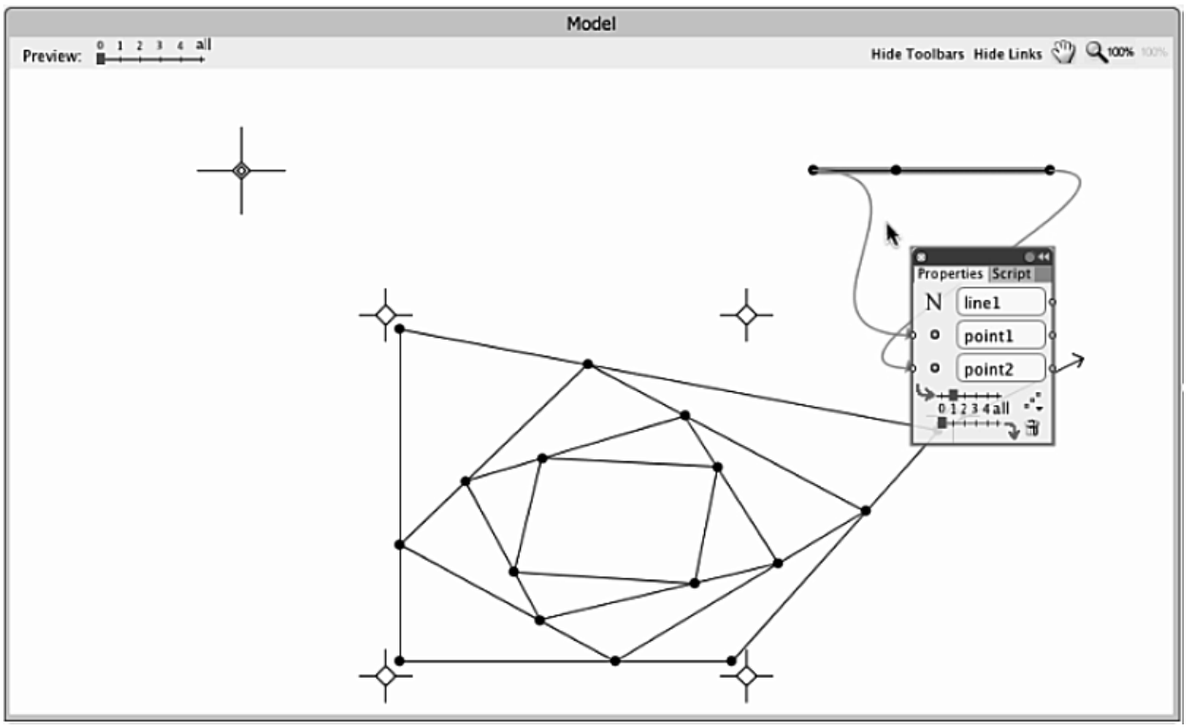
PROGRAMMING IN THE MODEL
A NEW SCRIPTING INTERFACE FOR PARAMETRIC CAD SYSTEMS

Maryam M. Maleki
Robert F. Woodbury
School of Interactive Arts and Technology, Simon Fraser University



6 Localized representation of dependencies in the model view in PIM

ABSTRACT

Programming, often called scripting, has become a key feature in most CAD systems and an equally key area of expertise in CAD. However, programming surrenders many of the benefits of direct manipulation and introduces notational elements that are cognitively distant from the designs being created. In addition, it creates barriers to use and is often perceived as being too difficult to apply. We introduce Programming In the Model (PIM) through a prototype, implementing live side-by-side views, multi-view brushing and highlighting, live scripting, auto-translating from modeling operations to script and localized relational information within model windows. A qualitative user study confirms PIM's features and raises issues for future development. A key result is the need for multi-directional *extreme liveness*, that is, maintaining consistency of action across views at the smallest possible scale. We argue that PIM principles are applicable in textual and visual programming alike.

