ILLUSTRATED PROGRAMMING

ABSTRACT

In the area of Generative Design, programs are becoming increasingly complex and harder to understand, communicate, and share, enlarging the gap between them and the architectural concepts they implement. To overcome this problem, we need to develop documentation techniques and program comprehension tools targeted to the Generative Design domain. This paper proposes Illustrated Programming as a coherent approach for improving program documentation and program comprehension, by establishing a correlation between the intended design, the Generative Design program, and the generated model. This correlation is achieved by the inclusion of sketches within programs and by bidirectional traceability and immediate feedback between programs and models.

1 On the left, the geometric structure and system of proportions of Milan’s cathedral, depicted by Cesare Cesariano in his translation and illustration of Vitruvius’ De Architectura (1521). On the right, Peter Eisenman’s (1961-1971) diagrams for House II (Miller House) show the evolution of the concept through a sequence of rotations of orthogonal grids.

António Leitão Universidade de Lisboa
José Lopes Universidade de Lisboa
Luis Santos UC Berkeley
INTRODUCTION

Architects have always been concerned with explaining their designs and, given that most of the architectural work has a strong visual component, it is not surprising that drawings are the preferred media for those explanations. In fact, ever since the invention of linear perspective by Brunelleschi (Tsuij, 1990), the act of \textit{projectum} (interpreted here in its Latin sense of anticipating a reality) has been being quintessential to the architectural practice. As an example, consider (Figure 1), which shows two sets of drawings separated by 440 years of architecture, proving that the importance of drawing has not changed over time in this field.

Drawings are powerful design tools because they convey complex ideas in a compact medium. Although the creative process in architecture is not linear, architects can synthesize the different design steps into a logical sequence of diagrams/sketches, which in the end clearly document the design decisions, the relationships between different parts of the design, and the impact of external factors in the final shape. This kind of illustrated narrative is extremely effective in telling the story of a specific design as well as a way of thinking about and solving design problems (Do et al., 2001), therefore, it has been extensively used, namely, in pedagogical books, empirical studies of drawing in design, and architectural publications and competitions. (Figure 2) shows a sequence of manually sketched diagrams from Louis I. Kahn’s Goldberg House.

The need for documenting the design process is evident, particularly in architectural projects developed (partially or fully) through \textit{Generative Design (GD)} programs. By \textit{Generative Design}, we refer to the use of algorithms, implemented through programming languages, to create geometric shapes. Due to the increased complexity of both GD programs descriptions and the shapes those programs can achieve, explaining a program’s structure, behavior, and parameters, is critical.

By definition, the GD program can itself be considered a description of a design, as it formally specifies the modeling process of the design. Unfortunately, this formal specification can only be easily understood for simple design problems. For any sufficiently complex program, it is additionally helpful to have program documentation and tools for program comprehension. (Storey, 2006)
Program documentation is very important for software development and maintenance (Souza et al., 2005). Unfortunately, writing documentation is perceived as a tiresome task and thus avoided. This negatively affects software development in general and GD in particular. In the current state of GD, program documentation is generally poor, making it very difficult not only to understand any non-trivial GD program, but also to share, adapt, and extend such programs.

In fact, the little documentation that does exist comes in books and research papers, where fragments of programs (McCullough, 2006, Silver, 2006, Preisinger, 2013, Hemmerling, Marco and Lemberski, 2012) or even entire programs (Williams, Chris J. K. and Kontovourkis, 2008) are proudly presented as illustrations or even used as background images. While in many cases these do achieve interesting aesthetic results (Miller, 2011), as is visible in Figure 3, their explanatory power is limited, and the program that is being illustrated is, in general, much larger than what can reasonably fit in one or two pages. This forces authors to drastically zoom out the program, rendering it unreadable (Castro e Costa, 2012) or to present only a handful of small program fragments (Buell et al., 2011), leaving the rest undocumented. In the end, these programs are not useful as program documentation: they are merely used to show the kind of programming that is used (for example, textual or visual) or the degree of complexity reached in the elaboration of the GD program.

