Interface. xv, 10, 19, 20, 23–25, 28, 51

Chapter 1

Introduction

1.1 From Paper to Bits

Through the years, computers have been taking more ground in the field of architecture. It began with the appearance of Sketchpad[1] in the 60s that showed that computers could be used to create drawings interactively. After the appearance of desktop computers, drafting software like AutoCAD popularized Computer Aided Design (CAD) in architecture. As opposed to drawing by hand, using computers to create these drawings allowed architects to make changes without having to redraw the drawings, therefore saving them much time. Still, computers were only used for creating technical drawings, leaving most of the thinking about 3D space outside of the computer.

As computers became powerful enough to display 3D graphics, and 3D modeling started to appear, it became possible to model buildings in the computer and preview them in interactive 3D views. Having a 3D model of the building and being able to explore it – to see it from many angles – makes it easier and much more intuitive for the architect to think about the building. What was only possible to do with a physical model or a perspective drawing, was now faster and more affordable.

This has allowed CAD applications to be used to create visualizations and documentation for architectural projects. However, modeling a building still requires time-consuming and repetitive tasks that are not trivial to accomplish using the functionalities provided by a 3D modeling software.

1.2 Scripting

The recognition of this problem has led to the emergence of programming for 3D modeling software.[1] Using programming, the architect is able to describe what he wants to model in a program 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. As a result, the architect gets to do more in less time.

Some examples of programming languages being used in CAD applications are AutoLisp and Visual Basic for AutoCAD, GDL[2] for ArchiCAD, Python and Grasshopper for Rhinoceros 3D, and Dynamo for Revit.

[1] Also called scripting.

1.3 Generative Design

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[3] that can be used by architects, as shown in [4].

Furthermore, as stated in [5], 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 the understanding of the problem improves or as the project's needs change. By using GD, changes only need to be made to programs instead of 3D models. Programs are unambiguous parameterized representations of designs, which only need small changes to parameters or functions to express changes in designs. In contrast, if 3D models are used, expressing changes in designs requires changing many more parts of those 3D models. However, in order to create the programs that generate the architect's designs, the architect needs to have a programming environment, or Integrated Development Environment (IDE), that lets him write them and see their results.

1.4 IDEs for Generative Design

To create GD programs, the architect needs to use a programming language, its runtime environment (like the CAD application where the models are generated), and an IDE.

The programming language defines what is a program, that is, what concepts can be part of a program and how they can be combined to perform tasks that can be understood and executed by a computer. For example, it may define a program as a sequence of operations that need to be performed, and define which primitive operations exist, like creating a box or moving an object.

The IDE provides tools – editors, compilers, debuggers, among others – that let the architect create programs. A basic IDE may have a text editor, where the architect types a textual representation of the program, and an interpreter, that interprets the program and performs the desired operations. Examples of GD IDEs include Grasshopper 3D[2] for Rhinoceros 3D,[3] Dynamo[4] for Revit,[5] and Rosetta[6].

In the case of GD IDEs, there are specific requirements that need to be fulfilled to make them useful for exploration of GD, since these IDEs have to support the architect while he creates programs and explores design possibilities. First of all, they have to be able to display geometric results, otherwise, the architect will not be able to see what his programs generate. Secondly, the architect needs to be able to see the effects from changes to parameters or to any part of the program. This will require the environment to run quickly programs, generate results, and display them to the architect, thus giving him immediate feedback. Apart from this, the environment also needs to make it easy for the architect to build his program and correct the bugs that might appear. It may do so by showing him the available functions or by highlighting results from a given part of a program. Finally, the IDE needs to support a programming paradigm that is easy for the architect to understand.

Most GD IDEs use either data-flow programming or procedural programming. The first is often supported by a visual editing environment, while the second is often supported by a textual editing

[2] http://www.grasshopper3d.com/ (last accessed on 10/05/2017)
[3] https://www.rhino3d.com/ (last accessed on 10/05/2017)
[4] http://dynamobim.com/ (last accessed on 10/05/2017)
[5] http://www.autodesk.com/products/revit-family/overview (last accessed on 10/05/2017)

environment. Moreover, most IDEs are aimed at the creation of programs in one specific programming language, as is the case of Grasshopper 3D and Dynamo. There are, however, some IDEs that can be used to create programs in several programming languages, such as Rosetta, that supports languages like Racket[7], Python[8] and AutoLisp.

In order to help architects with the creation of GD programs, visual programming IDEs provide tools that are capable of showing all the nodes/functions that can be added, and showing the relationship between nodes and their results. They are, therefore, more popular among novices since they do not require as much training to be used when compared with textual programming IDEs. However, as stated in [9], visual programming languages do not scale well for big GD projects when compared to textual programming languages. Textual programming languages have abstraction 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 has been done in Rosetta to solve this problem, as shown in [10], where it is possible to see which objects each part of the program has generated and which parts of the program were responsible for generating each object.

Up until now GD IDEs have been desktop applications. The next section shows how this affects the architect.

1.5 Disadvantages of Current CAD Applications

GD IDEs can serve the needs of architects. Even if they need to be on the move, architects can still use a laptop to run them. However, GD IDEs do have problems coming from their desktop application nature.

One of these problems is the fact that the user is often aware of the update process. Software needs to be updated, be it to correct errors or add new features. However, this means that the user may have to choose when to update it or that it may update at inappropriate times. Consequently, this may lead to users not updating their applications.

Another problem is that desktop applications can be pirated since they are distributed as complete packages that run solely on the user's computer. Like so, it is easier to create a slightly modified version that bypasses licensing checks, therefore, allowed users that have not purchased a license use the application.

Apart from these problems, current CAD applications also deal poorly with collaboration. Although they allow for remote collaboration on their documents with technologies like AutoCAD 360[6] or A360,[7] the same is not true for GD IDEs. To collaborate remotely in GD, architects need to resort to generic collaboration services. They can use services like Skype[8] to speak with teammates or present their work, and they can use file sharing services, like Dropbox,[9] to build GD programs collaboratively. Still, it would be preferable to have collaboration integrated into the application.

Web applications are less prone to these problems. The web has seen a big increase in popularity which has become even stronger with the standardization of both new and existing web technologies such as HTML5[11] and WebGL[12] that allow web applications, accessed using web browsers, to achieve user experiences on par with desktop applications. 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

[6] http://www.autodesk.com/products/autocad-360/overview (last accessed on 10/05/2017)
[7] http://www.autodesk.com/products/a360/overview (last accessed on 10/05/2017)
[8] https://www.skype.com (last accessed on 10/05/2017)
[9] https://www.dropbox.com/ (last accessed on 10/05/2017)

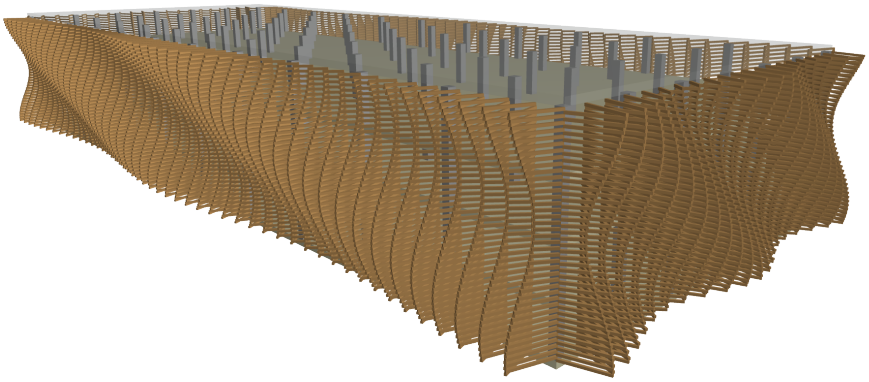


Figure 1.1: Rendering of a building design made using GD.

when Microsoft Office 365 appeared. Furthermore, complete CAD web applications have also appeared. One example is OnShape,[10] a CAD application for Product Design/Engineering completely accessible from the web browser.

In addition to providing the functionality of the desktop application, web applications also enhance collaboration by keeping every document accessible from the web, keeping track of all versions of documents, and allowing people to work simultaneously on the same documents. This support for collaboration makes it easy for users to collaborate effectively.

On top of that, web applications can also be transparently updated since users only access their web pages, meaning that they are always using the latest update of the application. Moreover, since web applications run on computers outside of users' control, they are less exposed to piracy. Like so, web applications also solve the desktop application problems described earlier. This suggests that GD IDEs should become web applications.

Becoming web application also has the added bonus of being available on any computer, which could prove useful if architects use many computers or if they want to use the IDE as a sketchbook for GD ideas.

Apart from problems with updates, piracy, and collaboration, CAD applications lack performance. Since CAD applications were initially developed for human interaction, they were not designed to handle the enormous amount of elements that GD programs generate, becoming slow quickly as a number of elements increases. In order to do GD, architects need to have an interactive environment where they can get feedback from their programs as fast as possible. If the CAD application takes too long to generate the results, then the GD environment will not be able to give this feedback.

This problem often reveals itself when working with Rosetta. For example, when working on a building design like the one shown in Figure 1.1, the time needed for generating the model increases when a CAD backend like AutoCAD is used, as seen in Figure 1.2. By looking at the figure, we can see that running times for CAD backends (AutoCAD, Rhinoceros, and SketchUp) span at least a few seconds.[11] Like so, architects will have to wait before they see the updated results of their programs, thus, the immediate feedback is lost.

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

[11] OpenGL viewer is a backend that only produces a visualization of the results, bypassing CADs and, therefore, achieving lower running times.

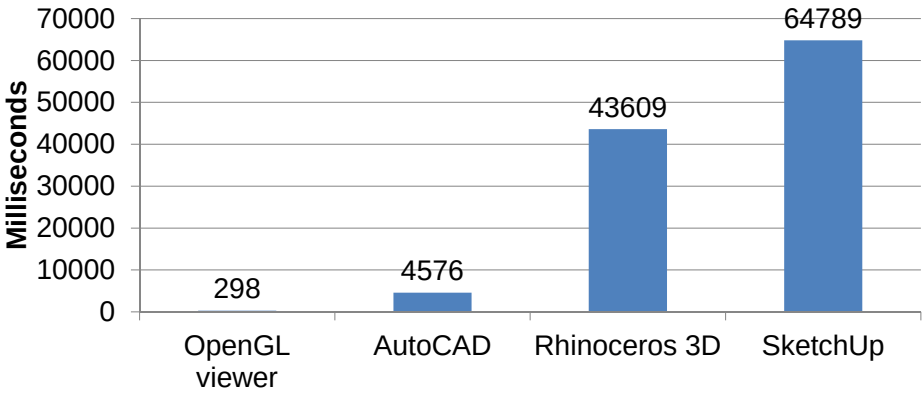


Figure 1.2: Generation times for different CAD applications using Rosetta.

1.6 Goals

A GD IDE needs to have more **accessibility**, not requiring its users to install or update it and being available in any computer, and needs to be more **interactive**, letting architects explore GD easily, giving them feedback and showing them the relationship between program and results. Moreover, it needs to **integrate** easily with the CAD applications already used by architects, so they can still integrate their GD experiments into their normal workflow.

This thesis aims at increasing 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 the implementation of a companion application for allowing models to be exported to traditional CAD applications. In this way, architects can write their GD programs and visualize the results without worrying about installing and updating the IDE, and can easily integrate results into their normal workflow by using the companion application.

1.7 Document Structure

The remainder of this document is structured as follows:

Chapter 2 explores related work regarding domain-specific IDEs, with emphasis on those for 3D modeling. A comparison is also presented to show their differences and similarities.

Chapter 3 presents the problem and the details of the solution, its components and how they relate to each other. It also explains how the most important features were implemented.

Chapter 4 evaluates the solution according to the models it can produce and to its performance.

Chapter 5 concludes the document, presenting the main aspects of each chapter and presenting some directions to be considered for future work.

Chapter 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 Impromptu

Impromptu[13, 14] is a programming environment developed to explore manipulation of musical structure in live performance; an IDE for musicians and sound artists.

Using Impromptu, live performance takes place as a musician programs algorithms that produce music in front of an audience. During his performance, the musician writes the program that produces the sounds and chooses the right moments to make changes to move between different parts of the music.

To enable live programming, Impromptu brings together four components, as stated in [13]: an audio synthesizer, a real-time scheduling engine, a Scheme interpreter, and an IDE. The first three components make up the runtime environment, responsible for producing the sound, while the last provides an user interface, where the musician writes code and sends it to the runtime in much the same way as using a Read Eval Print Loop (REPL).

As he sends code, the musician builds the algorithm that produces the music incrementally.[1] To be able to keep producing sound for the audience, he can program simple patterns, send them to the runtime environment, and then change them as he programs more elements to compose the music. As he adds more elements, he can start to make small changes to some of them which will be immediately audible.

The immediacy from program changes to audible ones makes it easy to understand the connection between code and sound, and also lets the musician experiment with new ideas. Similarly, someone doing GD will also benefit from having this immediacy in his IDE.

[1] The code sent to the runtime can define/redefine functions or variables, and schedule audio or functions to be played or run later.

The separation of components referred earlier also allows several musicians to collaborate at the same time in a performance[13]. With their computers connected to the same runtime environment, they can build different parts of the music or, more interestingly, make changes to the others' parts of their shared program. This is also an interesting aspect to explore in architecture, where several architects can work on a GD program at the same time, each one showing his ideas on how to evolve the design.

There is, however, a major difference between composing music and architecture. As sound is an ephemeral phenomenon, music's complexity comes from how sounds are organized in time as opposed to architecture's complexity, which comes from how shapes are organized in space. Like so, the instantaneous feedback must be adapted to meet architecture's needs.

Impromptu is in the process of being replaced by Extempore.[2] Instead of having its own editor, Extempore's runtime exposes its Application Programming Interface (API) publicly and has plug-ins for several popular code editors. Extempore also includes a second programming language for programming Digital Signal Processors (DSPs), which have stricter real-time constraints.

2.1.2 LightTable

LightTable[3] is a code editor for the Clojure programming language[15] and is an example of a desktop application that uses web technologies for its implementation. More specifically, it uses nw.js[4] as its runtime, allowing it to be implemented using JavaScript, HTML markup and CSS. LightTable is written in ClojureScript[16], a subset of Clojure that compiles to Javascript.

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 a metaphor.[5] The metaphor comes from looking at the way work is done in other fields of engineering, where engineers spread all materials relevant to their work over large tables, from tools to reference information. 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. In one of its experimental versions, LightTable also supported displaying running programs in the table. Figure 2.1 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 interesting feature deals with making navigation in Clojure code bases easier. In Clojure, functions are defined inside namespaces and all Clojure definitions (functions, variables, macros) are stored in text files. Navigating among definitions and the current namespace structure should not get in the way of editing code. To make editing easier, LightTable provides a *namespace browser* that allows programmers 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. Figure 2.2 shows an experiment where these two are used.

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. Figure 2.3 shows an example of such functionality. This functionality is part of LightTable's *instarepl*, which constantly evaluates its contents, showing results and data-flow.

[2] http://extempore.moso.com.au/ (last accessed on 10/05/2017)
[3] http://lighttable.com/ (last accessed 10/05/2017)
[4] http://nwjs.io (last accessed on 10/05/2017)
[5] http://www.chris-granger.com/2012/04/12/light-table-a-new-ide-concept/ (last accessed on 10/05/2017)

Figure 2.1: LightTable's drafting table. A game is being run inside it while some of its code is displayed in separate editors.

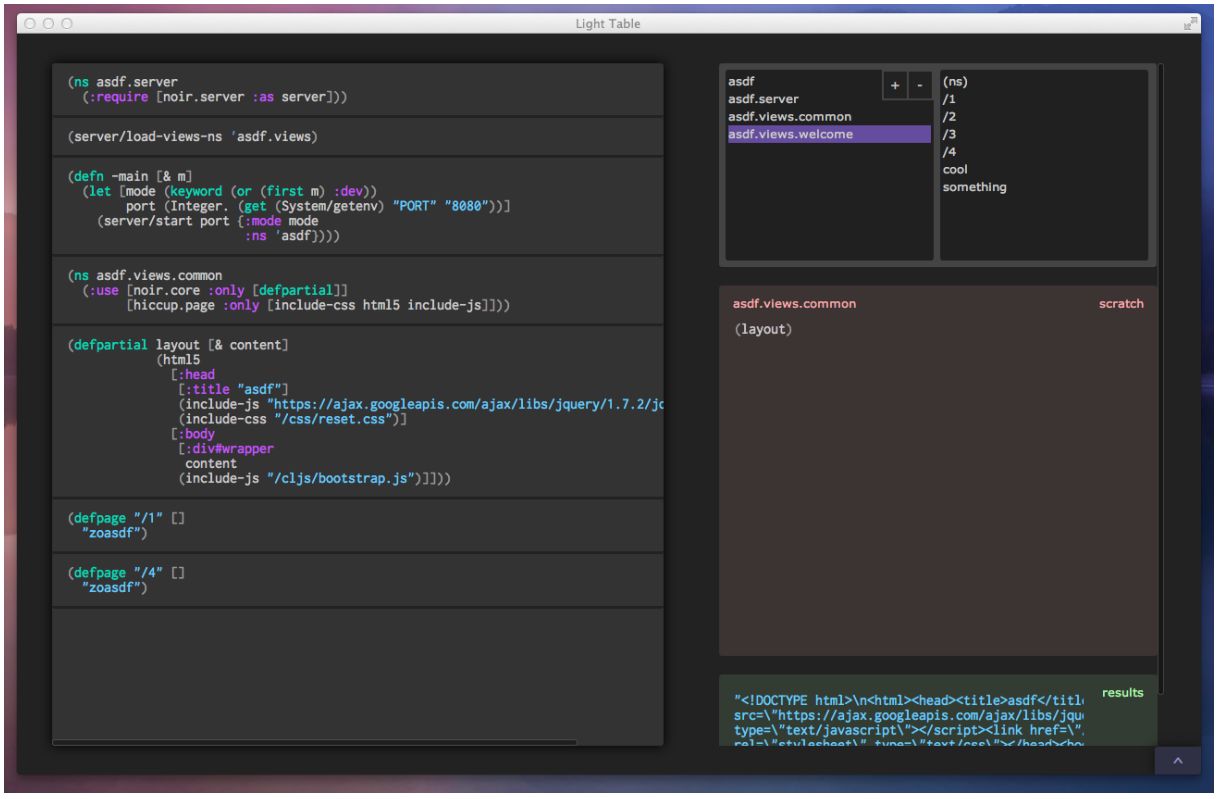


Figure 2.2: An experiment showing a *code document* on the left and a *namespace browser* on the top right.

Figure 2.3: An example of LightTable's value overlaying. The occurrences of the variables *a*, *b* and *ab* were replaced with their values while evaluating the expression *(x 3 7)*, where function *x* is applied to arguments *3* and *7*; the result of this expression is also overlaid.