1 INTRODUCTION

Computer programming can improve designers' efficiency by allowing them to program iterative operations (Burry 1997: 493), making modules for future use (Woodbury 2010: 30), and helping others by sharing code (Gantt and Nardi 1992). It also helps them with their design task by giving them freedom from the limitations of the interface (Streich 1992: 402) and allowing them to explore unconventional and more complex forms (Aish 2003: 340). CAD developers try to perfect their software to meet designers' needs, but at some point designers will encounter the software's limits and want to exceed them, such that their creativity and designs are not limited or determined by the system. Aish (2003: 339) argues that one of the key requirements for exploratory design is having geometric freedom, which can be achieved by using computer programming to access the hidden functions in the computational tool that are not usually exposed in the GUI. To do so, designers must have necessary programming skills. This is especially true in using parametric CAD systems when designers must think about the underlying data structure of the geometric model and define the relationships between geometric components (Woodbury 2010: 23). These domain experts who write code become *end-user programmers* with varying levels of programming skill (Nardi 1993).

While designers are proficient with the two-dimensional or three-dimensional model itself, most of them are not as comfortable with programming/scripting environments. Like other end-user programmers, they face a challenge when they move away from their domain and enter the computer programming space. Their goal is not to become professional programmers or to create the most efficient, reusable code, but to write code to support themselves in the task in hand (Ko et al. 2011). Thus, they think twice before starting to use a new tool by weighing the time and effort that learning the tool takes against the benefits it brings to their work (Blackwell 2002). Since the scripting interface and its notation are very different from the CAD modeling interface, this creates a fear of code among designers that prevents them from learning scripting and benefiting from its capabilities in their design process.

In this research we address this issue in parametric CAD systems and suggest several principles and features in the context of a prototype we call PIM, short for Programming In the Model. We first explain the cognitive issues of programming and how we address them in PIM. Then we describe PIM features and present the results of our user study. Finally, we discuss our findings and future directions.

2 THE COGNITIVE BASIS FOR PIM

2.1 PROGRAMMING IN CAD

Blackwell (2002: 5) summarizes the primary cognitive features of programming tasks as "a) loss of the benefits of direct manipulation and b) introduction of notational elements to represent abstraction." Let us see what these mean for modeling tasks in CAD:

LOSS OF DIRECT MANIPULATION

In CAD, direct manipulation is the primary method of interaction with the model. CAD users click on geometric objects such as points and lines to edit them, click on a location in the model space to specify coordinates, and click and drag parts of the model to move or scale them. Programming in CAD is usually done in a scripting window that is separate from the model and in most cases temporarily blocks access to the model until the script window is closed. Designers must focus their attention on a new window and interact with programming elements instead of the model that is the subject of their design. As a result, they lose the benefits of direct manipulation, including immediate visual feedback on their actions (Shneiderman 1983: 59) and the sense of directness between their thoughts and the actions of the system (Hutchins, Hollan, and Norman 1985: 317).

PIM'S APPROACH

By looking closer at scripting in CAD, we see that it usually comprises the same actions as modeling tasks, such as creating new objects, editing existing objects and modifying their relationships. We aim for an environment that gives users the option to continue using the same modeling tools and to manipulate the model directly during scripting.

INTRODUCTION OF A NEW NOTATION

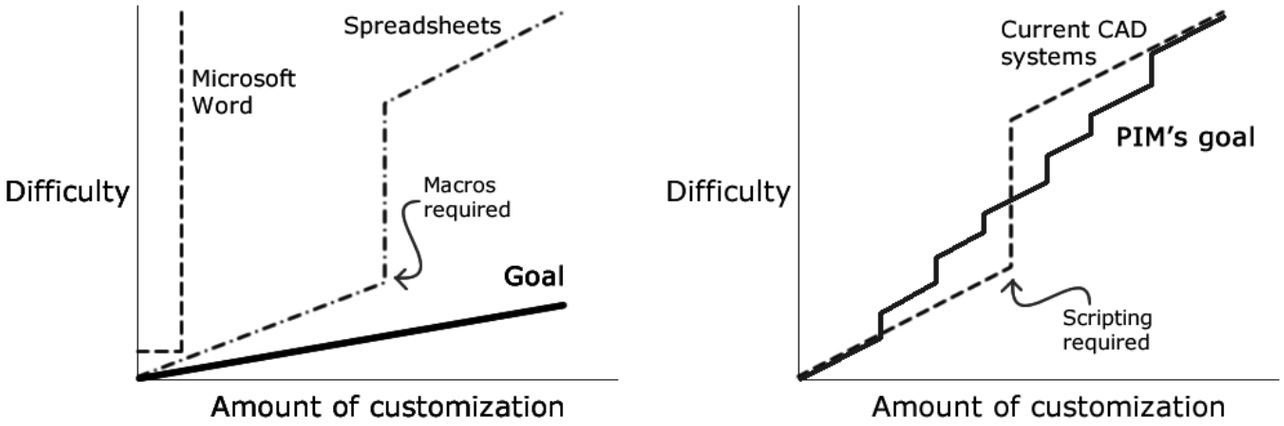
Modeling notation (and it is a notation, just a graphical one) comprises line, curve, surface and solid components; move, rotate, copy and scale operations; dimensions and materials. The scripting language, however, is a totally different one. Depending on the system, functions, loops and conditionals, classes and instances, arguments, variables and types, and commas, semicolons and brackets are used to write a program.