When program documentation is poor, obsolete, or absent, the remaining option is to study the program itself, a process known as program comprehension.

Program comprehension is the process of acquiring knowledge about a program (Rugaber, 1995), which allows us to create a mental model of the program’s structure and behavior. This is a necessary step before making any modifications to the program. When there is not enough documentation, this mental model must be constructed from reading the program. Unfortunately, due to the large amount of detail required by current programming languages, designers must spend a significant mental effort to extract the relevant architectural ideas from the irrelevant details.
In the field of GD, the relevant architectural ideas are conveyed in the form of geometric models. In order to adapt the program to generate a different geometry, it is first necessary to understand the relationship between the program and the generated model. It is thus important to correctly identify which part of the program is responsible for a given part of the generated model and which parts of the model were generated by a given part of the program. This identification is normally not trivial, but, as we will show, it can be drastically improved with adequate tools.

ILLUSTRATED PROGRAMMING

In order to (1) mitigate the problem of lack of documentation and (2) improve program comprehension, this paper proposes Illustrated Programming (IP), a programming approach that establishes a correlation between the intended design, the corresponding GD program, and the generated models. This correlation encompasses two independent but related ideas:

1. Sketch-program correlation, where sketches embedded directly in the program document the correlation between architectural concepts and the corresponding parts of the GD program

2. Program-model correlation, where the Integrated Development Environment (IDE) allows the user to identify which parts of the program are responsible for which parts of the generated model and vice-versa

In short, the sketch-program correlation provides an explanation for the structure of a GD program whereas the program-model correlation provides an explanation for its behavior. In the next sections we detail these two aspects of IP and explain its implementation in Rosetta, a development environment for portable Generative Design (Lopes, José and Leitão 2011).

SKETCH-PROGRAM CORRELATION

In the past, there were attempts in the field of software engineering to improve the quality of program documentation, most prominently, literate programming, a programming paradigm that promoted the fact that programs are written for people first and foremost, and that documentation should be emphasized just as much as code. Unfortunately, these attempts did not reach the intended goals, mainly because writing good documentation takes a considerable amount of time and effort.

However, the reality in architecture is quite different from that in software engineering: it is part of the design process to produce documentation in the form of sketches. This means that it is not necessary to write huge amounts of textual documentation to explain a GD program. We only need to annotate the already existing sketches and combine them with the program, thus providing visual explanations of what the program is supposed to do.

(Figure 4) shows Rosetta, which runs in DrRacket (a descendant of DrScheme (Findler et al., 2002), implementing this process. In the image, we can see sketches developed by the designer for the Atomium building, that explain to the programmer the intended design, alongside the program that implements it. Note the annotations on the sketches, which clearly identify the parameters of the program.

Although, in many cases, a designer and a programmer are a same person, this is not strictly necessary. Actually, this approach also promotes collaborative design processes, where each participant assumes a different role (Santos et al., 2012). In the presented case, the designer and the programmer were in fact in two different continents.

Using sketches as documentation and combining them with the GD program allows us to establish a Sketch-program correlation. However, this correlation does not tie in any way the program code with the produced 3D model, which we discuss in the following section.

PROGRAM-MODEL CORRELATION

In GD, the designer interacts with a computer program, which can be seen as an intermediary between the concept the designer wants to achieve and the geometric model produced by that
However, current programming languages require a large amount of details, which are directly related to the programming language and not to the design. Therefore, they complicate programs and interfere with their comprehension, making it harder to understand the relationship between program and the generated geometric model. In order to overcome this problem, we resort to the concepts of traceability and immediate feedback.

**TRACEABILITY**

Traceability is the ability to establish a relationship between the elements of the program and those of the model, and it is particularly important for program comprehension, maintenance, and debugging. Without it, it can be difficult to understand the causes of errors, the changes needed to adapt a GD program to different purposes, or the impact of changes to a program.