This last feature runs into a problem when a function is called more than once. While it should display the data-flow of all calls, it only has space for one, since there is only one function definition. Moreover, this problem extends to any part of the code that runs more than once.

Finally, the way LightTable is structured is also relevant as it makes it possible to modify its behavior at runtime. The main design pattern used in LightTable is the Behavior Object Tag (BOT) pattern. As illustrated by one of its developers,[6] LightTable can be described as a set of *Objects*, each having a set of *Behaviors* and being tagged with a set of *Tags*. The *Behaviors* describe how an *Object* reacts when events are raised on it, and *Tags* are groups of *Behaviors*. When an event is raised on an *Object* both its *Behaviors* and those from its *Tags* are notified.

2.1.3 IPython

IPython[17] 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 2.4, IPython's notebooks can contain not only source code but also results and rich text. The 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 an 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.

As its name suggests, IPython's main programming language is Python[8]. Nevertheless, other programming languages can be used in IPython, particularly those popular in the scientific community like R, or Julia.[7]

One trend in its community, supported by IPython notebooks, is to make results in publications more reproducible. Typically, the code used to compute the results showed in publications is not made available to the public, so it is difficult to verify them. Instead of just publishing a PDF or making a blog post, authors also publish their IPython notebooks, thus allowing everyone to run their source code. Having access to a working copy of the notebook, one can also experiment with it to better understand it, form own conclusions, and find potential errors.

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

[6] http://www.chris-granger.com/2013/01/24/the-ide-as-data/ (last accessed on 01/11/2015)

[7] More language kernels can be found in IPython's github page: https://github.com/ipython/ipython/wiki/IPython-kernels-for-other-languages (last accessed on 10/05/2017)

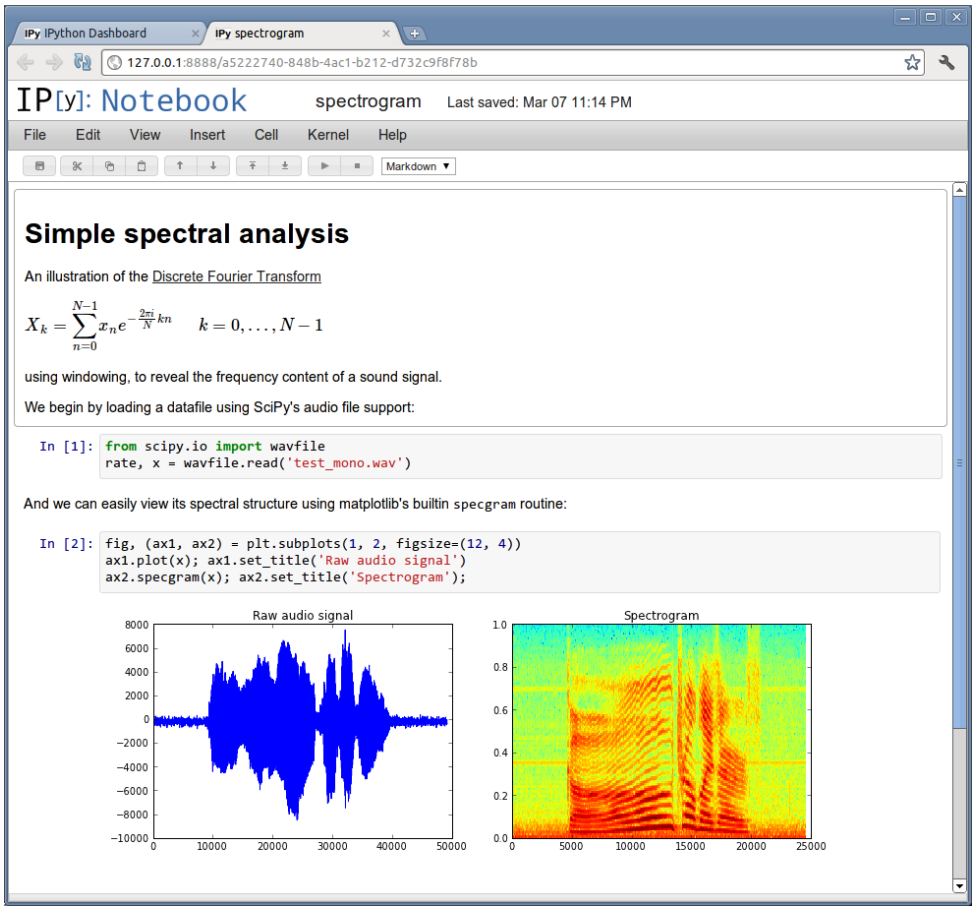


Figure 2.4: An IPython notebook with rich text, mathematical notation, source code and results from executing such code.

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[17].

IPython has given rise to the Jupyter project,[8] which aims to make IPython's model of interaction ready for the cloud. In the context of Jupyter, IPython is only concerned with providing an interactive Python experience.

2.1.4 OpenJSCAD

OpenJSCAD[9] is a project aiming to implement OpenSCAD[10] using web technologies. Instead of using OpenSCAD's language, OpenJSCAD uses JavaScript as its programming language. Most of OpenSCAD functionality is implemented in OpenJSCAD.

Similarly to the GD approach, to actually model in OpenJSCAD, one has to write a program that generates the model.

Being an environment for modeling 3D printed objects, OpenJSCAD supports importing and exporting 3D models from/to STL and AMF files, which are commonly used as 3D printing formats.

[8] https://jupyter.org/ (last accessed on 10/05/2017)
[9] http://openjscad.org/ (last accessed 10/05/2017)
[10] http://www.openscad.org/ (last accessed 10/05/2017)

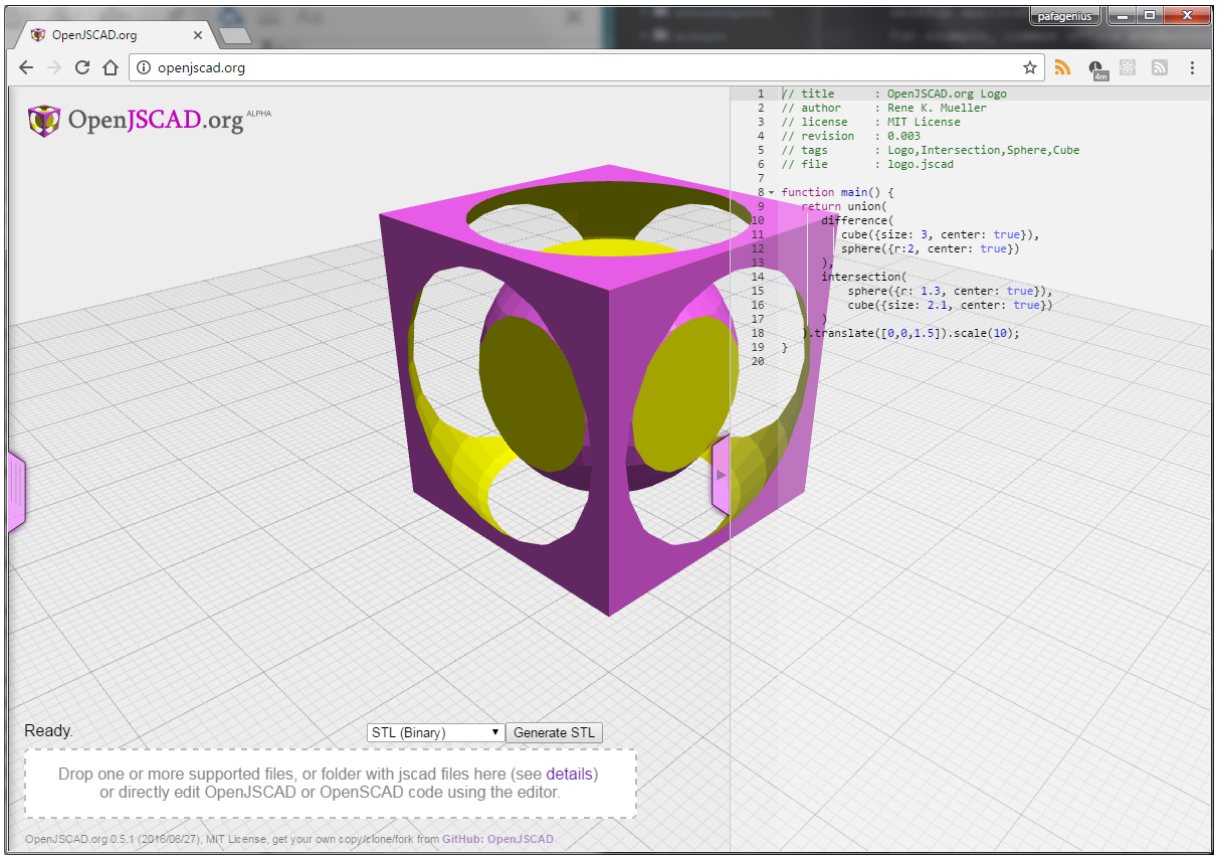


Figure 2.5: OpenJSCAD web page.

OpenJSCAD provides two user interfaces, one Command Line Interface (CLI) and one Graphical User Interface (GUI), the latter implemented as a web application (Figure 2.5).[11] Moreover, the first is used for batch processing, while the second integrates an editor for editing programs and a 3D view for viewing the corresponding results. By providing the Graphical User Interface (GUI) in a web page, OpenJSCAD can be used 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. This way, one can write either `a.union(b).translate([1,0,3])`, `translate(union(a,b), [1,0,3])` or `union(a,b).translate([1,0,3])` depending on what is more readable.[12]

Both OpenJSCAD's 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.

A problem that can arise while writing and testing JSCAD programs is that there is no help on getting the parameters for operations right, and it is frustrating to spend time trying to understand why a certain operation is not producing the desired result. This problem is even more relevant when most operations have multiple optional parameters and there are parameters that replace other parameters. Currently, the only help available is OpenJSCAD's documentation (a tour on its functionality), the online community around OpenJSCAD and, since Javascript is its language, web browsers' Javascript developer tools.

[11] https://github.com/Spiritdude/OpenJSCAD.org (last accessed on 10/05/2017)

[12] The first can be read as *a united with b, translated by [1,0,3]*, the second as *the translation of the union of a and b by [1,0,3]* and the third as *the union of a and b, translated by [1,0,3]*.

2.1.5 Processing

Processing[18] is a programming language and a development environment aimed at "promoting software literacy in the visual arts and visual literacy within technology".[13]

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 the Processing programming language, one needs to use a desktop application, the Processing Development Environment (PDE), which serves as Processing's IDE. As shown in Figure 2.6, the Processing Development Environment (PDE) can run Processing programs and includes a text editor with syntax highlighting and a text output display.

```
1  //Hello mouse.
2  void setup() {
3     size(400, 400);
4     stroke(255);
5     background(192, 64, 0);
6  }
7
8  void draw() {
9     line(150, 25, mouseX, mouseY);
10 }
```

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,[14] that extends the PDE to support Python, and p5.js,[15] that provides JavaScript libraries to create interactive web pages.

2.1.6 DesignScript

DesignScript[19] 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[19]. 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.

It supports both imperative (following instructions step-by-step) and associative (propagating changes in a dependency graph) control flows. The programmer can choose to have portions of the code following one type of control flow and other portions following the other.

[13] Quoting https://www.processing.org, 9/Nov/2015.
[14] http://py.processing.org/ (last accessed on 10/05/2017)
[15] http://p5js.org/ (last accessed on 10/05/2017)

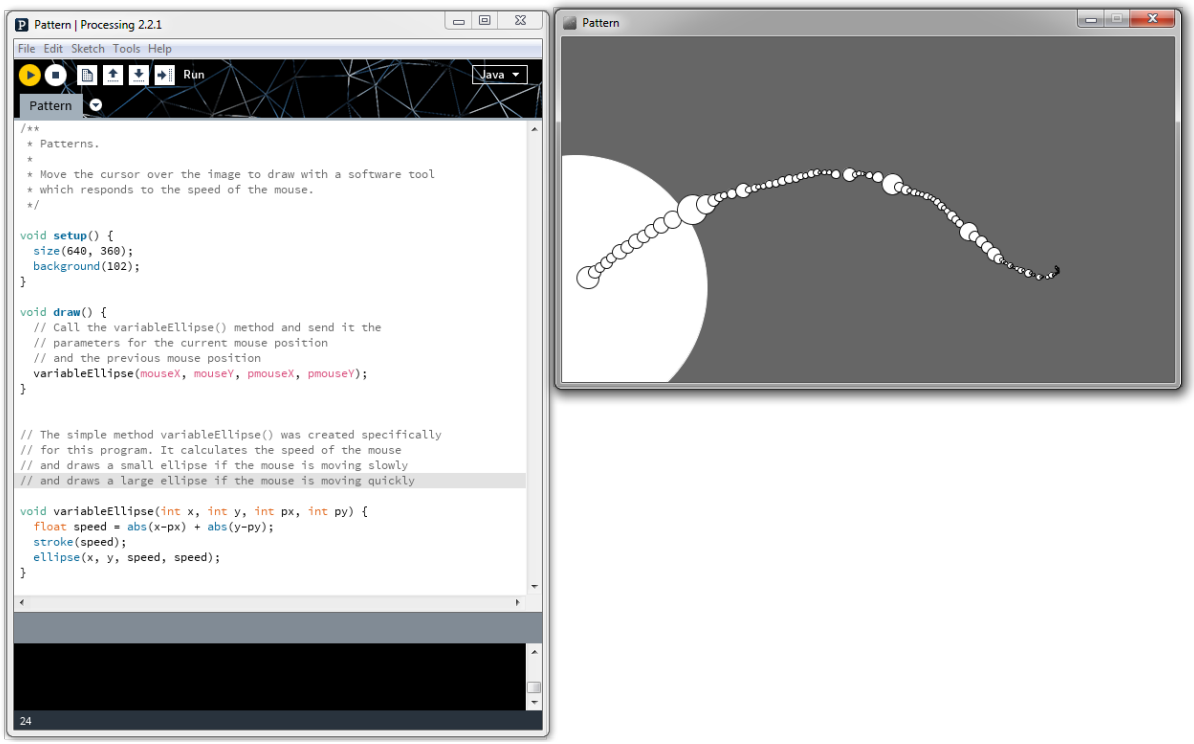


Figure 2.6: On the left: The PDE displaying an example *sketch* while it is being run. On the right: The drawing window to which the *sketch's* instructions are applied.

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.[16]

Editing DesignScript programs can be done either by writing statements or by creating a graph. The graph representation is a more natural visualization of the dependencies between the variables of a program, when the associative paradigm is being used. Function calls and variables are represented by nodes, drawn as boxes with slots, that can be connected by wires, which in turn represent data dependencies between nodes. The written representation of DesignScript is a sequence of statements that specify the relationship between a variable and other variables; defining functions, using the imperative paradigm and reassigning variables is also possible.

DesignScript is used in several environments, all of which are desktop applications. These include a textual editor in Autodesk AutoCAD (Figure 2.7), a dedicated graph editor called DesignScript Studio (Figure 2.8) and Dynamo (Figure 2.9), which integrates with Autodesk Revit. Both DesignScript Studio and Dynamo use graph based program editing.

Debugging DesignScript programs depends on the environment being used. The textual environment allows the user to follow the execution of the program step-by-step, while also allowing him to set watches and breakpoints. The graph-based environments allow the user to see the relationship between program and results by highlighting results of each node, and can also show him a list of the results of each node. Both the textual and the graph-based environments provide a preview of program results.

[16] If only one of its parameters is receiving a list instead of a value, the function is called once with each value on the list. If more than one parameter is receiving a list, the function is called with the values combined by either a cross product or a zip of the lists.

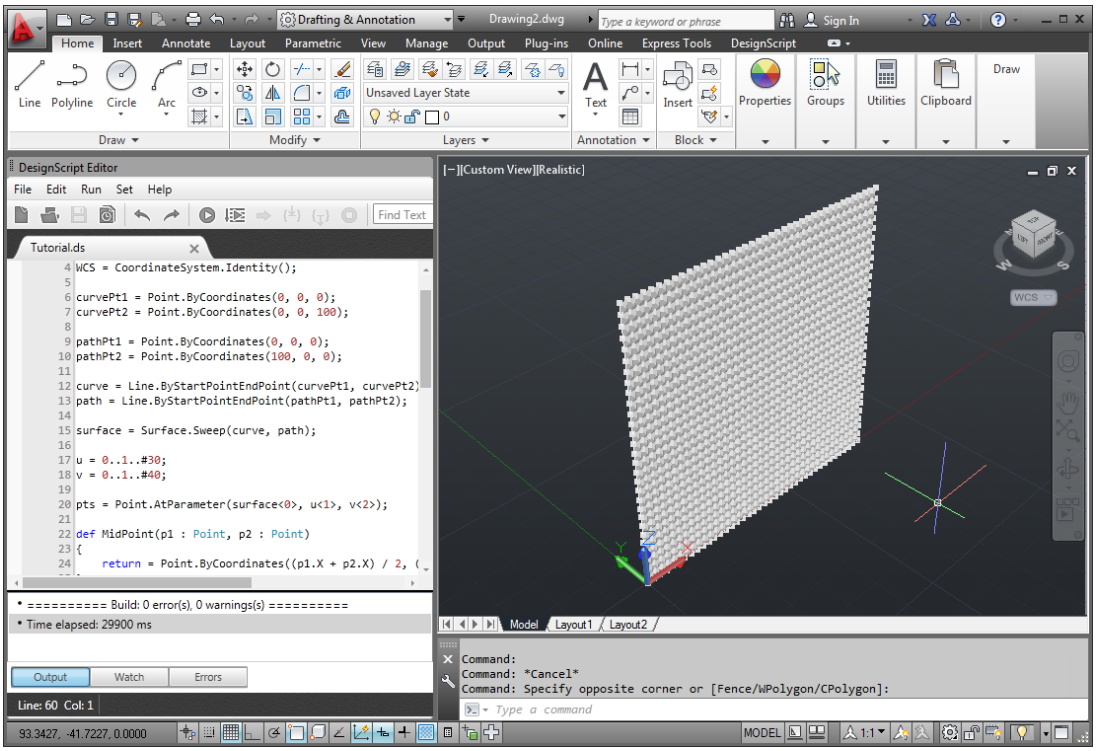


Figure 2.7: A DesignScript program being edited in a special text editor inside AutoCAD. This text editor provides auto-complete and a debugger.

2.1.7 Rosetta

Rosetta[6, 20], shown in Figure 2.10, is an environment for GD.

Most CAD applications used by architects allow them to make GD programs. Unfortunately, as each of these applications has specific primitive operations, programs written for one CAD application will only work in that application, a phenomenon called vendor lock-in. As an example, if the architect writes a program for AutoCAD, he cannot use it in Rhinoceros3D and, as a result, if he really wants to use it in Rhinoceros3D, he has to rewrite the program taking into account the differences between primitive operations of both CAD applications.