PIM'S APPROACH

We propose an environment that allows users to employ regular modeling and GUI notations during programming, then translates their modeling actions into the programming notation and vice versa. This gradual generation of the code is a learning tool for novice end-user programmers. More experienced end-users may choose to let the system generate simple pieces of code, but write the more complex ones in the script.

2.2 LEARNING BARRIERS

Dertouzos (1992) and later (Myers, Smith, and Horn 1992) introduce the concept of a "gentle slope system" (Figure 1). To customize such systems, users only need to learn a small number of features. In other words, they climb a small step to move forward. Some systems require a lot of learning before users can accomplish a task. Often, users hit a wall that they need to climb before they can continue. As shown in Figure 1, spreadsheets are relatively easy to use, up until the point when users open VBA



1 left: Gentle slope systems.

2 right: PIM's goal is to break down the learning step between modeling and scripting to create a gentle slope system.

(Visual Basic for Applications) to write a piece of code. That is when they face a steep learning curve because of the new programming language and lack of direct access to the spreadsheet interface.

PIM'S APPROACH

The goal of Programming In the Model is to create a more gentle slope system by breaking the barrier between modeling and scripting into small, able-to-be-reordered steps. Bearing in mind that PIM is not a new programming language, we do not claim to reach the perfect shallow curve in Figure 1 (labeled as Goal). PIM operates on existing CAD systems and their scripting languages. We accept the level of difficulty of these scripting languages and only claim to break the difficulty curve into smaller steps that make it easier for end-user programmers to learn and use these languages (Figure 2).

2.3 FEAR OF CODE

According to Blackwell's (2002) attention investment model, users weigh the perceived costs and risks against the immediate rewards before attempting an action. If writing a piece of code seems too difficult or time consuming, end-user programmers may choose to manually perform a repetitive task and move on, instead of investing attention in programming.

PIM'S APPROACH

We hope to lower the perceived cost/benefit ratio of scripting in PIM compared to traditional scripting languages. PIM aims to break the fear of code in order to encourage designers to use small pieces of code more frequently, therefore making it more likely for programming to become a tool in their design process.

3 PROGRAMMING IN THE MODEL (PIM)

In this section we introduce a number of ideas and recommendations for a scripting interface in CAD with a more gentle slope that reduces the fear of code in designers and helps them overcome some of the barriers that prevent them from using programming in their work. We describe these ideas using a prototype that we call PIM, which includes a limited number of geometric

objects and an interface that comprises a model window, a dependency graph (node-link) diagram, and a script window. The current implementation of PIM, showing only a subset of our ideas together with a video demo, presents a complete picture of PIM features. We created PIM as a platform to implement and test these ideas, not as a new CAD system. Our hope is that these ideas be adapted and applied to existing CAD systems.

SIDE-BY-SIDE AND LIVE WINDOWS

In most CAD systems, an open script window blocks the model and freezes it, meaning that users cannot interact with the model or graph until they close the script window. PIM shows all representations of the design side-by-side, including the model view, the dependency graph, and the script window. They are all concurrent and interactive, so designers can access the model and the graph during scripting (Figure 3).