Various techniques have been employed to improve traceability in GD. For example, in Grasshopper, when the user selects a component that generates geometry, the corresponding part in the model is highlighted. Even more helpful would be the converse association, that is, selecting a shape in the model to automatically highlight the corresponding program component, but, unfortunately, this is not supported.

Moving from visual to textual programming languages, such as, RhinoScript or AutoLisp, the situation becomes considerably worse. In general, there is no traceability at all, at least, not one that relates the program with the model.
In order to improve GD traceability, we implemented in Rosetta a bidirectional link between the program and the model. Starting from the model, it is possible to point to some shape element and immediately identify the part of the program that was responsible for its creation. Starting from the program, it is possible to identify which elements of the model were generated from each element of the program. This is shown in (Figure 5) and (Figure 6).

(Figure 5) illustrates a typical scenario where the user selects an expression in the program and Rosetta shows the set of shapes that resulted from that expression. Note that this set contains all shapes that were created by the expression during the complete execution of the program. (Figure 6) shows navigation in the opposite direction: selecting an element of the model in the CAD tool instructs Rosetta to highlight the program elements that were responsible for its creation.

Note that the correlation between the program and the generated model allows the designer to use both approaches at the same time, moving from one to the other as he sees fit, thus speeding up the comprehension process. Moreover, traceability can be selectively enabled on a per-module basis. As a result, modules for which traceability is enabled highlight the entire trace, whereas, modules for which traceability is disabled are treated as a black-box, that is, only the entry/exit points are shown. This is especially useful for reducing the amount of visual noise in the highlight and focus only on the parts of the program that are relevant.

The current implementation of traceability uses instrumentation, which consists of adding instructions to the program such that each entry/exit point of a function can be recorded and later reconstructed to show a trace. Program performance is sensitive to this technique because the more entry/exit points exist the more instrumentation is used. For example, a program that produces hun-
dreds of shapes in a single function will run faster than a similar program that produces the same shapes spread across hundreds of functions. At the moment, we are working on this problem and intend to improve performance in the future.

IMMEDIATE FEEDBACK

Traceability allows an architect to understand the correlation between a GD program and the generated model. However, it does not allow the designer to easily understand the correlation between the program inputs and outputs. To this end, the program must be re-executed for each different set of inputs and the model re-visualized, a slow-pace process that will bore even the most patient architect. Immediate feedback attempts to solve this problem by allowing the designer to quickly visualize the impact of changes to the program inputs. With this mechanism, the designer adjusts the program inputs, which have an immediate effect on the generated model, until this model reflects his intentions. This not only allows for better correlation between the program and the model but also allows designers to endeavor in design exploration.

Many design tools acknowledge the usefulness of immediate feedback. This can be seen in the ability of some GD tools, such as, Grasshopper and Rosetta, to connect program inputs to specialized widgets, such as sliders, that react to changes by recomputing the generated design. Each change in a slider causes Rosetta to recompute the design. However, this re-computation process only operates in real time for very simple GD programs. Complex programs can take a considerable amount of time to re-compute and the interactive use of input widgets can become annoying, a problem that affects both Grasshopper and Rosetta.
Unfortunately, this problem cannot be easily solved because not all program operations can compute in constant time. A more reasonable assumption is that computation time grows linearly with the input size. This is what happens, for instance, with Grasshopper components that map a given operation over a sequence of values: we can expect the computation time to be at best proportional to the length of the sequence. However, in the case of multiple sequences operated in cross-reference, the computation time becomes at best polynomial, making it difficult to have immediate feedback. Adding recursion to the program can make it more difficult still, as it opens the door for computations that require exponential time (or worse). As a result, immediate feedback will never scale to arbitrarily large inputs, particularly when we depend on CAD tools that were designed for the speed of human operation, and not for the large volume of operations generated by GD programs.