Rosetta aims to overcome the vendor lock-in phenomenon by allowing architects to write portable GD programs that generate equivalent models in any CAD application. It lets architects choose both the front-end programming language and the back-end application, where the primitive operations will be performed[6].[17]

With this, architects can start to use different CAD applications for different purposes. For example, they can use a faster but less functional CAD application during exploration, and switch to a slower one when they need more information or when ready to move on to a later design phase. Additionally, without vendor lock-in, architects also become free to share programs with others using different CAD applications.

Adding to the support of several programming languages, Rosetta also allows programs written in one programming language to use functionality defined not only in programs written in the same language, but also in 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.

[17]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

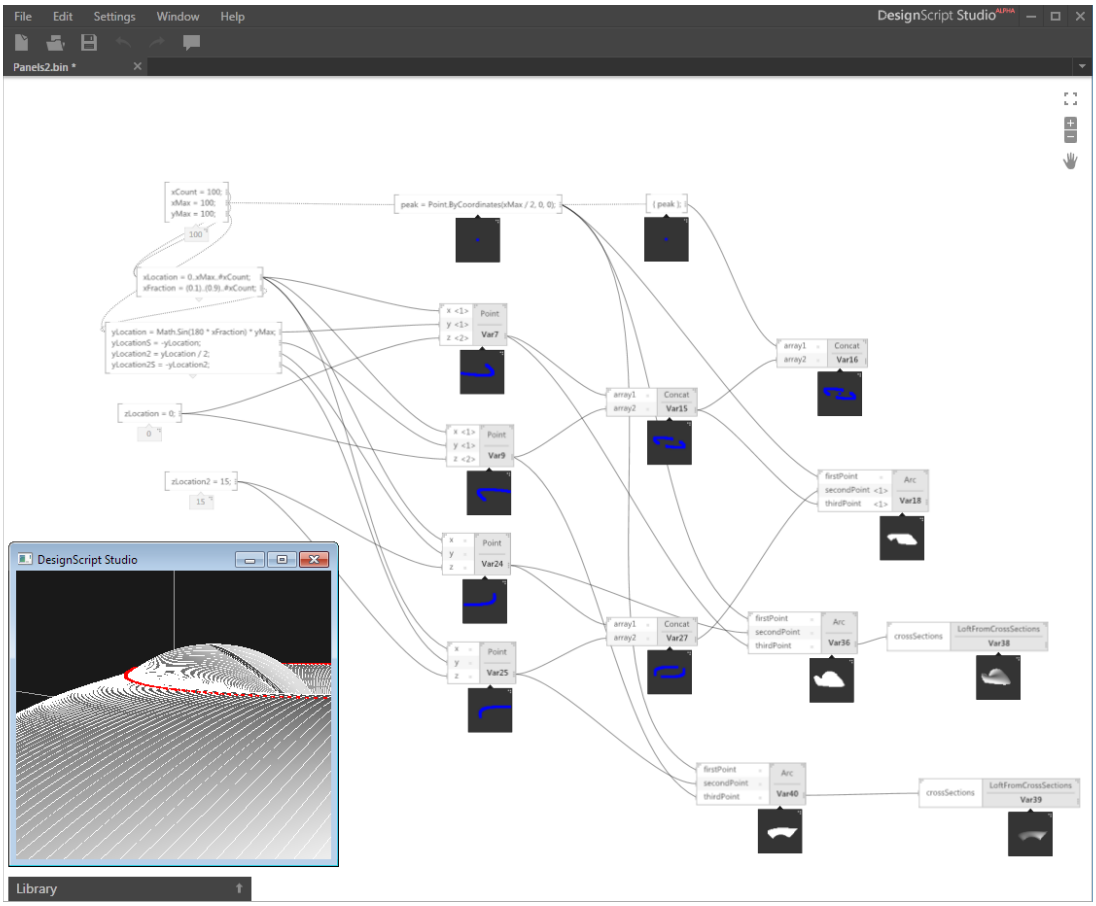


Figure 2.8: A DesignScript program as a graph in DesignScript Studio. Each node can display a preview of its results. To the bottom left corner is a preview of the whole program results and a folded library tab. The library tab contains everything that can be used in the program.

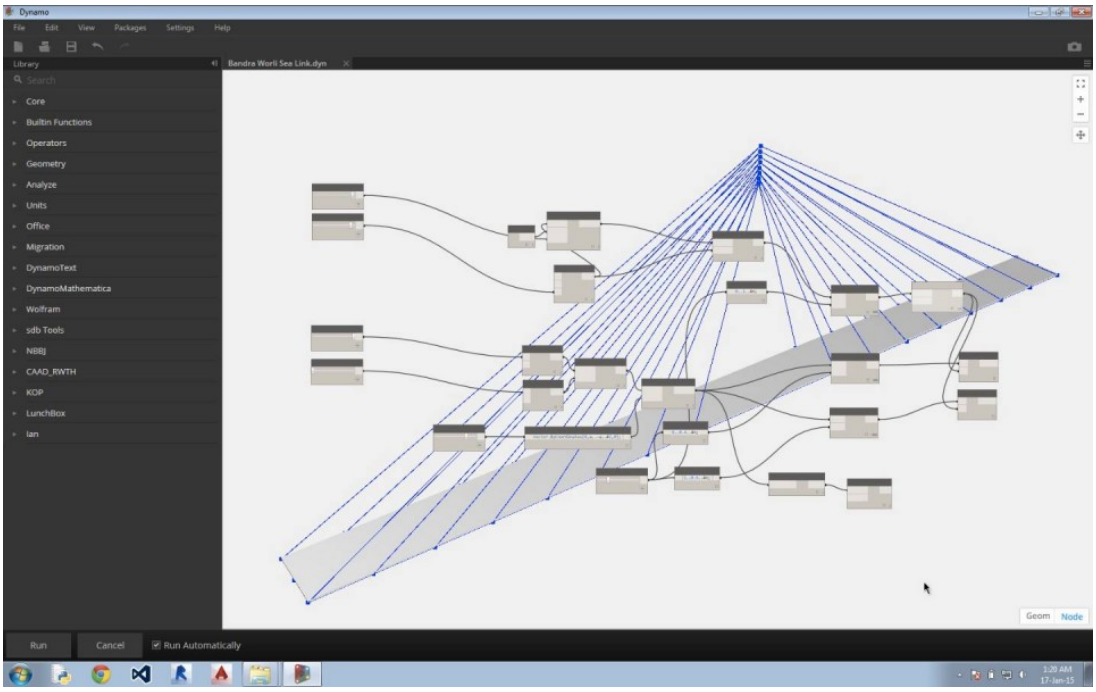


Figure 2.9: Another DesignScript program as a graph in Dynamo. Like in DesignScript Studio, a preview of the result of the program is displayed. A difference is that there is only one preview “canvas”; the preview from selected nodes is highlighted.

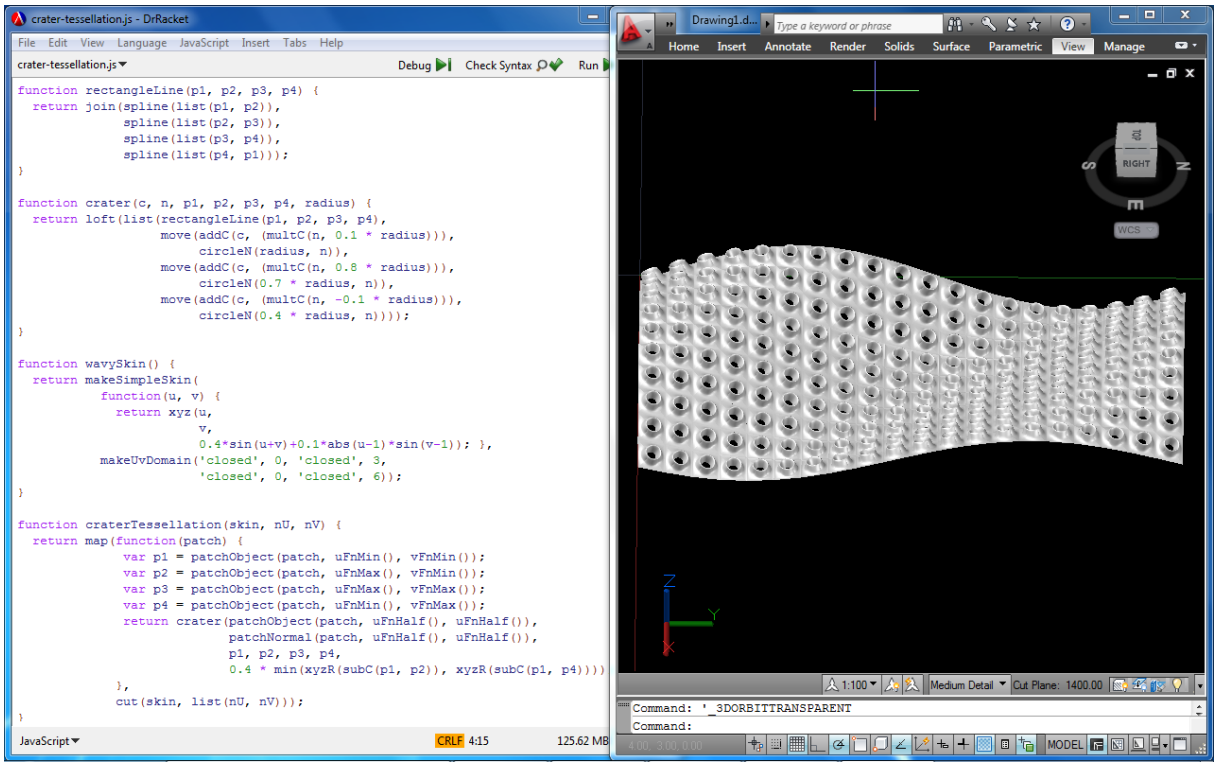


Figure 2.10: A Rosetta program (left) and AutoCAD displaying its results (right). The program is written in Javascript.

2.1.8 Clara.io

Clara.io[18][21] is a cloud application for 3D modeling and animation, similar to desktop software like Blender[19] and 3ds Max.[20] Being in this category, Clara.io is more suitable for artistic work as opposed to technical work. Nonetheless, it can be used in architecture projects to make architectural visualizations.

Using Clara.io closely resembles working with Blender and 3ds Max. Modeling is done by adding 3D entities to the scene's scenegraph and afterwards adding modifiers to them and/or directly manipulating them in 3D space(position, rotation, scaling, selection, grouping).

Apart from having features close to other 3D modeling software, as a cloud application, Clara.io also supports cloud storage of scenes, real-time collaboration, and integrates with VRay cloud rendering. Furthermore, Clara.io only requires a web browser compatible with HTML5[11] and WebGL[12], without any installation. Like so, its users are free to switch between computers without having to configure anything and without having to move their work.

Other aspect of Clara.io is its collaboration support. Due to the limitations of the files, users normally have to take turns at changing the scene to avoid edit conflicts. This makes the process of editing the scene much slower since most changes do not produce conflicts as they involve different parts of the scene and, therefore, could be made simultaneously. Clara.io's real-time collaboration lets users do just simultaneous edition. Moreover, it also has the benefit of each user always seeing the most up to date version of the scene.

When it comes to programming, Clara.io has support for scripting and allows the creation of plug-ins to extend its functionality.[21] It is possible to use Clara.io's scripting to do GD, however, if a user wishes

and also graphics libraries like TikZ and OpenGL.

[18] https://clara.io/ (last accessed on 10/05/2017)

[19] https://www.blender.org/ (last accessed on 10/05/2017)

[20] http://www.autodesk.com/products/3ds-max/overview (last accessed on 10/05/2017)

[21] https://clara.io/learn/sdk (last accessed on 10/05/2017)

to do so, he needs to learn the scene's data model and how to use the scripting API to manipulate it, which poses a considerable barrier when compared with existing GD environments.

2.1.9 OnShape

OnShape[22] is a cloud based CAD application that can be accessed using either a web browser or its mobile app. As opposed to the previous systems, OnShape is a CAD application for mechanical modeling and, as such, its users are familiar with CAD applications like SolidWorks[23] and Autodesk Inventor.[24] Being a mechanical modeling tool means that the focus is directed towards assembling individual objects instead of buildings, so the working scales are smaller and there is more information on how parts relate to each other.

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. It is also possible to create *branches* of the document when it is necessary to work on different design options or to work on changes in isolation. Later, changes made in a branch can be merged back into the main branch.

OnShape also includes a dedicated programming language, FeatureScript,[25] 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.

2.1.10 Mobius

Mobius is a visual programming environment for 3D modeling. It tries to make visual programming environments as powerful as textual programming environments by bringing associative and imperative programming together, and by supporting higher-order functions, common in textual programming languages[22].

In Mobius, programs are a combination of nodes in a data-flow graph that ultimately produces a 3D model. The architect can implement nodes by assembling an imperative procedure out of virtual building blocks, which can be loops, function calls or variable assignments. This way, Mobius tries to get the best of visual and textual programming languages.

To support higher-order functions, Mobius adds an output to every node that contains the function representing that node. This way, the node can be passed to other nodes as an input. Nodes receiving the function can then use it to, for example, generate an element of a grid parameterized by its coordinates or to generate the shape of a balcony.

Finally, Mobius is implemented as a web application, and like so is available to any computer connected to the Internet, not requiring any installation.

[22] https://www.onshape.com/ (last accessed on 10/05/2017)
[23] http://www.solidworks.com/ (last accessed on 10/05/2017)
[24] http://www.autodesk.com/products/inventor/overview (last accessed on 10/05/2017)
[25] https://www.onshape.com/featurescript (last accessed on 10/05/2017)

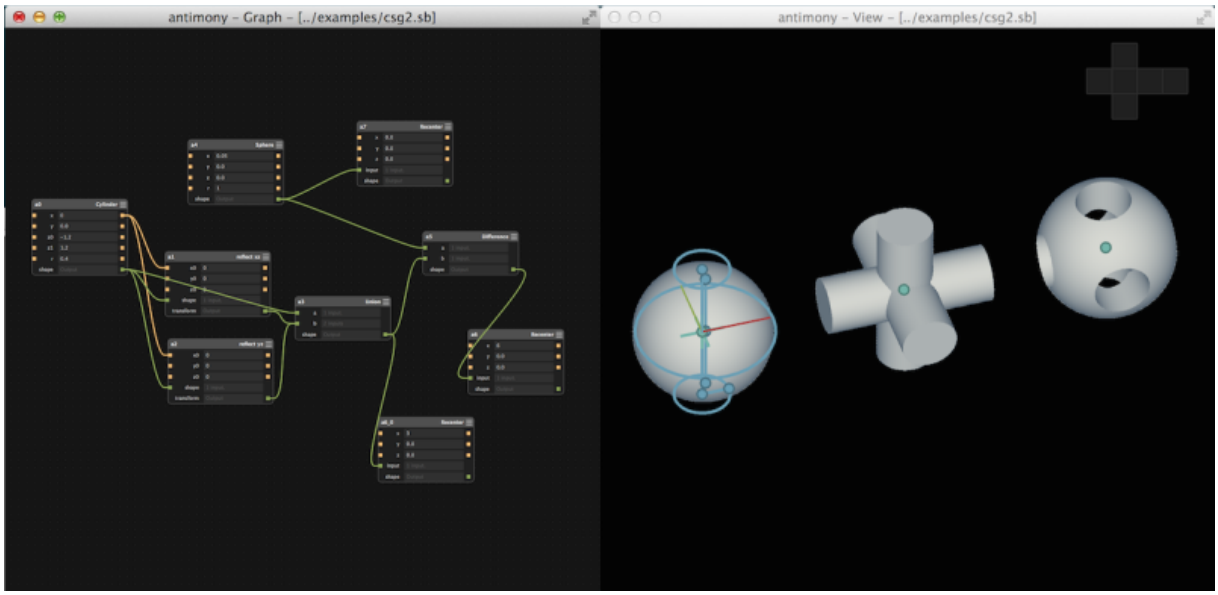


Figure 2.11: Antimony allows the user to adjust parameters using handles in the 3D view.

2.1.11 Antimony

Antimony[26] is a desktop application for solid modeling where models are described by data-flow programs. Programs are edited using a node-based representation like in visual programming environments. When editing a program, Antimony also displays handles in the 3D view that can be dragged with the mouse to change free node inputs, as shown in Figure 2.11.

Antimony was implemented as a tool to be used from design to fabrication, so it can generate files for machining or 3D printing its models.

It is possible to implement custom nodes by writing a Python program with custom directives. This program defines what are the inputs and outputs of the node, and what is its UI in the 3D view. As Antimony uses Signed Distance Functions (SDFs) as its representation of solids, shapes are defined by specifying a formula for the shape's SDF and its domain bounds.

2.2 Comparison

Not all of the presented environments are directly related to GD. We looked at this diverse set of environments to get the broader picture of domain-specific programming environments, 3D modeling and CAD applications, and some of their cloud counterparts.

In terms of application domain, the environments presented in this chapter cover a variety of fields that use programming for automating tasks. Table 2.1 shows a comparison of the application domain of each of them, as well as the purpose that programming takes in them.

Programming has a major role for users in most of the presented environments as seen in Table 2.1, with the exception of Clara.io and OnShape, where it is only used to extend the functionality available in the main UI.

The editing paradigms used in environments have been based on text editing, node graph editing, a combination of these two, or a combination of node graph and block-based editing. The type of editing paradigm each environment supports is shown in Table 2.2. Environments with textual editing provide different levels of *"editing power"*, starting with basic text editing with syntax highlighting (*"text"*

[26] https://github.com/mkeeter/antimony (last accessed on 10/05/2017)

Environment	Domain	Programming Purpose
Impromptu	music, live coding	schedule sound generation + implement DSPs
LightTable	code editing	general
IPython	scientific computing	general
OpenJSCAD	3D CAD	generate 2D/3D models
Processing	visual arts	draw, generate sound, react to input
DesignScript	CAD, architecture	generate 2D/3D models
Rosetta	CAD, architecture	generate 2D/3D models
Clara.io	3D modeling, animation	extend UI, automate modeling tasks
OnShape	3D CAD, engineering	implement custom features
Mobius	3D CAD	generate 2D/3D models
Antimony	3D CAD	generate 2D/3D models, implement custom nodes

Table 2.1: General environment comparison.

Environment	Editing Paradigm	Programming Paradigm
Impromptu	text	imperative
LightTable	++text	functional / imperative
IPython	++text	imperative
OpenJSCAD	text	imperative
Processing	text	imperative
DesignScript	text + node	associative
Rosetta	++text	imperative
Clara.io	text	imperative
OnShape	++text	imperative
Mobius	node + block	data-flow + imperative
Antimony	node + text	data-flow + imperative

Table 2.2: Comparison of paradigms used for programming. The "++" symbol means that the environment has supplemental features to the basic editing paradigm. The "+" symbol between paradigms means that the environment supports a combination of those.

in Table 2.2) and adding features like code completion, showing documentation, helping navigation, and expression evaluation (*"++text"* in Table 2.2).