HIGHLIGHTING TO NAVIGATE THE MODEL, THE GRAPH, AND THE SCRIPT

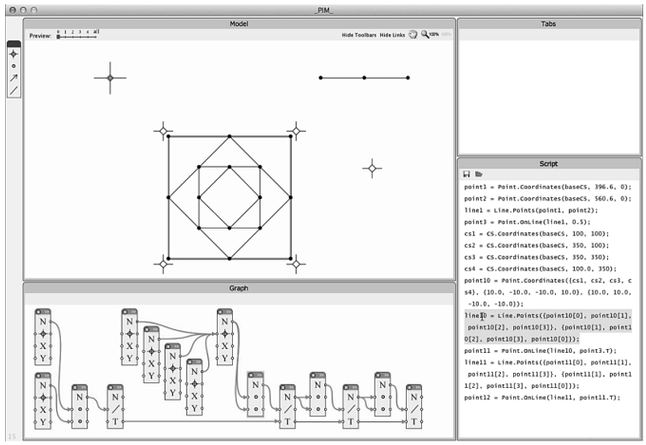
To help navigate through these representations, PIM offers brushing and highlighting of the data in the script, graph and model. When users hover the mouse over an object in any of these windows, all references to that object are brought into focus and highlighted in the other windows (Figure 3).

LIVE SCRIPTING FOR IMMEDIATE FEEDBACK

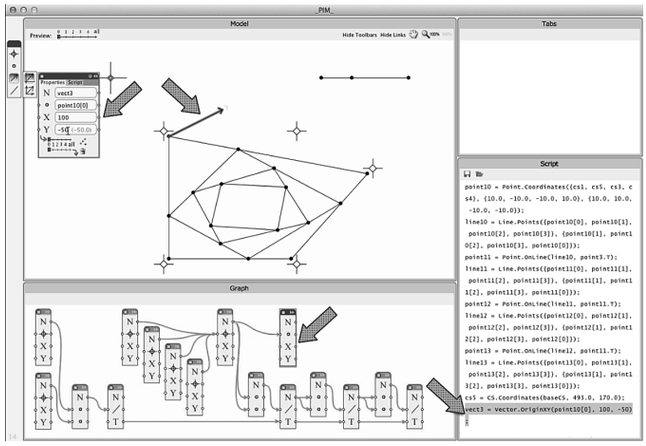
If the model is blocked and does not update during scripting, designers have to work *"blindly"* in the script without seeing the design in the model until they compile (or run) the script. One of the most important features of PIM is to give immediate feedback to designers. When users edit an object in the script window, the change is immediately reflected in the model and the graph, so that they can see and evaluate the effect of that action on the model right away without having to leave the script window (Figure 4). Having access to the model and graph during scripting also means that users do not have to remember the name of the object that they want to refer to in the script, but can simply point to it in the model or graph and its name is inserted in the script where needed.

AUTO-TRANSLATION OF MODELING ACTIONS INTO SCRIPT

Immediate feedback goes both ways. To create an object, designers can use the toolbars in the model view, and the action is immediately reflected in the script (Figure 4). For designers who are not yet familiar with scripting, the real-time script generation offers a learning opportunity: by following the generated script, they can learn the syntax for the action they just performed. These features affect how they write functions or any block of code in PIM. Instead of writing the function in an abstract mode in the script window without seeing what it does until later when



3 PIM's model, graph, and script windows are live and update automatically with brushing and highlighting for navigation.

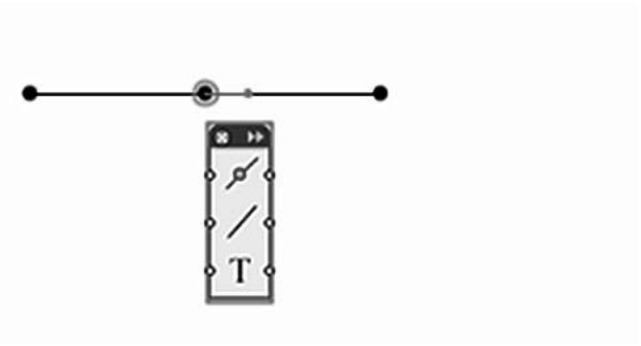


4 Liveness in PIM means that any action in any window is immediately reflected in the other two.

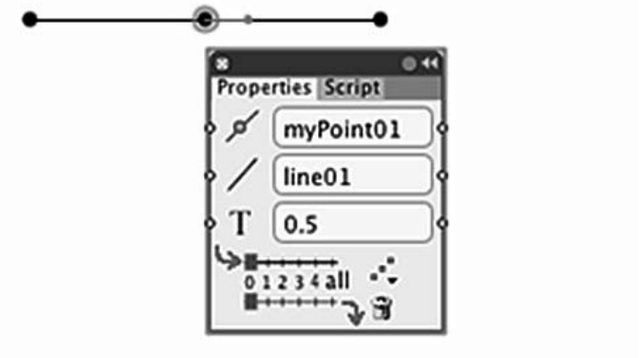
they make a function call, designers can continue modeling as usual and PIM will wrap the work into a function. At any time during this process designers can choose to work in the script, the model or the graph, and the other two representations are updated in realtime.