This situation can be improved by sidestepping most of the functionality of traditional CAD tools and focusing only on the generation and visualization of a geometric model. To this end, Rosetta includes a backend that does not depend on a full-fledged CAD application. Instead, it connects almost directly to the graphics device of the computer, using the OpenGL graphics library. (Figure 7) shows several Rosetta programs and the corresponding models in this backend.

This backend allows much faster rendering and, as a result, the designer can enjoy immediate feedback for larger inputs and more complex designs. Once satisfied with the design, he or she can then switch to a normal CAD backend, such as, Rhinoceros or AutoCAD, and continue working as before. Note that, by using Rosetta, switching backends does not entail any change to the GD program being developed. (Table 1) presents the time needed by different backends for updating identical geometry, clearly showing that the OpenGL backend is considerably faster than the others.

<table>
<thead>
<tr>
<th>Example/Backend</th>
<th>AutoCAD</th>
<th>Rhino</th>
<th>OpenGL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orthogonal Cones</td>
<td>1022</td>
<td>191</td>
<td>1</td>
</tr>
<tr>
<td>Moebius Truss</td>
<td>24253</td>
<td>6094</td>
<td>217</td>
</tr>
<tr>
<td>ScriPecture</td>
<td>10712</td>
<td>2994</td>
<td>67</td>
</tr>
</tbody>
</table>

Table 1: Time (in milliseconds) needed to update the generated design.

At this moment, the OpenGL backend is still being developed and only supports a subset of the functionality that is provided by the other backends. However, our evaluation which is summarized in (Table 1) shows promising results.

RELATED WORK

The idea of Illustrated Programming was inspired both by Literate Programming (Knuth 1984) and Learnable Programming (Victor 2012a, 2012b).

Literate Programming (LP) is a programming paradigm invented by Donald Knuth in 1984. At that time, substantial improvements had been made in programming methodology but little progress had been made in program documentation. To significantly advance this aspect of programming, Knuth advocated that programs should be considered works of literature, written using prose and a good presentation order, for a human audience. Then, two different tools were used: weave produced a nicely formatted and indexed document for human consumption, while tangle extracted and composed the source code so that it could be compiled and executed by a computer.

LP was not widely adopted due to the considerable effort needed to write good prose and a good explanation of a program. However, GD is a specific domain where sketches already carry an important explanatory role and usually exist beforehand. By including them in GD programs, we make those explanations easily available to anyone that wants to understand the program, while avoiding the considerable effort needed for writing extensive textual documentation.

There are other domain specific languages for GD, such as Grasshopper, which, given their visual nature, allow to treat the development environment as a canvas where both drawings and programs can coexist. Unfortunately, for non-trivial programs, this canvas tends to become huge, making it difficult to navigate within the program and understand its structure and behavior (Leitão et al., 2012). It is possible to reduce the apparent size and complexity of a program through the use of clusters and wireless connections, but it still remains difficult to understand large and complex programs.

In the case of textual programming languages used in GD, such as RhinocerosScript, AutoLisp, and DesignScript, they do not support the inclusion of images in the code. Rosetta does not have this limitation, as it takes advantage of the support provided by DrRacket for literate programming (Flatt et al., 2009) and for inclusion of images in programs, thus significantly improving sketch-program correlation.
Regarding program-model correlation, we observe that some visual GD programming languages support, at least partially, traceability and immediate feedback. In the case of traceability, both Grasshopper for Rhino and Dynamo for Revit/Vasari can highlight the particular geometry generated by a selected set of components/nodes. However, the opposite, for example, selecting the geometry and highlighting the parts of the code that generated it, is still not possible.

Immediate feedback is also supported in Grasshopper and Dynamo via sliders and live programming. Each change in a slider re-executes the program using, as input, the values of all sliders, allowing the visualization of the impact of different inputs in the final output. Live programming re-executes the program on each program change, allowing the designer to build the program incrementally while visualizing its output. In the domain of textual languages for GD, both Autodesk’s DesignScript (Aish, 2012) and Rhino’s Yeti (Davis et al., 2012) support live programming. Although DesignScript does not fully support sliders, its integration with the Dynamo platform overcomes this limitation.