Table 2.2 also shows the programming paradigms supported by each environment's language. They include imperative, functional, associative, and data-flow programming.[27] Visual environments use associative and data-flow programming languages, while textual environments use imperative and functional programming languages.

Some environments have been implemented as desktop applications, while others have been implemented as web applications, as shown in Table 2.3. Implementing an application as a web page has the advantage of becoming accessible to any computer connected to the Internet, without requiring users to install or updated it.

In Table 2.3 we can also see that all web applications, except Mobius and OpenJSCAD, support remote storage directly. By supporting remote storage directly, it becomes easier to use the application regardless of the computer being used. Without this, users have to use either a physical storage device

[27] Associative and data-flow programming are similar. The difference is that associative programs build a dependency graph during their execution, propagating changes as needed, while data-flow programs have a static dependency graph and the data just flows from data sources to the connected nodes.

Environment	Platform	Remote Storage	Collaboration
Impromptu	desktop + local server	—	shared runtime
LightTable	desktop	—	—
IPython	web page (UI) + server(computation)	on server	—
OpenJSCAD	web app., command-line	—	—
Processing	desktop	—	—
DesignScript	desktop	—	—
Rosetta	desktop	—	—
Clara.io	cloud(web app.)	cloud, version controlled	share scenes, scene real-time collaboration
OnShape	cloud(web app., mobile)	cloud, version controlled	share documents, real-time collaboration
Mobius	web app.	—	—
Antimony	desktop	—	—

Table 2.3: Comparison of environments by their relation to the cloud.

Environment	Coding features
Impromptu	evaluate expression (like a REPL)
LightTable	instarepl; documentation; code completion; (code document); (draft table)
IPython	code completion; different cell types; REPL
OpenJSCAD	user-defined sliders;
Processing	syntax error annotation;
DesignScript	library tab; traceability (program→results)
Rosetta	user-defined sliders; REPL; traceability (program↔results)
Clara.io	—
OnShape	documentation; code completion;
Mobius	sliders on node inputs; traceability (program→results)
Antimony	autorun; 3D view handles

Table 2.4: Features / User experience comparison.

or a separate remote storage system.

Collaboration is only directly supported in Impromptu, Clara.io and OnShape, which allow for real-time collaboration, thus enabling their users to work on the same artifact simultaneously — i.e. an Impromptu runtime environment, a Clara.io scene or an OnShape model. For the rest of the environments, collaboration is the user's responsibility.

Table 2.4 presents a non-exhaustive list of features provided by each environment.

2.3 Problems to Address

In this chapter, we have presented environments that support activities related to programming, modeling and GD.

The environments supporting GD directly — namely Rosetta, DesignScript, OpenJSCAD, Mobius and, to some extent, Processing and Antimony — 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 manageable for bigger problems. Given that we want to support the generation of complex building models, we will focus on textual editing.

Moreover, the platforms in which the environments were implemented were either the desktop or the web applications. While desktop applications need to be installed in computers to be used, and have complex update procedures, web applications only require a web browser. Unfortunately, the existing web applications that support GD, namely OpenJSCAD and Mobius, lack editing features and maturity that desktop applications like Rosetta and DesignScript have. They do not provide the traceability present in both Rosetta and DesignScript, and they also do not provide the interoperability that Rosetta has.

As a result, there is no capable GD environment on the web that is also capable of providing traceability, interoperability, and intuitive editing features. This is the main problem we want to address.

With this in mind, we will be presenting our own solution in the next chapters.

Chapter 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 3.1: (1) the web application that supports the UI for creating programs; and (2) the remote CAD application, that offers an API for running programs in CAD applications. The first four tasks described above can be handled by the web application alone since there is no need to interact with CAD applications running on the user's computer. The fifth task requires both the web application and the remote CAD service. As also seen in Figure 3.1, apart from using the web application programming environment UI, the architect also needs to install the remote CAD application if and when he wants to export results to CAD applications.

We made a test implementation of this architecture which we will call Luna Moth from here on.

In the rest of the chapter, we describe these two components and how each of the tasks is implemented in Luna Moth. Section 3.2 describes the web application and its tasks. Section 3.3 describes the remote CAD service and the export process.

3.2 Web Application

To handle the tasks of letting the architect develop programs, running programs, supporting program understanding, and displaying results, the UI of the web application has the layout shown in Figure 3.2. 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).

To help with the implementation, we have used the THREE.js library[1] to interact with WebGL[12] and the Ace Editor library,[2] to provide a syntax highlighted text editor.

[1] https://threejs.org/ (last accessed on 10/05/2017)
[2] https://ace.c9.io (last accessed on 10/05/2017)

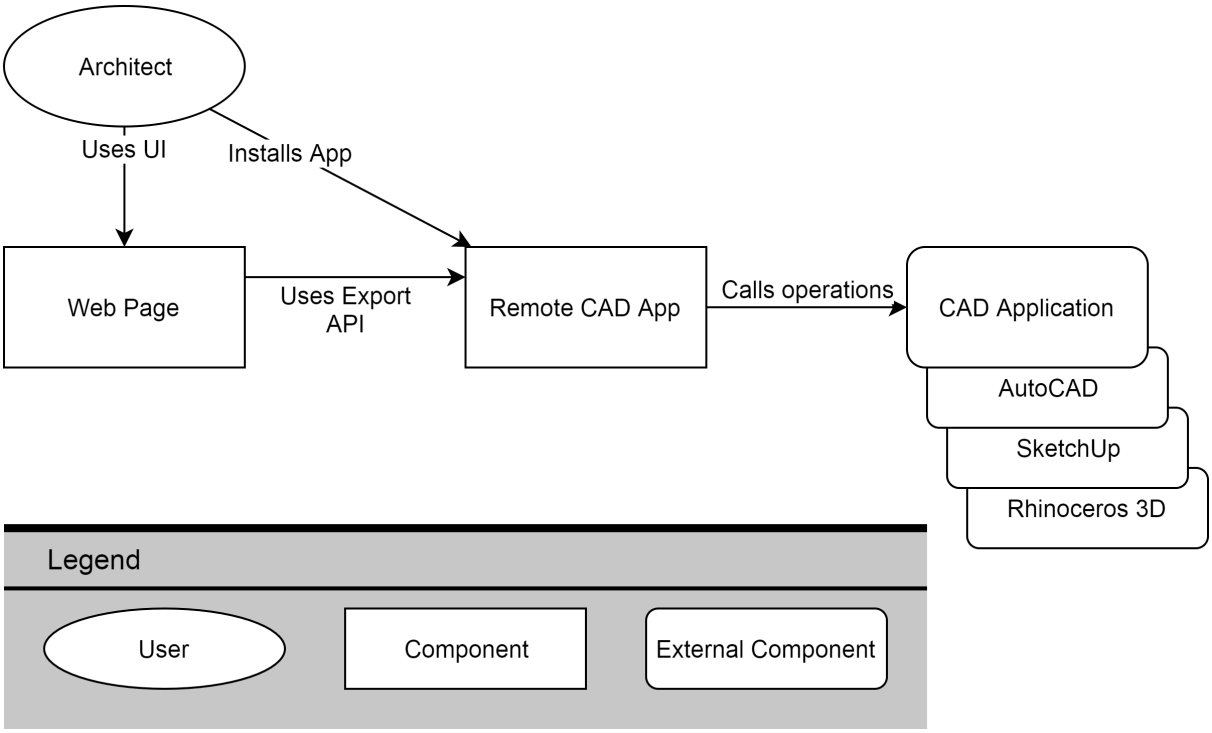


Figure 3.1: Architecture of the solution.

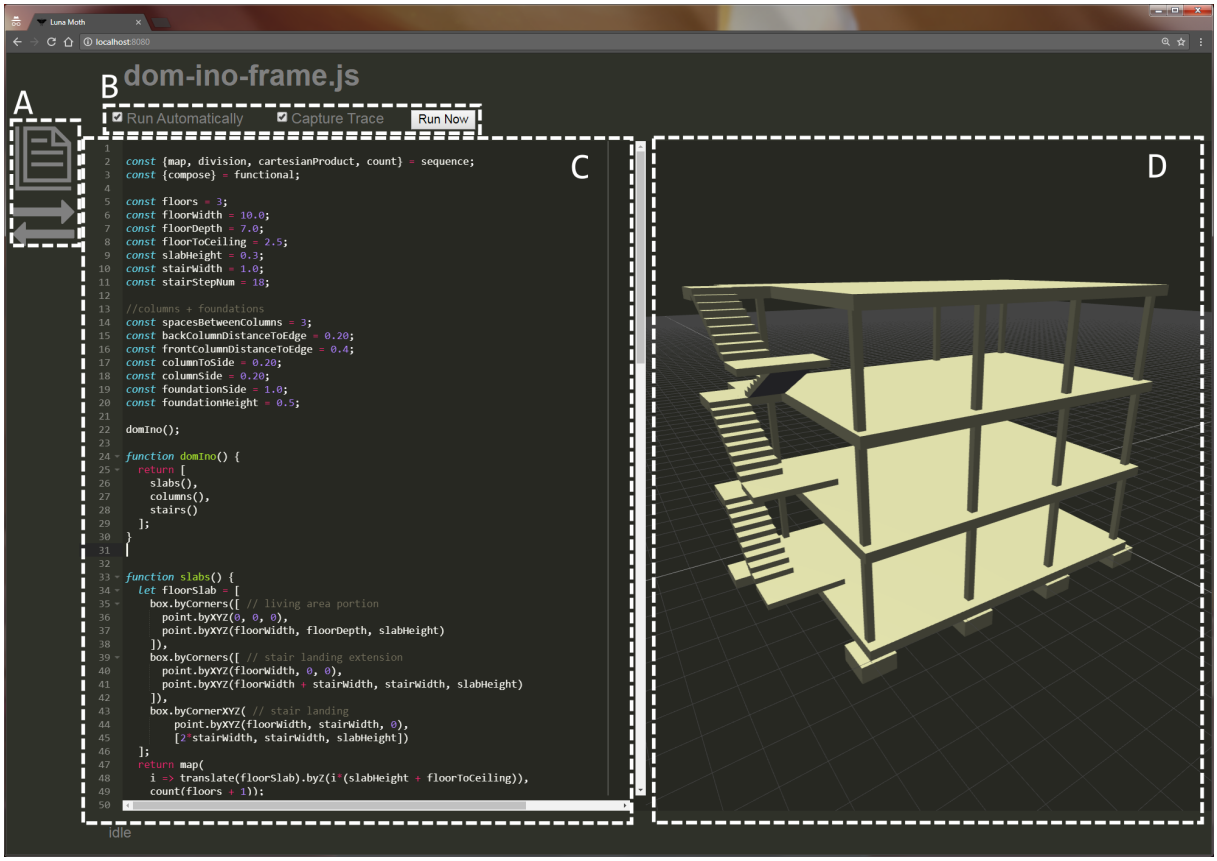


Figure 3.2: Layout of the UI of the web application.

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[23]. Luna Moth enables him to do this by providing immediate feedback to changes and by showing traceability.

3.2.1.1 Immediate Feedback

One of the ways to understand the program is to understand the relationship between its inputs and outputs, that is, understanding how each input affects the generated model. To do this, the architect needs to change the inputs, run the program, see the results, and repeat until he understands the relationship. This is a tedious process that will bore most architects. Luna Moth can help them by making this process more immediate. It must provide quick ways of changing inputs and it must rerun the program when inputs change, so the architect can see the effects of his changes immediately. This immediacy allows the architect to quickly understand the relationship, which means he is ready to start exploring the design sooner. Afterward, he can use the same immediacy to explore the design more efficiently. To this end, we developed and implemented two different features, namely, the ability to run programs while the architect is writing them, and the ability to efficiently adjust literals.

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 a literal. When experimenting with a GD program, architects find themselves repeatedly adjusting literals that were hardcoded into the program. This is usually done by retyping parts of the literal's text and, then, re-running the program to see the effect. Unfortunately, this is annoying and, worse, it makes it difficult to fine-tune the literal. One possible way to improve this behavior is to automatically re-run the program on every change. This has the advantage of allowing immediate feedback to changes in literals.

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 errors get amplified when the programming environment provides real time feedback and, like so, begins rerunning the program before the error is corrected, leading to annoyance and to reduced UI responsiveness.

There are several ways we can extend the user interface to make adjusting literals more user friendly that do not involve manually replacing digits. They have a simpler mapping to the user's intent. The following list describes some of them:

- **Virtual joystick** Clicking on a literal will display a virtual joystick close to it. When clicked and dragged, it changes the value repeatedly over time, being faster or slower depending on how much the handle is moved from its center.
- **Click and drag** Clicking and dragging on a literal will change it according to how much the pointer moved from the starting position.

- **Sliders** Clicking on a literal will display a slider. Dragging the slider's handle will change the literal depending on how much the handle is moved from the center of the slider. The slider can have different scales, linear or non-linear, to map the center-handle distance to the amount of change for the literal.
- **Keyboard shortcuts** Pressing key combinations while having a literal selected, or the text editor's caret on it, will increment or decrement the literal.

Each of the above has its advantages and disadvantages. The *Virtual joystick* and the *Sliders*, for example, have the advantage that they are visible to the user and, like so, are easier for him to see and understand. However, by being visible, they can also block parts of the program that he may want to see. The *Keyboard shortcuts* have the advantage that they do not introduce any visual clutter into the program editor, are more precise, and that they are accessible without getting the hands out of the keyboard. However, they do require the user to learn their key combinations, which is often avoided by new users. Lastly, the *Click and drag* also has the advantage of not introducing visual clutter into the editor. Moreover, as with *Virtual joystick* and *Sliders*, since the interaction is made using the mouse, it is more intuitive for new users. However, less visual clutter means that the user may not realize that it is possible to adjust the literal.

Taking the advantages and disadvantages of each alternative into account, we decided to implement the *Click and drag* way of adjusting literals since it is a good compromise between being easy to use for new users and not introducing visual clutter to the editor. Figure 3.3 shows an example of its usage.

Implementation To implement this behavior, we need to focus on the program editor, as this is where the relevant information is.

The behavior starts when the user presses a mouse button while the pointer is over the text editor. If it is indeed hovering a numerical literal, we setup an event listener that will update the literal every time the pointer moves until the mouse button is released.

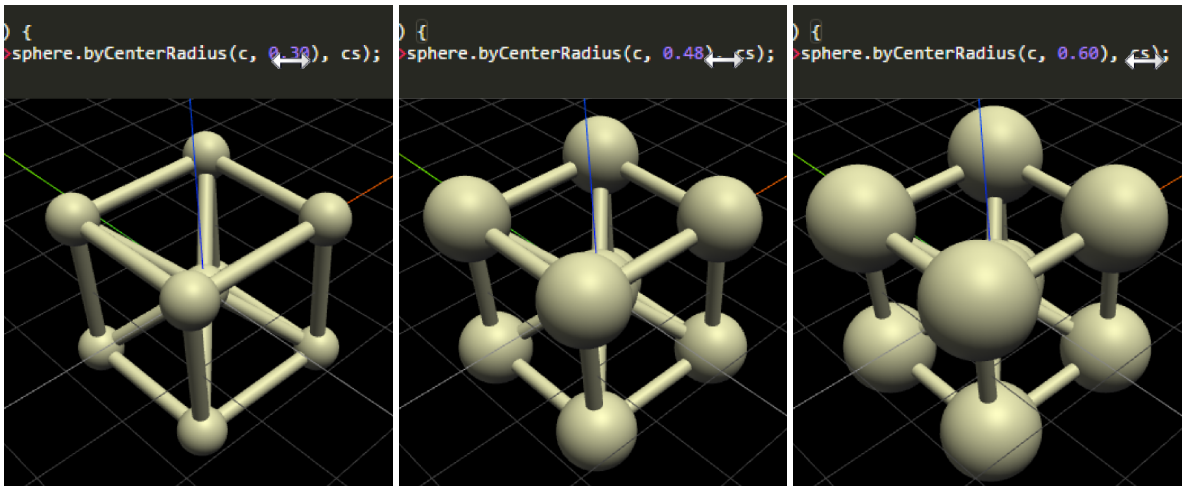
To check that the pointer is over a numerical node, we use the pointer position and the Abstract Syntax Tree (AST) of the current program. Since the program will change during the behavior, we keep the path taken through the AST to the numerical literal. We also keep the starting pointer position as a reference for calculating the new literal value when the mouse is moved.

We update the literal by changing the portion of the source code that it represents to the digits representing the new value. The new value has the same number of fractional digits as the original literal. To get the new value, we first extract the number of fractional digits and an integer containing the literal's digits and sign (ignoring the decimal point) from the literal's source text. Afterward, we add the horizontal distance between the starting point and the current pointer position to the number. Finally, we convert it back to text and re-add the decimal point.

3.2.1.2 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. This helps the architect build an understanding of the program from the parts to the whole, and identify both causes of errors and parts that need to be changed to accommodate new design requirements. This is especially true when he is dealing with programs that generate complex models, where it is much harder to simply look at the program and results and mentally infer the relationship.



(a) Right after clicking. (b) After dragging a little. (c) After dragging a little more.

Figure 3.3: Example of literal adjustment.

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

There are several ways that the environment can use to graphically show the relationship between program and results. We will describe some of them next:

- **Highlighting expressions and results** Every object that is part of the results of a program was created by one expression of that program. On the other hand, every expression of the program creates one or more objects of the results. The environment can make these relationships visible by changing the appearance of expressions and results when the user points either at an expression or at an object.
- **Timetable** As described in Learnable Programming[24], making things visible makes them more real to the programmer. Timetables were proposed as a way to visualize the control flow of programs. They act as a map of the program's execution. By seeing the control flow, it is easier to see what the program does and it is easier to point at interesting locations. By allowing this, timetables provide better navigation inside a program's execution.
- **Display data** Also described in Learnable Programming[24], displaying data would increase the amount of traceability the system provides. Apart from being able to see the control flow and the 3D results, seeing the values (e.g. numbers, booleans) that variables and expressions had throughout the program would give the programmer a more complete view of the execution.

Each of these alternatives gives the user a different look into what happens as the program runs. This suggests that they could be used together to increase the environment's traceability. However, each of them requires additional data collection, which will inevitably increase the running time of programs. Like so, we have to choose the one that gives the best compromise between being helpful to the programmer and not slowing program execution too much.

When it comes to helping the programmer, the second and third alternatives offer the programmer a better overview of what the program did when it ran, while the first only shows the end-to-end relationship between code and results.

In terms of performance, it is clear that the first alternative will require less data collection since some expressions are guaranteed not to produce 3D results, like arithmetic expressions, while the other two

require every step of the program to be recorded.