LOCALIZED INFORMATION WITHIN THE MODEL WINDOW

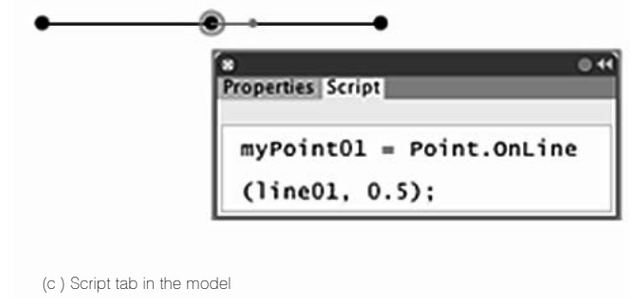
These features help designers when it is necessary to work in the script, but switching back and forth between these windows creates a cognitive load for designers and takes focus away from the model where the design work largely takes place. PIM's approach is to give users access to all the information about the object in the model view. For each object, there exists an expandable edit toolbar that presents different types of data about it, including inputs, replication (Aish and Woodbury 2005) and an editable copy of the script that creates that object (Figure 5). Thus, users have



(a) Edit toolbar



(b) Expanded edit toolbar



(c) Script tab in the model

5 Localized information appears in the model next to the object and includes name, type, inputs, script, and dependencies.

the option to perform as much of the task as possible in any one of these representations.

In most CAD systems, to explore the relationships between an object and the rest of the model, designers have to highlight the object and find it in the graph, follow the links and find what object(s) are upstream or downstream and then highlight and find those objects back in the model. As a result, there is a lot of switching between windows to figure out the dependencies. PIM gives designers the option to see upstream and downstream dependencies in the model view. These links directly connect model objects, not the nodes that represent those objects in the graph, so it is easy to find out what object is upstream or downstream of an object in the model (Figure 6). The graph window still exists and helps to get a sense of the whole dependency structure or to do more complex tasks.

4 EVALUATION

We conducted a qualitative user study to evaluate Programming In the Model. The participants came from the field of architectural design in academia and industry. All twelve participants had experience with parametric modeling and expertise in scripting, ranging from novice to expert. We used the prototype and a video demo to demonstrate PIM features to the participants. They used the prototype to perform a number of modeling and scripting tasks. The tasks and the discussions were recorded and analyzed, using an open coding method. We do not describe the study and analysis in detail here due to space limitations, but briefly present some of our findings.