Compared to the previous languages, Rosetta provides almost the same features but goes even further in the case of traceability, by providing full bi-directionality. Regarding immediate feedback, Rosetta contains an OpenGL backend, which is much faster than the usual CAD software at the expense of only providing visualization services. Rosetta currently does not allow live programming, but there are live programming experiments using DrRacket (McLean et al., 2010) that we plan to explore in the future.

There are other studies that confirm our point of view, for example, Programming-in-the-model (PIM) (Maleki, Maryam and Woodbury, 2013). PIM uses three views over a single design, namely, the model, graph, and the script. The model view is the one normally shown in CAD software, the graph view shows the dependencies through a node-link diagram similar to that of visual languages, and the script view shows the code. These views are synchronized such that changes in one view are reflected in the others. Also, the correspondence between elements is highlighted in the different views when a component is selected.

There are two important differences between Rosetta and PIM. First, PIM promotes live programming, which requires either very simple GD programs or inordinate amounts of computational power, while Rosetta only uses traceability and immediate feedback on demand, thus supporting more complex programs in common computers. Second, in spite of the use of multiple views, PIM does not seem to allow the inclusion of sketches in the code.

CONCLUSION

Generative Design is reaching a point where the programs are becoming so complex that it is now important to develop good tools for program documentation and program comprehension.

Based on Literate Programming and Learnable Programming, we introduce the concept of Illustrated Programming. This concept states that a good GD Programming Environment should help the designer in establishing a strong correlation between the design, the GD program, and the generated model.

To this end, we extended Rosetta, a tool for portable GD, to allow the integration of sketches in the GD program and to provide traceability and immediate feedback. Sketches are used as program documentation, establishing a link between the intended design and the program. Traceability allows the visualization of the bi-directional link between the GD program and the generated geometry. Finally, immediate feedback allows the user to quickly visualize the impact of changes to the program inputs.

With Illustrated Programming, we improve the quality of GD programs by offering a visual explanation of the structure and behavior of the program. This facilitates development, understanding, and maintenance of programs. Moreover, it promotes program sharing, communication, and collaborative work throughout the design process.

We are currently evaluating Illustrated Programming in large case studies to understand the extension of its benefits and to diagnose and correct its limitations. We plan to improve the Sketch-program correlation with a better mechanism for updating a sketch (currently, the updated sketch needs to be manually reinserted in the program) and with a visual notification of the up-to-date status of program fragments and corresponding sketches. Regarding the Program-model correlation, we will improve immediate feedback by optimizing the OpenGL backend implementation and by introducing live programming. However, we believe that the live programming mode should be optional and only available in the OpenGL backend so that it can have an acceptable performance for larger GD programs.

ACKNOWLEDGMENTS

This work was partially supported by Portuguese national funds through Fundação para a Ciência e a Tecnologia (FCT) under contract Pest-OE/EEI/LA0021/2013, by the Rosetta project under contract PTDC/ATP-AQI/5224/2012 and by the PhD grant under contract SFRH/BD/98658/2013.
REFERENCES


IMAGE CREDITS


Figure 4-7. Image credit to Authors (2014).

ANTÓNIO LEITÃO is assistant professor at Instituto Superior Técnico, teaching Computer Science to students of Architecture, and researcher at Inesc-ID, working on the application of Computer Science methods to the problems of Architecture.

JOSÉ LOPES is a Software Engineer at Google, passionate about programming language design and implementation, compilers, and type systems.

LUIS SANTOS is a Portuguese architect and researcher that works on bridging design practice and research on generative and performance-based design. He recently was accepted in the PhD architecture program at UC Berkeley, College of Environmental Design.