With these two views in mind, there seems to be a tie between alternatives. Nonetheless, by taking into account that both *Timetable* and *Display data* require space in the UI, we decided that it was best to implement the *Highlighting expressions and results* alternative since its impact on the UI is smaller. Moreover, it can achieve better performance.

Implementation The implementation of traceability by highlighting expressions and results requires three things: (1) collecting traceability data while running a program; (2) detecting that the user is pointing at either an object from the results or an expression of the program; and (3) highlighting both the pointed object and the corresponding creator expression, or, in the opposite direction, both the pointed expression and the corresponding created results, as exemplified in Figure 3.4.

To get the traceability data it needs to show the program-results relationship, Luna Moth instruments the program. We will go into more detail on this matter in Subsection 3.2.2.

In regard to detecting what the user is pointing at, we follow two approaches depending on the pointer being over the program or over the 3D view. When the pointer is over the program, we use the program's AST annotated with source code locations and the pointer position to find the deepest expression that has recorded traceability data. When the pointer is over the 3D view, we use ray-casting to detect which of the result objects is below the pointer and closest to the camera.

After detecting that something is being pointed at, Luna Moth needs to highlight it and the corresponding part in the program editor or the 3D view. To do that, it first uses the traceability data to find the corresponding created objects for an expression, or the first expression that created the pointed object. Afterward, it highlights both. It highlights expressions by changing the program editor's background behind them to a different color. To highlight 3D objects, it changes their material to a transparent and emissive one, and overlays them on top of the remaining objects.

When working on a larger design, the user may find that the data collection needed to support traceability has a performance impact that he might not want to accept. In order to solve this problem, we made traceability data collection optional to the user. This way, he can decide to turn it on when he wants to debug his program or to turn it off when he wants his program to run faster.

3.2.2 Running Programs

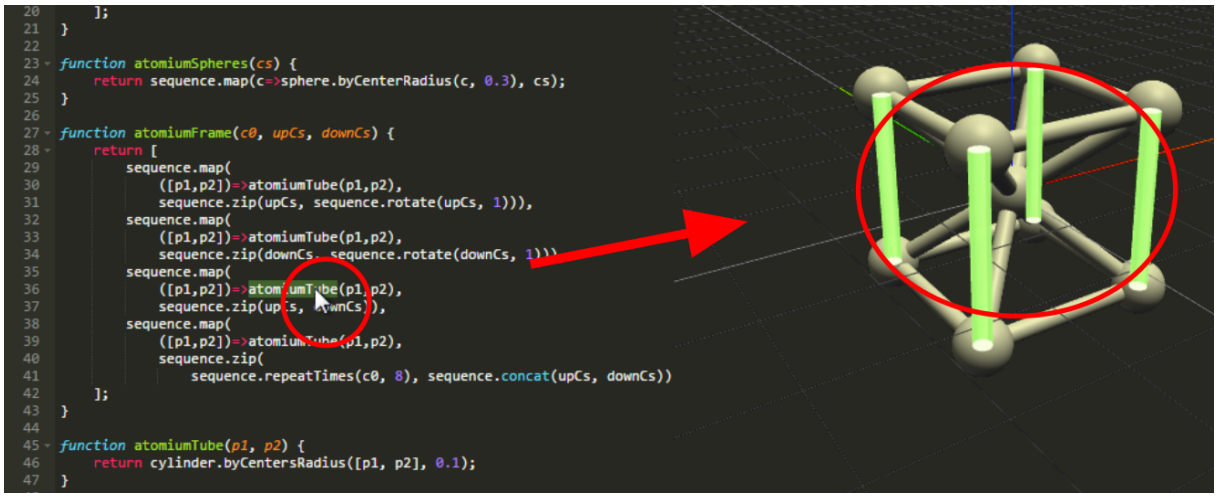
One of the fundamental parts of the programming environment is that it runs programs. To run programs, the environment needs to support a programming language. This includes the syntax and semantics of the language and the primitive operations that are available to programs. After knowing these, the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

3.2.2.1 Programming Language

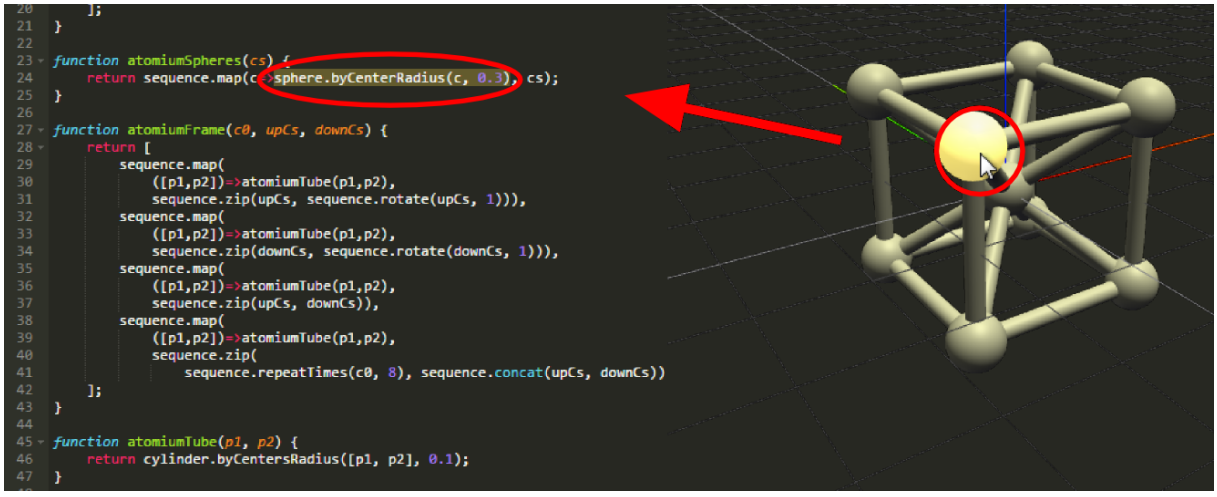
Everyone can write a text or draw a diagram of what they want the computer to do. However, if they do not use a concrete and unambiguous language that can be translated automatically by the computer, it will not be possible to get the computer to do what they want.

Luna Moth has to use a programming language that fits the architect's needs. It can either be an existing language with slight adaptations or a new one specially tailored for their needs.

Still, creating a new programming language is not a realistic option since it would require a great implementation effort to get a working version. Moreover, since we want programs to run as fast as



(a) From code to results.



(b) From results to code.

Figure 3.4: Two examples of the traceability mechanism. The first from program to results and the second from results to program.

possible to be able to give architects immediate feedback, further effort would be needed to make it perform fast.

Furthermore, existing programming languages have had time to gather communities that can help the architect when he does not know how to do something or when he is having difficulties correcting a bug. A new programming language would have a severely limited community. This suggests that we should use a programming language that already exists.

Architects' experience with programming can vary, however, we can assume that there are more novices than experts. Like so, Luna Moth needs to have a programming language that is easy to learn and use. We can consider programming languages that are commonly regarded as good introductory languages such as Python, Racket, and Processing as possible candidates. Additionally, we can also consider DesignScript, since it was made specifically with computational design in mind[25], and also JavaScript, since it is readily available in web browsers and was originally made for web designers.

Programming languages have to be translated into the host's language. For example, to run programs on a typical computer processor, they need to be translated into the `x86` instruction set; to run programs on a virtual machine like the JVM, they need to be translated to its bytecode. The same is true to run programs on a web browser. In this case, instead of translating programs into an instruction set or a bytecode, they need to be translated into JavaScript. Using JavaScript as Luna Moth's language would spare us from a translation step as it is already the target language. This lead us to choose JavaScript as the programming language supported by the environment.

After having a programming language, there still needs to be a concrete API that supports the creation of GD programs. There are things that are essential to support like the creation of geometry, such as coordinates, lines, surfaces, and solids. Without the API supporting these, the architect would need to implement these concepts even before tackling his design problem. Furthermore, since architects are not trained programmers, it should be easy to create programs that are easy to understand. Because of that, operations should remain mostly purely functional, i.e. without side-effects. This way, each operation does only one thing, without modifying the value of any variable, and as a result it is easy to compare its arguments and results. It behaves more like a mathematical expression.

With a programming language and an API, there is still one thing that needs to be decided for Luna Moth to be able to display results to the architect: how to collect results from programs?

We considered various alternatives:

- **Special entry point** Like in OpenJSCAD, we could require that every program must define a function with a special name (e.g. "main"). The environment would then call this function and use its return value as the results of the program.

- **Display function** Another way to get results from a program would be giving it access to an output function to the program. During its execution, the program would call this function, maybe several times, to produce its output.

- **Imperative primitives** A third way to get results would be to add a side-effect to all the 3D object producing functions/primitives. Apart from creating the object, they would also add it to the program's results. This would be similar to having a display function, as all primitives become specialized display functions.

- **Top-level expressions** A fourth way to get results would be to collect them from the expressions at top-level position within the program.

Each of this alternatives encourages a slightly different programming approach. The *Imperative primitives* and *Display function* alternatives encourage an imperative programming approach, since the

programmer must give instructions to create objects or display them. The *Special entry point* and *Top-level expressions* alternatives, on the other hand, encourage a functional programming approach since the programmer must specify what objects constitute the results.

We wanted the environment to encourage users to follow a functional programming approach. Therefore, we did not allow primitives to have side-effects and, like so, we have excluded the alternatives to get results that encouraged imperative programming, namely, *Imperative primitives* and *Display function*.

Having a *Special entry point* is similar to collecting the results of *Top-level expressions*, however, it requires the programmer to define a function that returns the results of the program. When the programmer wants his program to have several results, he will have to aggregate them into a value to provide it as a return value. This fact lead us to exclude using a *Special entry point* as the way Luna Moth uses to collect results.

With all this in mind, we chose to collect the results of *Top-level expressions* as the way to collect program results. Like this, it is easy for the programmer to specify the results of programs since he only has to write the expression at the program's top-level. Moreover, this approach still encourages him to follow a functional programming approach. Finally, Listing 3.1 gives an example of what a program looks like in Luna Moth.

```
const xyz = point.byXYZ;

function abacus(height, side) {
  return box.byBottomWidthHeightZ(xyz(0,0,0), [side, side], height);
}

function echinus(height, side, neckSide) {
  return coneFrustum.byBottomTopRadiusesHeight(xyz(0,0,0), neckSide/2, side/2, height);
}

function shaft(height, side, neckSide) {
  return coneFrustum.byBottomTopRadiusesHeight(xyz(0,0,0), side/2, neckSide/2, height);
}

function column(height, side, neckSide) {
  return [
    translate(abacus(height*0.05, side)).byZ(height*0.95),
    translate(echinus(height*0.05, side, neckSide)).byZ(height*0.90),
    shaft(height*0.90, side, neckSide)
  ];
}

column(6.00, 0.8, 0.7);
```

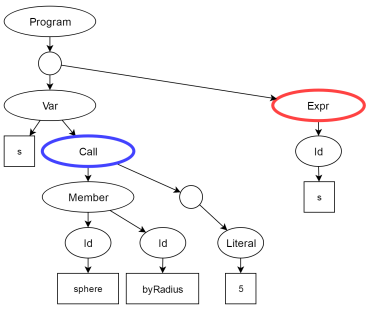
Listing 3.1: An example of a program written in Luna Moth.

3.2.2.2 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 data. It does these two by instrumenting the program with additional calls to special functions that record the desired information, then running the program, and afterward collecting the results from the instrumentation. As stated in 3.2.1.2, the collection of traceability data is optional since it adds some overhead to program execution, which may be undesired when programs start to take too much time to run.

Before

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



After

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

Figure 3.5: A program before and after the transformation. Its AST is also shown, with the function call highlighted in blue and the top-level expression highlighted in red.

To be able to perform the instrumentation, Luna Moth parses the program using Esprima, a JavaScript parser,[3] to get its AST.[4]

Afterward, the AST is transformed by adding a recording call to (1) all top-level expression statements and (2) all function call expressions. Figure 3.5 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, afterward, the recorded information is made available to the rest of the system.

3.2.2.3 Displaying Results

After running the program, Luna Moth needs to display a rendering of the results to the architect. Displaying results is deeply tied to what programs can produce since their results must be converted to a representation that can be rendered. The API used by those programs is what determines which results can be produced. Luna Moth uses THREE.js for 3D rendering. By using THREE.js, the implementation effort is reduced since, on top of implementing rendering, THREE.js also implements several geometric primitives useful for GD and implements ray-casting. However, since Luna Moth provides a functional API and THREE.js's API is imperative, it needs to cover the differences between these two APIs.

The main difference between them lies in how they support reuse. THREE.js's API makes it difficult to reuse objects since it only allows scenegraph objects to have one parent. If the programmer wants to reuse one object, he has to clone it. Luna Moth's API, on the other hand, does not impose this rule. To cover the difference, we use two steps: (1) we run the program using the functional API, creating intermediary results; and (2) we convert those results to objects from THREE.js's API, cloning them when necessary.

Consider the program in Listing 3.2, which creates spheres on the vertices of a cube of width `w`. Figure 3.6 shows the overall process from having a program to displaying its results. After running the program, results are in the form shown in 3.7a, where the variable `g` holds several translations (`+xyz`) of the same sphere (`sph`). The next stage converts these results to a THREE.js scenegraph, creating a mesh for `sph`, then copying it for each translation, resulting in the "`sph : Mesh`" objects seen in 3.7b. Afterward, these object are grouped as children of the "`g : Object3D`" object. After having results converted to THREE.js objects, Luna Moth issues their rendering repeatably as the architect interacts with the 3D view.

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

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

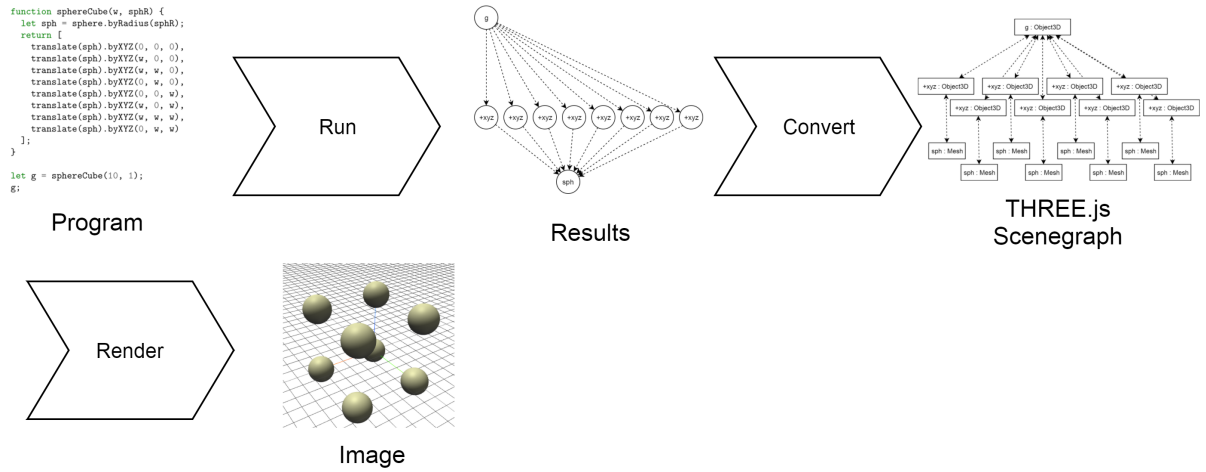


Figure 3.6: The process used to display results.

Luna Moth also keeps record of the correspondence between intermediary results and THREE.js objects. Together with the correspondence between program nodes and intermediary results, this information is used to enable support for showing traceability.

```
function sphereCube(w, sphR) {
  let sph = sphere.byRadius(sphR);
  return [
    translate(sph).byXYZ(0, 0, 0),
    translate(sph).byXYZ(w, 0, 0),
    translate(sph).byXYZ(w, w, 0),
    translate(sph).byXYZ(0, w, 0),
    translate(sph).byXYZ(0, 0, w),
    translate(sph).byXYZ(w, 0, w),
    translate(sph).byXYZ(w, w, w),
    translate(sph).byXYZ(0, w, w)
  ];
}

let g = sphereCube(10, 1);
g;
```

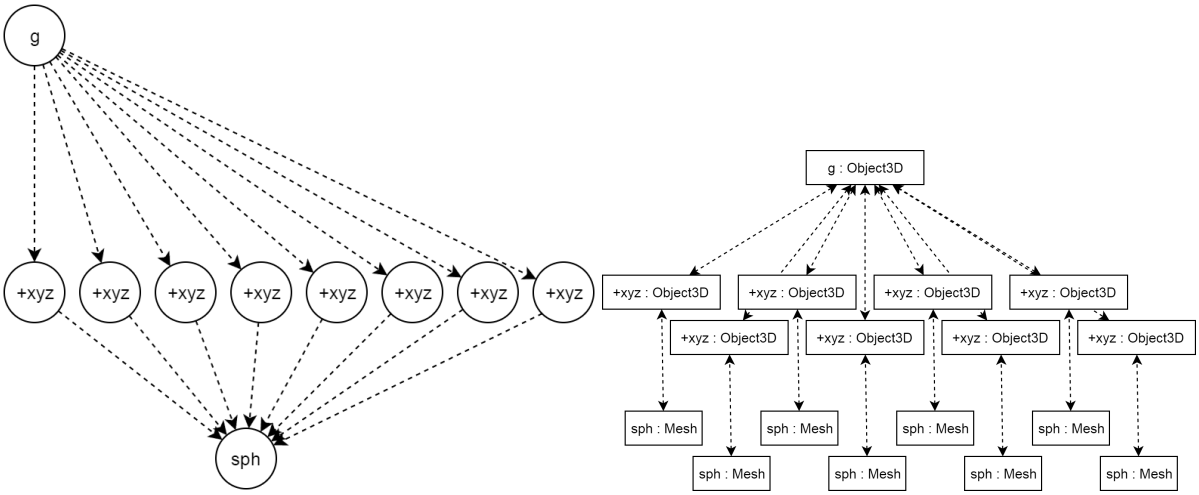
Listing 3.2: A program creates spheres on the vertices of a cube.

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. To do so, it must locate the CAD application and connect to it. In addition, there must be a well-defined interface of operations that can be performed in the CAD, which needs to cover the operations that can be performed in programs, as well as operations for managing the CAD's document.

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 serves as



(a) Program results (b) THREE scenegraph

Figure 3.7: Comparison of program results structure and THREE.js scenegraph. Arrows indicate that the object at their root has a reference to the object they point to.

a bridge between CAD applications installed in the computer and the environment running on the web page. 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. The architecture for this part of Luna Moth is illustrated in Figure 3.8.

Figure 3.9 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, afterward, the model resulting from running the program is available in both CAD applications.

Note that it is only needed to install the *remote CAD app* if and when the architect wants to get the program's results on a CAD application. Furthermore, the application only needs to be installed in the computer that actually has the CAD application installed.