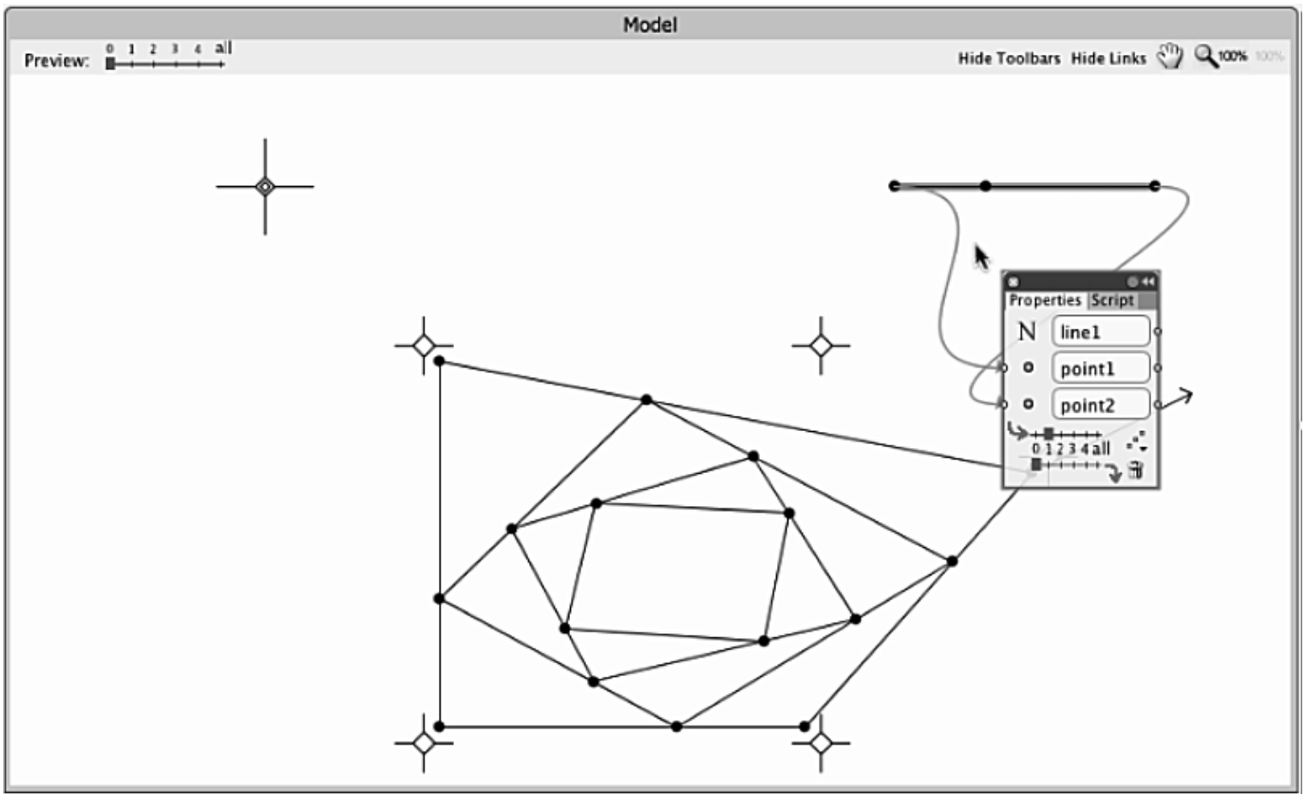
As we expected, the live script window was well received by the participants. They referred to the conventional scripting environment as a black box that is unknown to designers with a language close to “nuclear physics” that is hard to translate into a familiar language. The fact that the script window is always open in PIM and dynamically updates with every modeling action reduces the mystery behind the scripting language and shows how code and modeling relate. Participants thus confirmed our vision of live programming and wanted even more.

They expect the interface to be as live as it can possibly be and reflect the smallest modeling actions in the script and vice versa. For example, it is not enough to show the syntax that creates a vector after users are finished making the vector in the model. They need to see all the steps of creating a vector translated into code, such as initiating the vector, naming, providing inputs, and so on. Such liveness must work in all directions. For instance, creating a vector in the script produces a visual vector as soon as sufficient information is available. We coin the term “extreme liveness” as both a design goal and a descriptor of such interfaces, which is crucial in making effective mental connections between the code and the model.

Another barrier that resonated with the participants was navigating through the information presented in the script and graph and connecting it with the model that they were designing. Highlighting was used widely by the participants to search and navigate within each window and also across different windows, especially to locate the object of interest in the script when it was known to them in the model. They also used highlighting to make sense of the script by quickly hovering the mouse over the syntax and finding the object that it represented in the model. This action immediately turned the abstract code into a concrete object that was more meaningful in the context of the modeling task.

Localization of information (including code and dependencies) in the model received mixed feedback from the participants. Some found the script accompanying the object in the model extremely useful by sorting the data in the two-dimensional and three-dimensional spaces. They preferred to use this version of the script over the script window to edit individual objects. They stated that the script tab in the model further reduces the "distance" between the model and the code. Other participants did not notice any value in the script tab. The idea of localized dependencies received positive feedback across the board, but the representation of the dependency data failed to achieve the desired effect. We chose a node-link representation with directed links connecting the upstream objects to downstream objects in the model. During the tasks, participants found it hard to follow the links and find them within the 2D model and most of them failed to identify one or more dependencies. We received several suggestions from the participants for improving or changing the way we represent dependencies in the model. These need to be investigated in the future.