3.3.1 Implementation

We implemented the *remote CAD app* as a small Racket server. It expects to receive HTTP requests for operations like adding shapes to the currently selected CAD applications or deleting all shapes they may have. It also expects requests for specifying what CAD applications are selected.

Apart from the connection between the remote CAD application and the web application, there also needs to be a connection between the remote CAD application and the CAD applications themselves. Different CADs have different APIs so the application needs to know how to interact with each one.

Instead of implementing the connection with CAD applications, the *remote CAD app* uses Rosetta as an intermediate. This way, it only has to cover the semantic differences between Rosetta's API and the remote CAD service's API.

In reality, when the architect wants to export to CAD, it is likely that he is using the web environment in the same computer where he has the CAD application installed. Like so, the web page, the remote CAD application and the CAD applications are running on the same computer. With this in mind, we chose to temporarily host the web application containing the IDE in the *remote CAD app*'s server. Using this setup, it is easier to test the communication between the page and the application. Moreover, this setup

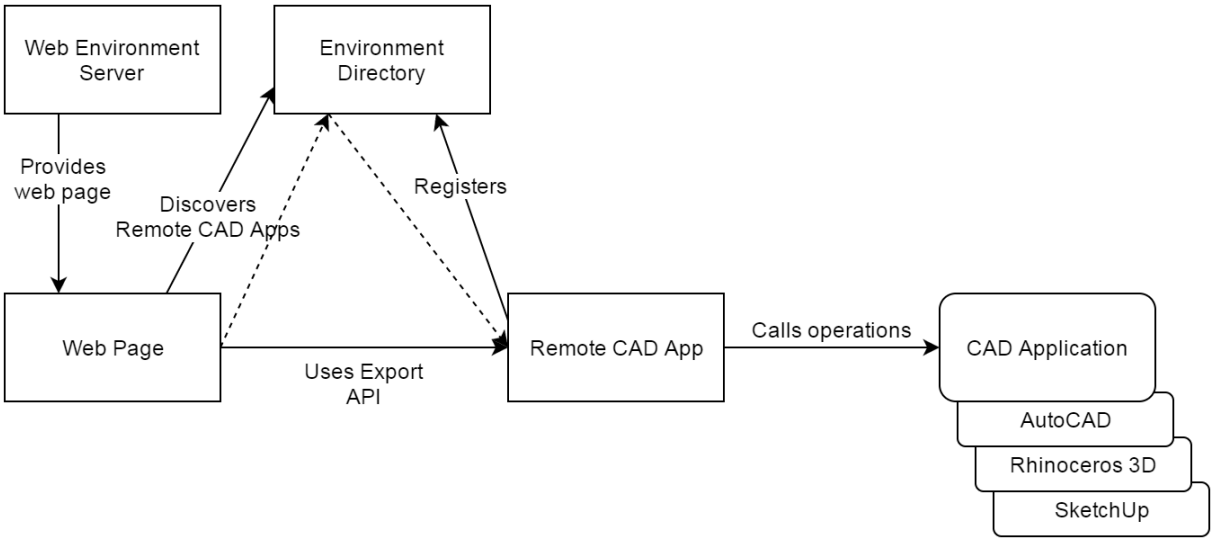


Figure 3.8: Illustration of the remote CAD service's architecture.

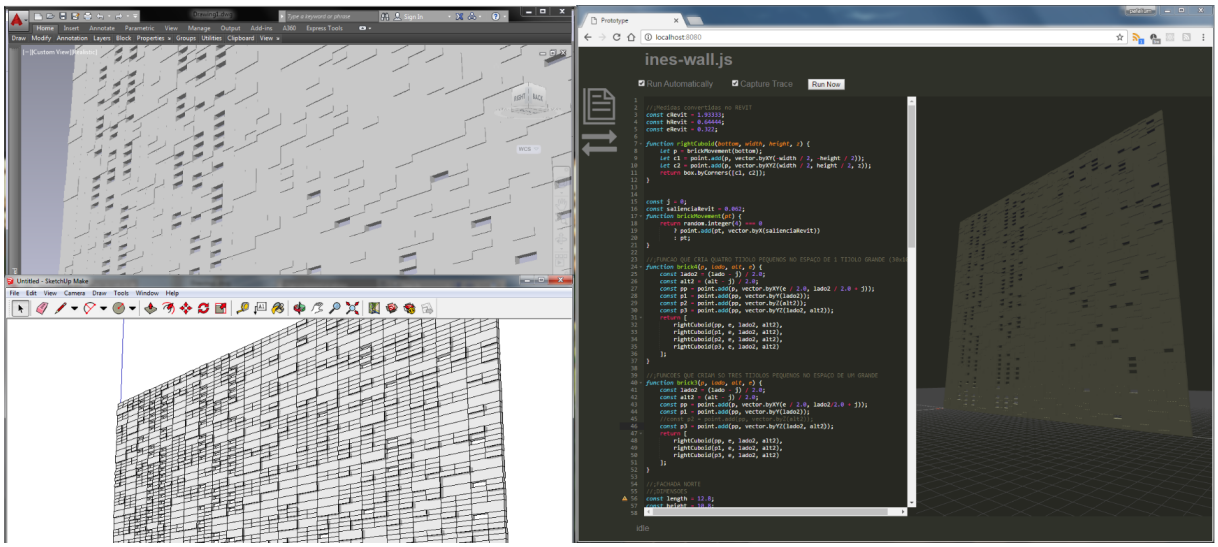


Figure 3.9: An example of the remote CAD service. The results of the program have been passed to both AutoCAD and SketchUp.

also avoids dealing with cross-origin restrictions imposed by web browsers for Asynchronous JavaScript and XML (AJAX) requests.

Chapter 4

Evaluation

The goal of our work was to bring GD to the web browser so it can be deployed and used anywhere easily. In this chapter, we assess how our solution accomplishes this goal. We start by looking at examples that were implemented in Luna Moth in Section 4.1. Afterwards, we measure Luna Moth's performance and compare it to other environments in Section 4.2.

4.1 GD Capability / Usage Examples

A programming environment needs to allow the creation of a wide variety of programs. We need to test its ability to be used for GD, or, more specifically, to produce diverse 3D models. To show this variety, we present several examples that make use of some primitives that were implemented and also some programming techniques that they use.

4.1.1 Example: Atomium

The Atomium, in Brussels, was built between 1956 and 1958 for Expo 58 (Figure 4.1). It consists of nine metallic spheres connected by sixteen tubes forming its shape, mimicking the arrangement of an iron crystal structure.

To create it in Luna Moth, we can start by deciding which primitives to use for the elements of the Atomium, that is the metallic spheres and the tubes. The spheres can be created using the `sphere` primitive.

```
function atomiumSpheres(cs, r) {
  return map(c => sphere.byCenterRadius(c, r), cs);
}
```

The tubes can be created using the `cylinder` primitive. They are placed at the edges of the cube defined by the outside spheres and also at the lines connecting them to the inside sphere. Sequence generation and manipulation functions – like `zip`, `rotateLeft`, `repeatTimes`, and `concat` – are used to create all pairs of points describing cylinder base centers.

```
function atomiumTube(p1, p2, r) {
  return cylinder.byCentersRadius([p1, p2], r);
}

function atomiumTubes(c0, upCs, downCs, r) {
```

Figure 4.1: The Atomium, Heysel, Belgium. Photo by Mike Cattel (https://www.flickr.com/photos/mikecattell/4042147060).

```
  return [
    map(([p1,p2]) => atomiumTube(p1, p2, r),
      zip(upCs, rotateLeft(upCs, 1))),
    map(([p1,p2]) => atomiumTube(p1, p2, r),
      zip(downCs, rotateLeft(downCs, 1))),
    map(([p1,p2]) => atomiumTube(p1, p2, r),
      zip(upCs, downCs)),
    map(([p1,p2]) => atomiumTube(p1, p2, r),
      zip(repeatTimes(c0, 8), concat(upCs, downCs)))
  ];
}
```

Now, the coordinates of each sphere center must be defined and sent to `atomiumSpheres` and `atomiumTubes`.

```
function atomiumFrame(sphereR, frameW, tubeR) {
  let c0 = xyz(0,0,0),
    c1 = xyz(-frameW, -frameW, +frameW),
    c2 = xyz(+frameW, -frameW, +frameW),
    c3 = xyz(+frameW, +frameW, +frameW),
    c4 = xyz(-frameW, +frameW, +frameW),
    c5 = xyz(-frameW, -frameW, -frameW),
    c6 = xyz(+frameW, -frameW, -frameW),
    c7 = xyz(+frameW, +frameW, -frameW),
    c8 = xyz(-frameW, +frameW, -frameW);
  return [
    atomiumSpheres([c0,c1,c2,c3,c4,c5,c6,c7,c8], sphereR),
    atomiumTubes(c0, [c1,c2,c3,c4], [c5,c6,c7,c8], tubeR)
  ];
}
```

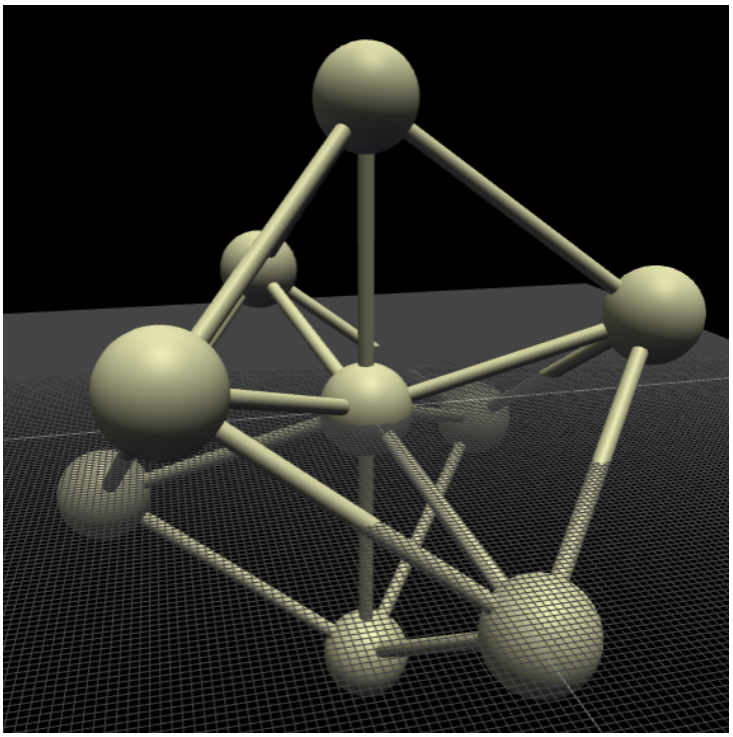


Figure 4.2: Atomium.

After creating the structure, we rotate it, aligning its z axis with an axis passing through $(0, 0, 0)$ and $(1, 1, 1)$. Figure 4.2 shows the resulting model.

```
function atomium(sphereR, frameW, tubeR) {
  return rotate(atomiumFrame(sphereR, frameW, tubeR))
    .aligningAxes(axis.z, axis.xyz);
}
```

4.1.2 Example: Trusses

Trusses are a common element in architecture that can adapt to different surfaces while maintaining strong structural properties. Trusses often consist of straight members connected together at their extremities, or joints. They can be planar, meaning that their members all lie on the same plane, or spatial, where members are distributed in three dimensional space. Spatial trusses can be represented by the mesh formed by their members. The following program implements spatial trusses with the same topology of the truss from Figure 4.3:

```
function trussKnots(pts, radius) {
  return map((pt) => sphere.byCenterRadius(pt, radius), pts);
}

function trussBars(ps, qs, radius) {
  return map(([p, q]) => cylinder.byCentersRadius([p, q], radius), zip(ps, qs));
}

function spatialTruss(curves, knotRadius, barRadius) {
```

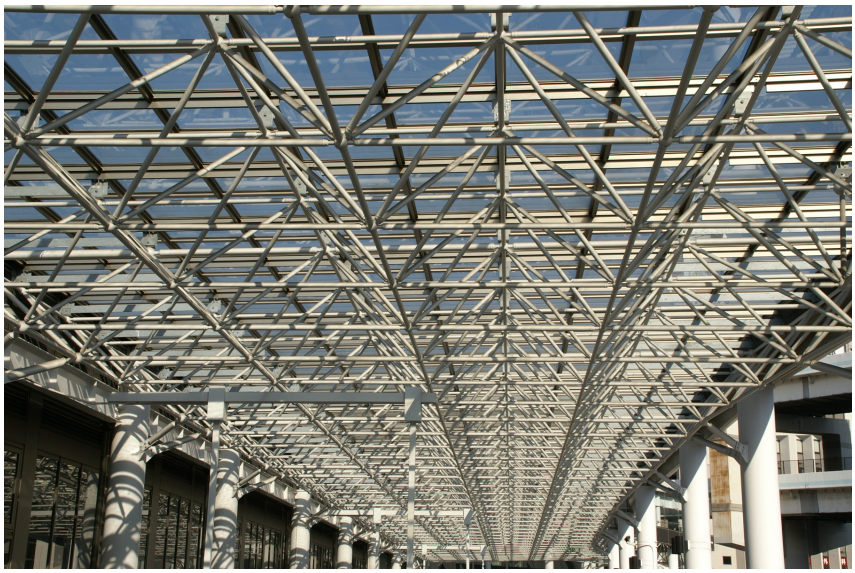


Figure 4.3: A spatial truss supporting a glass roof. Photo by ume-y (https://www.flickr.com/photos/ume-y/311887698).

```
  let as = curves[0];
  let bs = curves[1];
  let cs = curves[2];
  return [
    trussKnots(as, knotRadius),
    trussKnots(bs, knotRadius),
    trussBars(as, cs, barRadius),
    trussBars(bs, dropRight(as, 1), barRadius),
    trussBars(bs, dropRight(cs, 1), barRadius),
    trussBars(bs, drop(as, 1), barRadius),
    trussBars(bs, drop(cs, 1), barRadius),
    trussBars(drop(as, 1), dropRight(as, 1), barRadius),
    trussBars(drop(bs, 1), dropRight(bs, 1), barRadius),
    curves.length === 3 ?
      [
        trussKnots(cs, knotRadius),
        trussBars(drop(cs, 1), dropRight(cs, 1), barRadius)
      ]
      : [
        trussBars(bs, curves[3], barRadius),
        spatialTruss(drop(curves, 2), knotRadius, barRadius)
      ]
  ];
}
```

The `spatialTruss` function creates a space frame for any set of joint coordinates. Like so, if we wanted to produce a truss following an arc, we could define it with the following functions:

```
function arcCs(p, r, phi, psi0, psi1, n) {
  return map(psi => add(p, spherical(r, phi, psi)),
```

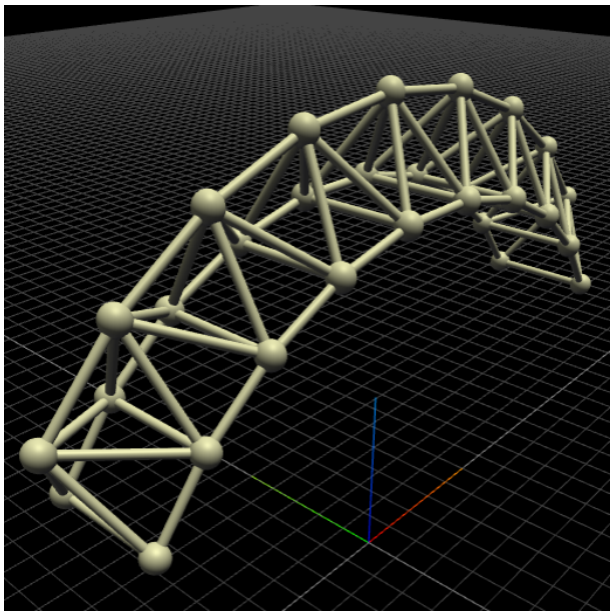


Figure 4.4: An arc-shaped truss.

```
    intervalDivision(psi0, psi1, n));
}

function arcTrussCs(p, apexR, baseR, phi, psi0, psi1, e, n) {
  let dpsi = (psi1 - psi0)/n;
  return [
    arcCs(add(p, polar(e/2, phi + Math.PI/2)),
      apexR, phi, psi0, psi1, n),//base arc
    arcCs(p, baseR, phi,
      psi0 + dpsi/2, psi1 - dpsi/2, n-1),//apex arc
    arcCs(add(p, polar(e/2, phi - Math.PI/2)),
      apexR, phi, psi0, psi1, n)//base arc
  ];
}

function arcTruss(p, apexR, baseR, phi, psi0, psi1, e, n) {
  return spatialTruss(
    arcTrussCs(p, apexR, baseR, phi, psi0, psi1, e, n),
    0.3, 0.1);
}
```

The `arcCs` function creates a given number of points in an arc. Function `arcTrussCs` then uses this function create to the three arcs that include all the coordinates of joints of the truss. It is then used in `arcTruss` in conjuntion with `spatialTruss` to actually create the arched space frame (Figure 4.4).

It is common that planar space frames are required to follow a surface. Unfortunately, `spatialTruss` requires not only points that lie on the surface but also points that are either above or below it to create the trusses's quadrangular pyramids. `insertPyramidApexCurves` is an helper function that takes a grid of points lying on a surface and creates a set of joint coordinates ready for use in `spatialTruss` by adding pyramid apexes for each quad defined by the grid. `spatialTrussInsertApex` combines

```
function wavyCs(p, baseR, l, phi,
  psi0, psi1, psiN,
  alpha0, alpha1, alphaN, rAmpl) {
  return map(
    ([i, alpha]) => arcCs(
      add(p, polar(i*l, phi + Math.PI/2)),
      baseR + rAmpl*Math.sin(alpha), phi,
      psi0, psi1, psiN
    ),
    zip(count(alphaN),
      intervalDivision(
        alpha0, alpha1, alphaN))
  );
}

function wavyTruss(p, baseR, l, phi,
  psi0, psi1, psiN,
  alpha0, alpha1, alphaN, rAmpl) {
  return spatialTrussInsertApex(
    wavyCs(p, baseR, l, phi,
      psi0, psi1, psiN,
      alpha0, alpha1, alphaN, rAmpl));
}
```

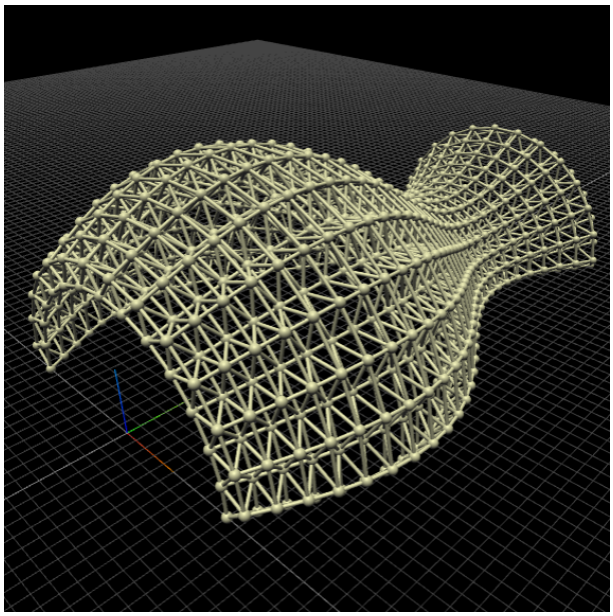


Figure 4.5: A wavy truss.

`insertPyramidApexCurves` and `spatialTruss` into one easy to use function.

```
function insertPyramidApexCurves(curves) {
  let css1 = curves,
    css2 = drop(curves, 1);
  let apexess = map(
    ([cs1, cs2]) => insertPyramidApex2Curves(cs1, cs2),
    zip(css1, css2));
  return interleave(css1, apexess);
}

function spatialTrussInsertApex(cs) {
  let c1 = (cs[0])[0];
  let c2 = (cs[1])[0];
  let c4 = (cs[0])[1];
  let d = Math.min(distance(c1, c2), distance(c1, c4));
  let knotRadius = d/5;
  let barRadius = d/15;
  return spatialTruss(
    insertPyramidApexCurves(cs), knotRadius, barRadius);
}
```

With this function, it is possible to create trusses on any surface. For example, on a wavy surface, as seen in Figure 4.5, or on a surface following the Moebius band, as seen in Figure 4.6.

```
function moebiusCs(r, u1, u2, m,
  v1, v2, n) {
  return enumerateMN(function(u, v) {
    return cylindrical(
      r*(1 + (v*Math.cos(u/2))),//radius
      u,//angle
      r*v*Math.sin(u/2));//z
  }, u1, u2, m, v1, v2, n);
}

function moebiusTruss(r, u1, u2, m,
  v1, v2, n) {
  return spatialTrussInsertApex(
    moebiusCs(r, u1, u2, m, v1, v2, n));
}
```

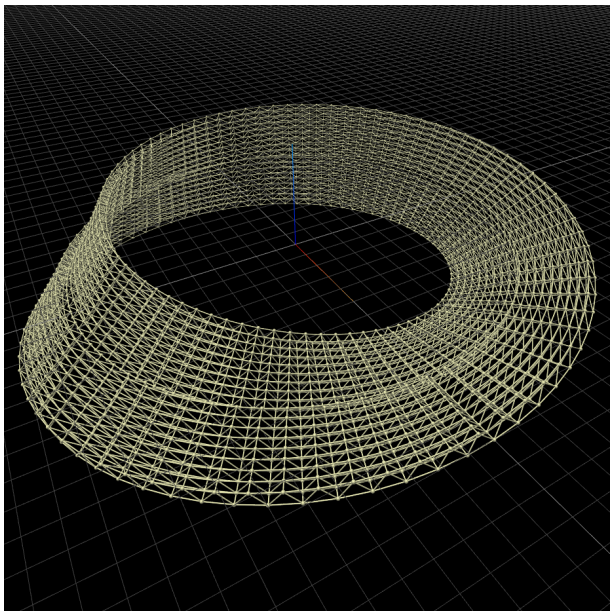


Figure 4.6: A truss on the moebius band.

4.1.3 Example: Randomness

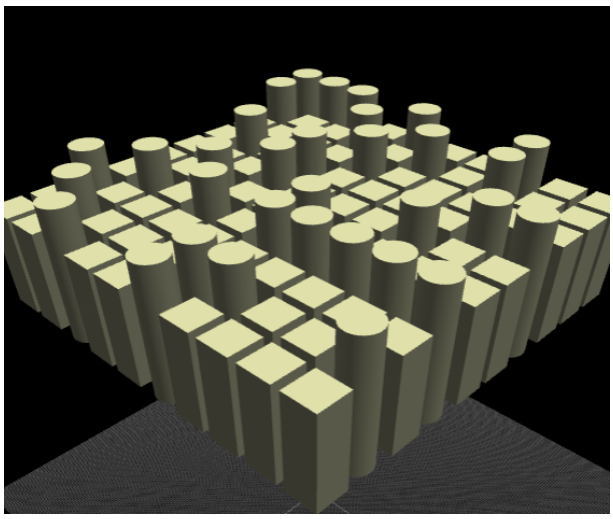
Computer programs allow highly repetitive designs to be generated quickly. However, architects may want to introduce some variation to make designs look more natural. They can do this by introducing randomness into parts of their designs. Luna Moth supports this by providing functions that generate random numbers.

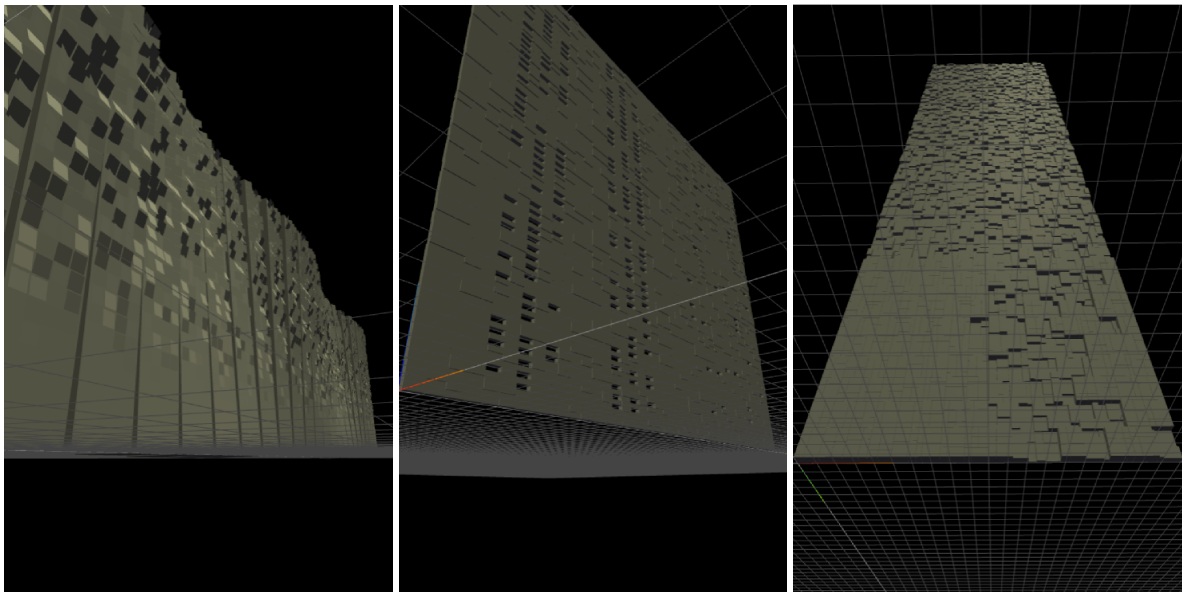
Randomness can be used to randomize the control-flow and other discrete aspects of the program. Suppose that we want to create a simplistic city where approximately one third of buildings are cylindrical while the rest are parallelepipedic. Consider the program:

```
function coordinateGrid(p, uu, vv, m, n) {
  return map(
    ([u, v]) => add(p,
      add(scale(uu, u), scale(vv, v))),
    cartesianProduct(count(m), count(n))
  );
}

function building(p) {
  return random.real(3) < 1
    ? cylinder.byCentersRadius(
      [p, add(p, z(75))], 10)
    : box.byBottomWidthHeightZ(
      p, [20, 20], 60);
}

function city(p) {
  return map(building,
    coordinateGrid(p, x(25), y(25),
      10, 10));
}
```





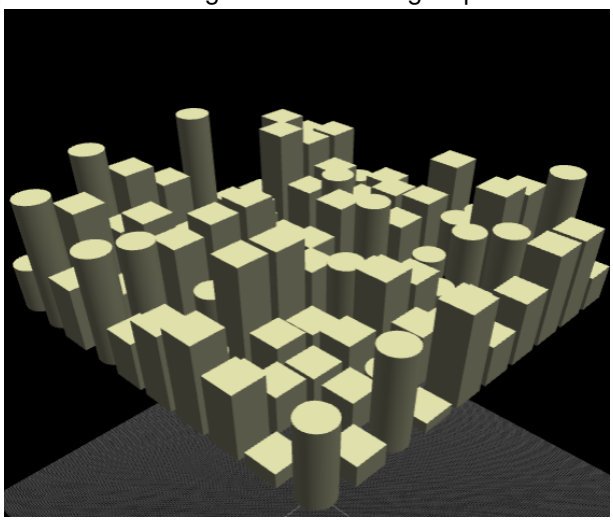
(a) Nolan Facade (b) Ines Wall (c) Sheung-Wan Hotel

Figure 4.7: Results of programs using randomness.

The function `building` uses `random.real` to decide whether to create a cylinder or a box. Calling `city` with a point creates a grid of boxes and cylinders where approximately one third are cylinders.

Random numbers can also be used to control continuous quantities like heights or distances. For example, we can modify function `building` to also pick a random height for the buildings it produces.

```
function building(p) {
  return random.real(3) < 1
    ? cylinder.byCentersRadius(
      [p, add(p,
        z(random.real.inRange(10, 75)))],
      10)
    : box.byBottomWidthHeightZ(p, [20, 20],
      random.real.inRange(5, 60));
}
```



The examples from Figure 4.7a, 4.7b and 4.7c also make use of random numbers. The first example uses them to vary the rotation of the square elements of the facade. The second uses random numbers to decide which areas should have more or less holes and to determine the displacement of bricks from the face of the wall. Lastly, the third uses random numbers to determine thicknesses of blocks and where to subdivide them, to decide whether to continue subdividing, and to decide which blocks from the smooth part (the lower part) should look like the blocks from the rough part (the upper part).

4.1.4 Example: Higher-order Functions

Higher-order functions are functions that receive functions as parameters and/or return functions as their result. One benefit that comes from higher-order functions is allowing their behavior to change without

requiring them to be modified.

Coming back to the example of creating a simplistic city, the implementation that was presented earlier was limited in terms of the creation of buildings. It required the `building` function to be modified in order to allow the city to have different kinds of buildings. It would be preferable to turn `city` into an higher-order function receiving functions responsible for creating different kinds of buildings. One possible higher-order version of `city` can be as follows:

```
function city(p, buildingFns) {
  return map(pt =>
    buildingFns[random.integer(buildingFns.length - 1)](pt),
    coordinateGrid(p, x(25), y(25), 10, 10));
}
```

With this modification, it becomes possible to create variations of the city by simply using a different set of building creation functions. Like so, we can extract the two buildings from `building` into separate functions. In addition, we can also implement other kinds of buildings, like the cone shaped buildings created by `coneBuilding`.

```
function cylBuilding(p) {
  return cylinder.byCentersRadius(
    [p, add(p, z(random.real.inRange(10, 75)))],
    10);
}

function boxBuilding(p) {
  return box.byBottomWidthHeightZ(p, [20, 20],
    random.real.inRange(5, 60));
}

function coneBuilding(p) {
  return coneFrustum.byBottomTopRadiusesHeight(
    p, 10, random.real.inRange(0, 8),
    random.real.inRange(5, 60)
  );
}
```

Now we can call `city` with these building creation functions, as seen in Figure 4.8.

4.1.5 Completeness

Apart from the previous examples, some additional programs were made using Luna Moth. Figure 4.9 shows their results. As shown, the environment can produce interesting results with varying degrees of complexity.

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.

```
city(xyz(0, 0, 0),
  [cylBuilding, boxBuilding,
    coneBuilding]);
```

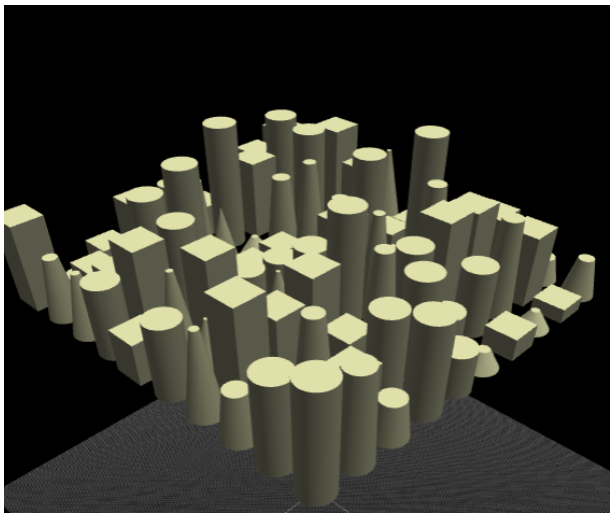


Figure 4.8: Making a city of boxes, cylinders and cone frustums.

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.

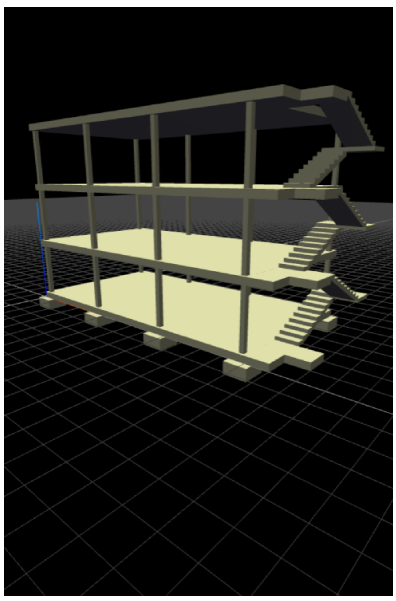
The web application and the remote CAD service were hosted at the same computer.

Times are the average of four runs. As Racket and JavaScript's runtime environments include Just-In-Time (JIT) compilation, we perform three warm-up runs for all IDEs before we actually start measuring running times. This way, they have the opportunity to perform optimization steps, thus, potentially improving running times.

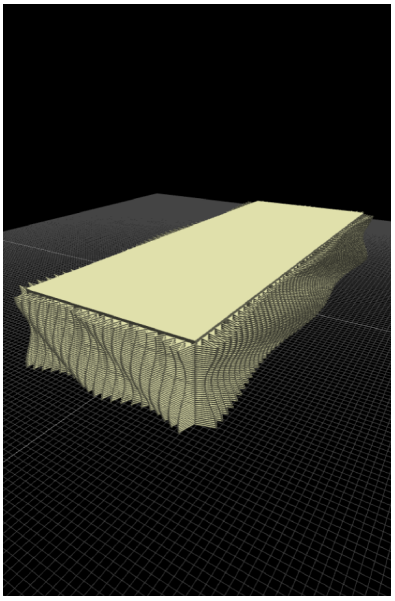
4.2.1 Running Performance

Our solution should provide immediate feedback to architects, therefore letting them easily understand the correlation between the program inputs and outputs[10]. Apart from making it easy to change program inputs, it needs to run programs and show results as fast as possible. It is important to compare the performance of our solution to the performance of other systems that can also provide immediate feedback.

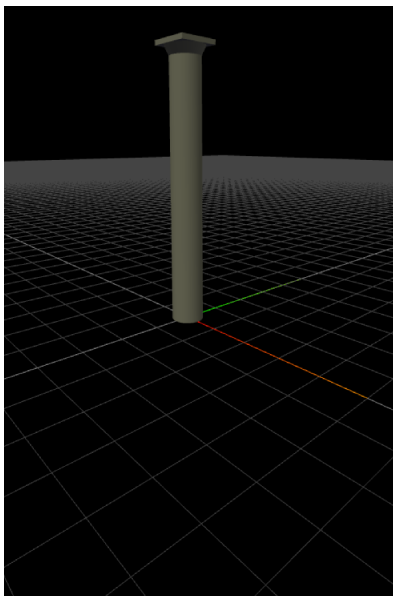
To see how the performance of our web-based GD environment compares to the existing environments, we compared the time each takes to generate identical models. First, we implemented a version of each program using each environment's programming language and, then, measured the time each



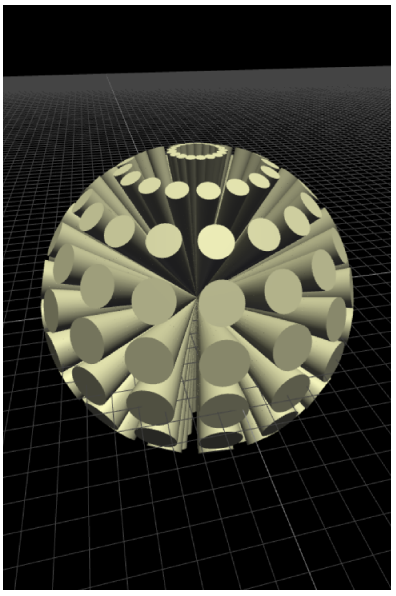
(a) Dom-ino



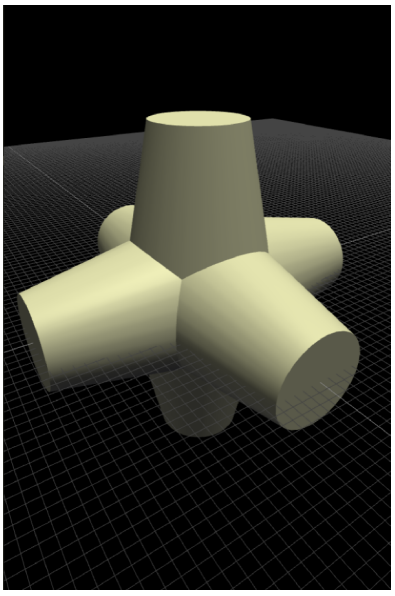
(b) Carmo Facade



(c) Column



(d) Sphere of cones



(e) Orthogonal Cones

Figure 4.9: Renderings of results of the implemented example programs.