Several participants raised the issue of complexity and asked whether PIM would be able to handle complex models. The problem has three sides. The first is performance: can our machines handle live coding with complex three-dimensional models? The answer is irrelevant. They may or may not be able to handle it now, but soon they will be, as they are improving everyday. That should not stop us from designing better interfaces and eliminating barriers. The technology will catch up. The second part of the question of complexity is in regards to data visualization and interaction: is it possible to represent all of this data over a complex three-dimensional model without compromising clarity of the model or the data? The issue of scaling is endemic in visual programming languages: they do not scale up well. We were aware of this challenge when we started this project and we are constantly looking for better and more scalable ways to represent data in PIM. Our strategy of localization is a partial solution:



6 Localized representation of dependencies in the model view in PIM.

by having only the current data of interest displayed within the model, we reduce immediate visual complexity, though at a cost of potentially losing larger programming context. The third aspect is abstraction, for which textual programming has established and well-understood conventions. In contrast, visual programming interfaces generally have much weaker and less developed tools for working with abstraction. By making links between modeling and programming more direct, for instance with edit toolbars in the model, PIM attempts to better connect the concrete world of modeling with the necessarily abstract world of programming. These are only early and partial solutions: complexity will be a challenge for PIM as it is for all visual programming systems.

5 DISCUSSION AND CONCLUSION

Multi-view, multi-level interaction is well-established in CAD (Aish and Woodbury 2005). Other fields point out directions for improvement. The notion of live coding is beginning to find its way into programming applications (Live Programming Workshop 2013; Burnett, Atwood Jr, and Welch 1998); see, for instance, Khan Academy's (2012) programming environment based on processing for teaching programming with graphics, and the Experimental Media Research Group's (2013) visual programming system NodeBox. Both environments have side-by-side windows, one for the code and one for the graphical output. The liveness of the environments means that the result of every line of code is immediately rendered in the output window without the need for users to run or compile the code. Autodesk's DesignScript employs the same principle with the resulting model appearing in the AutoCAD window. In all of these systems, the liveness is uni-directional (code to model), whereas PIM offers a multi-directional liveness between code, model and graph. Any action in any window is immediately reflected in the other windows. By having extreme liveness in all directions we support users in transfering their modeling knowledge and skills to programming.

Victor (2012) takes this concept further and suggests that liveness alone is not enough. He believes that designers need complete transparency in the code, from what everything means, to the state of variables and the flow of the program. They also need to think in concrete terms, with something that is specific and completely understood, before it is generalized into functions and classes. We use his advice in PIM and make the system fully transparent by highlighting everything across notations. That means lines of code, arguments and variables are highlighted in any notation they appear in when users hovers the mouse over their representation in any other windows.

PIM goes even further, in being able to overlay fragments of another representation on the representation being edited, for instance, graph or script fragments localized in the model. Such overlays mean that a designer can *compose* the interface specifically to the task at hand. Perhaps the strongest result from the user study is that participants both confirmed PIM's liveness and wanted to push it further still. It seems then that both the literature and our own findings identify a sound direction for future research towards extremely live, multi-representation and highly composable interfaces for parametric modeling.

When we talk about programming or scripting in CAD, the notation may be a textual syntax such as VBA in Solidworks or GCScript in GenerativeComponents, or a visual programming notation such as Rhino's Grasshopper. Visual programming has had some success in breaking programming barriers, particularly because of how its visual notation *speaks* to CAD users (mostly architects and engineers) with strong visual and spatial skills. It also allows a more direct manipulation of the program (Burnett 1999). Although visual programming appears to be easier to use and understand for designers, it suffers from some of the same problems of textual programming, including the different notation it uses (node-link) compared to the modeling notation, its lack of direct manipulation of the model (direct manipulation only happens on the nodes and links) and low level of liveness (the delay between user's action and its effect in the model.) PIM principles are applicable to textual and visual programming alike.

ACKNOWLEDGEMENTS

This research is partially supported by the Canadian NSERC Postgraduate Scholarships, Discovery Grant and Collaborative Research and Development programs; Bentley Systems Inc. and the Graphics, Animation and New Media (GRAND) Network of Centres of Excellence.

WORKS CITED

Aish, Robert. 2003. "Extensible Computational Design Tools for Exploratory Architecture." In *Architecture in the Digital Age : Design and Manufacturing*. New York, NY: Spon Press.