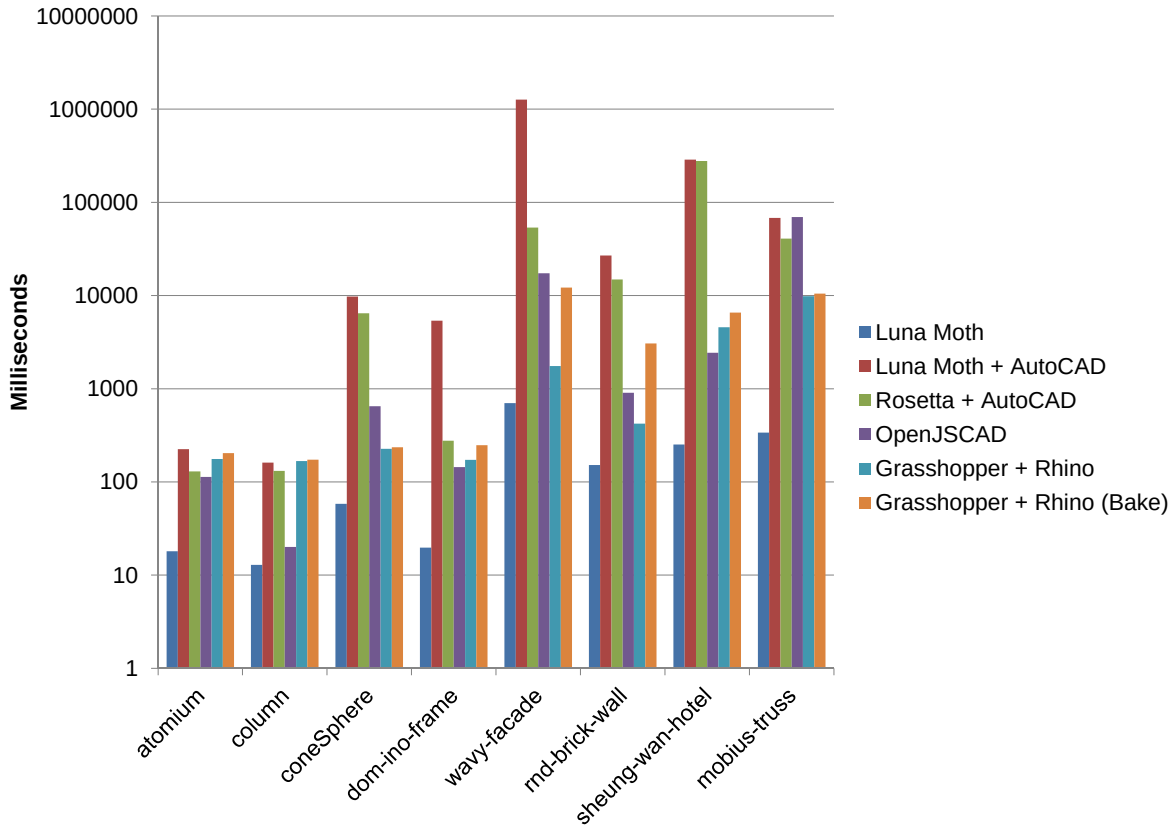


Figure 4.10: Comparison of running times for our IDE, its export process, Rosetta, OpenJSCAD, and Grasshopper. Note that the time is in logarithmic scale.

took to complete. These times are shown in the chart on Figure 4.10. 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*). More specifically, Rosetta's running times range from being around seven times to eleven hundred times the running times of Luna Moth (*Luna Moth*). In the case of OpenJSCAD, its running times range from being around two times to two hundred times the running times of Luna Moth. Finally, Grasshopper's running times range from being two times to thirty times the running time of Luna Moth.

With this in mind, we can say that our solution can provide much better feedback than the other IDEs.

4.2.2 Run vs Export

When time comes to pass on the building design to other people using different tools — like mechanical, electric, plumbing services or even other architects — the architect must provide the design in a format compatible with their tools. This has been covered in our solution by letting the program generating the design run in a CAD application.

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 4.10. 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.

One may need some clarification on how this difference appears. For each primitive call a program does, data has to be converted, sent over an HTTP connection to the remote CAD service, passed to Rosetta, then to the CAD application to actually perform the primitive, and finally a result identifier must be sent back to the web application. The difference comes from the time needed for the CAD application to perform primitives and from the time spent on communication between applications. As shown in [10], CAD applications are not well suited for GD since they were designed to support human interaction and, therefore, are not ready for the amount of operations produced by GD programs.

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 Grasshopper. For this comparison, we took the times from our export process, Rosetta connected to AutoCAD, and Grasshopper baking geometry to Rhinoceros.[1] These correspond, respectively, to *Luna Moth + AutoCAD*, *Rosetta + AutoCAD*, and *Grasshopper + Rhino (Bake)* in Figure 4.10. As can be seen, Luna Moth's export times range from $\approx$1x to $\approx$24x the times of running in Rosetta and range from $\approx$0.9x to $\approx$104x the times of running and baking in Grasshopper.

As shown in Chapter 3, Rosetta plays an important part in the process that Luna Moth uses to send results to CAD applications. Rosetta acts as an interface to communicate with CAD applications, while the remote CAD service makes that interface available to the web application as a web service. As with the export time evaluation in Subsection 4.2.2, the increase in running times comes from the overhead

[1] 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".

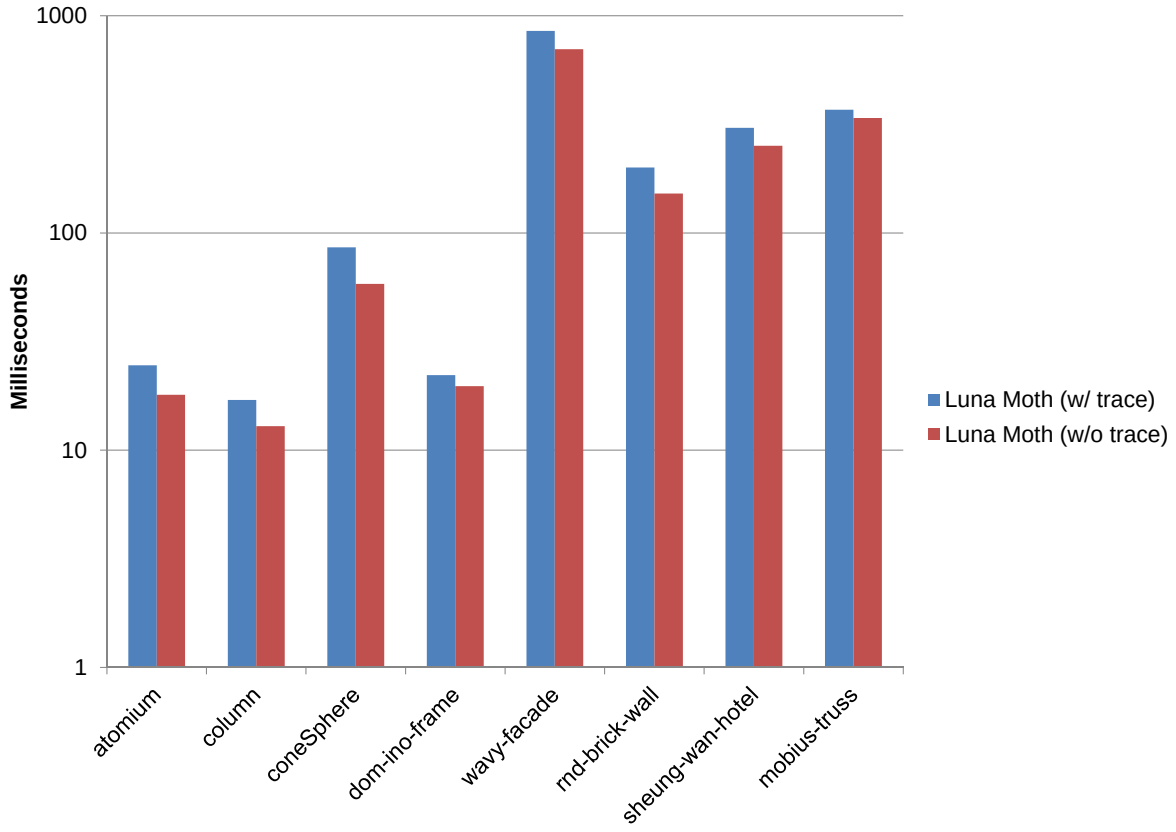


Figure 4.11: Running while collecting traceability data and while not collecting traceability data. Note that times are in logarithmic scale.

in communication between the web application and the remote CAD service. Moreover, although during export programs are run in the web application and Rosetta's role is reduced to make them happen in the CAD application, results are still sent one by one to the remote CAD service, meaning that a lot of time is spent transmitting data. Unlike with the export time evaluation, this comparison lets us distinguish the time of transmitting data from the time taken by Rosetta and the CAD application to apply operations.

4.2.4 Traceability Performance

Collecting traceability data requires that additional work must be done when a program is being executed. The additional work will inevitably increase the time it takes to finish executing the program. This can make the programming environment less capable of giving immediate feedback, therefore having a negative impact in its the usage experience.

To measure the impact that traceability data collection has on program running time, we also ran programs with and without data collection, and measured the time each took to execute. The chart in Figure 4.11 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.

Chapter 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, Chapter 2 explored programming environments for various domains, including GD, and also 3D modeling applications. These included desktop applications like Processing, a programming environment for the visual arts, and Rosetta, a programming environment giving architects program portability for both programming language and CAD application, and also included web applications like IPython, a programming environment for scientific computing with a web page based UI, and OpenJSCAD, an online programming environment for solid modeling.

Firstly, we compared these applications according to their application domain and the purpose they give to programming. Afterward, 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 Chapter 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 addition, to allow for adjusting literals and highlighting the relationship between program and results, Luna Moth analyzes the program's structure to know where literals are and to instrument it for the collection of both results and traceability data.

This allows architects to create programs incrementally. Instead of having to go through the edit-compile-run cycle to see the results of the changes they made, they make the changes and see the effects immediately. Afterward, if they have any doubt on how a certain part of the results came to

be, they can use the traceability mechanisms, pointing at that part, to see the part of the program that created them.

After implementing the solution, we evaluated it in different ways. In Chapter 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. In fact, we were surprised by the difference of at least an order of magnitude from Rosetta and Grasshopper, which shows that CAD applications are indeed too slow to be used in GD exploration. Regarding export times, we saw that exporting from Luna Moth is slower than running on Rosetta. This was to be expected since, although we run the program in the web application, we have to wait for network communications. Nonetheless, the ability to export to CAD applications remains a great advantage for architects since they only need to do it once: when they reached a final solution. 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. Moreover, when we take into account the benefit that showing the program-result relationship brings to the architect and the GD program creation process, the slowdown it brings is well worth it.

In conclusion, with Luna Moth, architects are able to explore GD without needing to install and update the programming environment and avoiding the installation and use of heavy-weight CAD applications, which become slow for GD exploration. By avoiding the complexity of CAD applications, Luna Moth has been able to have much better performance than other GD IDEs and, therefore, have faster feedback. In addition, since it is possible to export program results to CAD when needed, Luna Moth does not tie programs to a particular CAD application. By doing all of this, it fulfills the goals that were proposed, namely, being *accessible* from any computer, providing an *interactive* programming environment for exploring GD, and *integrating* easily with CAD applications that architects use and, therefore, into their workflow.

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.

As a result, we consider Luna Moth as a good step in the direction GD IDEs should take in the future.

The source code for Luna Moth, along with the source code of the example programs, is available on GitHub at `https://github.com/pafalium/gd-web-env`. It requires a modern web browser, like Google Chrome or Mozilla Firefox, to run the web application, requires Racket version 6.4 or above to run the *remote CAD app*, and requires node.js version 5.10.1 or above to be compiled. It has been tested both on Google Chrome and Mozilla Firefox, running on Windows 7. Nonetheless, since these web browsers and Racket are also available for Mac OS, Linux, and other versions of Windows, the web environment and the *remote CAD app* are also available for these Operating Systems.

5.1 Future Work

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

5.1.1 Programming Environment

There are many opportunities for improvement of Luna Moth as a programming environment. The following paragraphs describe some of these and speculate on possible solutions.

Writing aids / Code completion Luna Moth does syntax highlighting, runs programs on change, has traceability, and helps to adjust literals, but it does not include help to find the available primitives and how to use them. This can be improved by providing extensive documentation on the primitive library. However, this is not extensible to functions and variables that the user introduces in his program. Alternatively, we can improve this by analyzing the current state of programs and using the resulting information to provide code completion.

Illustrated Programming There could also be other ways of implementing traceability that would also enable users to document their programs. We could transform programs into something like IPython's notebooks or the examples given in Illustrated Programming[10].

Multiple programming languages We can also follow Rosetta's lead and start supporting more than one programming language commonly used by architects, so there is one less barrier for those who already know one language and want to avoid learning a new one. This would also attract people from the communities around the added languages.

Error reporting and debugging Debugging is currently dependent on the web browser developer tools, which might be too complex for architects that do not have professional programming experience. This suggests that a dedicated debugging tool, targeted to the intended users, would significantly improve the debugging experience.

One possible functionality for this debugging tool would be to try to show incomplete results up to the point where the program reached an error. On top of this, the tool could show and highlight the parameters of the call that produced the error in the 3D view and also highlight the corresponding call expression in the source code.

Timetable traceability We can also improve Luna Moth's traceability. One way it can be done is by implementing the timetables presented in Learnable Programming[24]. This would worsen the performance of the running process if implemented naively. It would nonetheless help architects understand how their programs ran.

Camera controls There are also some improvements that can be made to the way the user visualizes the results. Currently, the 3D view only lets the user navigate the 3D space as if moving around a turntable. It would be good to give him other ways of navigating, like flying or walking through the model. Furthermore, it would also be important to let him define different viewpoints that he could quickly switch between to get a better grasp of the results.

Improve code navigation Depending on the size of a project, it may be easier or harder to find a particular function or variable definition. If GD projects get big enough, it will be advantageous to support navigation. Navigation support can take inspiration from LightTable's code document and namespace view. Another alternative would be the "go to definition" functionality present in most software development IDEs.

Adjustment of literals An extension to helping adjust literals could also be done. Instead of just allowing the architect to change the parameters of his programs by adjusting literals in the source code, we could provide something like the 3D view handles from Antimony. Like so, when Luna Moth detected the definition of positions or vectors from literals, it would display distinct handles in the 3D view that the architect could interact with and, consequently, change the respective definitions in the source code. This would make the adjustment of parameters representing 3D values more intuitive, helping the architect understand more clearly the purpose of those parameters.

Primitives Further work must also be done to implement common primitives in architecture, such as CSG operators, control-point based surfaces and lines, or sweep and loft operators, to name a few.

User testing Finally, it would be important to have more user testing. By doing it, we can start steering Luna Moth's development according to the needs and difficulties of the users and, therefore, make sure it remains both useful and easy to use.

5.1.2 Cloud-related Improvements

Apart from opportunities in the programming environment, there are also some opportunities for improvement in areas that require communication between components. As before, the following paragraphs describe some of these and speculate on possible solutions.

Security One area that needs to be improved is security. Some of the pertinent points where it must be in place are the communication with remote CAD services, that must be private, authenticated and authorized, and the program running process, that must be sandboxed to stop them from changing Luna Moth's web page for malicious purposes.

Persistence and collaboration We presented persistence and collaboration as advantages of cloud applications, however, we did not implement them in Luna Moth. It would be interesting to investigate these features in the future. Moreover, it would also be of interest to integrate Luna Moth with cloud version control services like GitHub or cloud storage services like Google Drive (as happened to IPython notebooks with the coLaboratory[1] and Jupyter Drive.[2] projects)

Export performance The performance of the export process can also be improved. Instead of using HTTP requests to communicate with the application running on the user's computer, a WebSockets connection[26] could be used to avoid the overhead of HTTP. Alternatively, Luna Moth could send the entirety of the results on a single HTTP request, therefore, decreasing the time spent waiting for data traversing the network and receiving a response. Another option would be to run the user's program in the *remote CAD app* instead of the web page.

Workflow Integration As seen in Chapter 3, Luna Moth uses Rosetta to connect to CAD applications installed in the architect's computer. As of recently[27], Rosetta has been extended with backends for lighting analysis tools, such as Radiance[3] and DAYSIM,[4] with planned support for acoustic and structural

[1] http://colaboratory.jupyter.org/welcome/ (last accessed on 10/05/2017)
[2] https://github.com/jupyter/jupyter-drive (last accessed on 10/05/2017)
[3] https://www.radiance-online.org/ (last accessed on 10/05/2017)
[4] http://daysim.ning.com/ (last accessed on 10/05/2017)

analysis tools. We could take advantage these capabilities and also add support for these analysis tools to Luna Moth.

Bibliography

[1] I. E. Sutherland, “Sketchpad: a man-machine graphical communication system,” in *Proceedings of the SHARE Design Automation Workshop*, DAC ’64, (New York, NY, USA), pp. 6.329—-6.346, ACM, 1964.

[2] A. Watson, C. Solutions, C. S. Staff, and C. T. (Firm), *GDL Handbook: A Comprehensive Guide to Creating Powerful ArchiCAD Objects*. Cadimage Solutions, 2009.

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

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

[5] 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.

[6] J. Lopes, “Modern Programming for Generative Design,” 2012.

[7] M. Flatt and R. B. Findler, “The racket guide,” 2016.

[8] G. Rossum and F. Drake, “The python language reference manual,” *Network Theory Ltd*, 2003.

[9] 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.

[10] 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.

[11] I. Hickson and D. Hyatt, “Html5: A vocabulary and associated apis for html and xhtml,” *W3C Working Draft, May*, vol. 25, 2011.

[12] C. Marrin, “Webgl specification,” *Khronos WebGL Working Group*, 2011.

[13] A. Sorensen, “Impromptu: An interactive programming environment for composition and performance,” in *Proceedings of the Australasian Computer Music Conference*, no. July, pp. 2–4, 2009.

[14] A. Sorensen and H. Gardner, “Programming with time: cyber-physical programming with impromptu,” *ACM Sigplan Notices*, vol. 45, no. 10, pp. 822–834, 2010.

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

[16] M. McGranaghan, “Clojurescript: Functional programming for javascript platforms,” *IEEE Internet Computing*, vol. 15, no. 6, pp. 97–102, 2011.

[17] 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.

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

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

[20] 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.

[21] B. Houston, W. Larsen, B. Larsen, J. Caron, N. Nikfetrat, C. Leung, J. Silver, H. Kamal-Al-Deen, P. Callaghan, R. Chen, *et al.*, “Clara. io: full-featured 3d content creation for the web and cloud era,” in *ACM SIGGRAPH 2013 Studio Talks*, p. 8, ACM, 2013.

[22] P. Janssen, R. Li, and A. Mohanty, “MÖBIUS: A parametric modeller for the web,” in *Living Systems and Micro-Utopias: Towards Continuous Designing, Proceedings of the 21st International Conference on Computer-Aided Architectural Design Research in Asia (CAADRIA 2016)* (S. Chien, S. Choo, M. A. Schnabel, W. Nakapan, M. J. Kim, and S. Roudavski, eds.), pp. 157–166, 2016.

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

[24] B. Victor, “Learnable Programming,” *http://www.worrydream.com*, 2012.

[25] R. Aish, “Designscript: Scalable tools for design computation,” 2013.

[26] A. Melnikov and I. Fette, “The WebSocket Protocol.” RFC 6455, Dec. 2011.

[27] A. Leitão, R. Castelo Branco, and C. Cardoso, “Algorithmic-Based Analysis - Design and Analysis in a Multi Back-end Generative Tool,” in *Protocols, Flows, and Glitches - Proceedings of the 22nd CAADRIA Conference* (P. Janssen, P. Loh, A. Raonic, and M. A. Schnabel, eds.), CAADRIA, (Xi’an Jiaotong-Liverpool University, Suzhou, China.), pp. 137–146, 2017.