Aish, Robert, and Robert Woodbury. 2005. "Multi-level Interaction in Parametric Design." In *SmartGraphics*, 5th Intl. Symp., SG2005, ed. A. Butz, B. Fisher, A. Krüger, and P. Oliver, 151–162. LNCS 3638. Frauenwörth Cloister, Germany: Springer.

Blackwell, A. F. 2002. "First Steps in Programming: a Rationale for Attention Investment Models." In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, 2–10. Arlington, VA, USA.

Burnett, Margaret M, John W Atwood Jr, and Zachary T Welch. 1998. "Implementing Level 4 Liveness in Declarative Visual Programming Languages." In *1998 IEEE Symposium on Visual Languages*, 126–133.

Burnett, Margaret M. 1999. "Visual Programming." In *Encyclopedia of Electrical and Electronics Engineering*. New York: John Wlley & Sons Inc.

Burry, Mark. 1997. "Narrowing the Gap Between CAAD and Computer Programming: A Re-Examination of the Relationship Between Architects as Computer-Based Designers and Software Engineers, Authors of the CAAD Environment." In *The Second Conference on Computer Aided Architectural Design Research in Asia*, 491–498. Taiwan.

Dertouzos, Michael. 1992. ISAT Summer Study: Gentle Slope Systems; Making Computers Easier to Use.

Experimental Media Research Group. 2013. NodeBox. Sint Lucas School of Arts of the Karel de Grote-Hogeschool, Antwerpen (Belgium). http://www.nodebox.net.

Gantt, Michelle, and Bonnie A. Nardi. 1992. "Gardeners and Gurus: Patterns of Cooperation Among CAD Users." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 107–117. CHI '92. Monterey, California, United States: ACM.

Hutchins, Edwin L., James D. Hollan, and Donald A. Norman. 1985. "Direct Manipulation Interfaces." Human-Computer Interaction 1 (4): 311.

KhanAcademy. 2012. Khan Academy Computer Science. http://www.khanacademy.org/cs/new.

Ko, Andrew J., Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, et al. 2011. "The State of the Art in End-user Software Engineering." ACM Comput. Surv. 43 (3) (April): 21:1–21:44.

Live Programming Workshop. 2013. A History of Live Programming. http://liveprogramming.github.io/liveblog/2013/01/13/a-history-of-live-programming.html.

Myers, Brad A., David Canfield Smith, and Bruce Horn. 1992. "Report of the End-User Programming Working Group." In *Languages for Developing User Interfaces*, ed. Brad A. Myers, 343–366. Natick, MA, USA: A. K. Peters, Ltd.

Nardi, Bonnie A. 1993. A Small Matter of Programming: Perspectives on End User Computing. Cambridge, MA: MIT Press.

Shneiderman, B. 1983. "Direct Manipulation: A Step Beyond Programming Languages." Computer 16 (8): 57–69.

Streich, Bernd. 1992. "Should We Integrate Programming Knowledge into the Architect's CAAD-Education? Basic Considerations and Experiences from Kaiserslautern." In Proceedings of the 1992 eCAADe Conference: CAAD Instruction: The New Teaching of an Architect, 399–406. Barcelona.

Victor, Bret. 2012. Learnable Programming. http://worrydream.com/#!/LearnableProgramming.

Woodbury, Robert F. 2010. Elements of Parametric Design. Taylor and Francis.

MARYAM MALEKI holds a Master of Science in Architecture from Shahid Beheshti University and is currently a Ph.D. candidate at School of Interactive Arts and Technology (SIAT), Simon Fraser University, Canada. She was awarded a three-year postgraduate scholarship from the National Science and Engineering Research Council of Canada (NSERC) for her work on end-user programming in CAD. She also received the Young CAADRIA Award in 2010.

ROBERT WOODBURY holds a BArch from Carleton (Silver Medal) and MSc and PhD from CMU. He was appointed in Architecture and the Engineering Design Research Center at CMU from 1982-1993, at Adelaide from 1993-2001 and is now at Simon Fraser. He was founding Chair of the Graduate Program in the School of Interactive Arts and Technology at SFU and of the Canadian Design Research Network. He is Director, Art and Design, of the Graphics, Animation and New Media Canada Network of Centres of Excellence. He has been awarded the ACADIA Award for Innovative Research and the CAADRIA Tee Sasada Award. He is a former Olympian in sailing.