Liveness, Localization and Lookahead: Interaction elements for parametric design

Maryam M. Maleki
Simon Fraser University
Surrey, BC, Canada

Robert F. Woodbury
Simon Fraser University
Surrey, BC, Canada

Carman Neustaedter
Simon Fraser University
Surrey, BC, Canada

ABSTRACT

Scripting has become an integral part of design work in Computer-Aided Design (CAD), especially with parametric systems. Designers who script face a very steep learning and use curve due to the new (to them) script notation and the loss of direct manipulation of the model. Programming In the Model (PIM) is a prototype parametric CAD system with a live interface with side-by-side model and script windows; real-time updating of the script and the model; on-demand dependency, object and script representations in the model; and operation preview (lookahead). These features aim to break the steep learning and use curve of scripting into small steps and to bring programming and modeling tasks 'closer together.' A qualitative user study with domain experts shows the importance of multi-directional live scripting and script localization within the model. Other PIM features show promise but require additional design work to create a better user experience.

Author Keywords

Qualitative methods; Creativity support tools; User studies; End-user programming; Computational design; Computer-aided design; Parametric Design.

ACM Classification Keywords

H.5.2 User Interfaces, Graphical user interfaces
J.6 Computer-Aided Engineering, Computer-aided design:
D.2.6 Programming Environments, Interactive Environments

INTRODUCTION

Architects and engineers use Computer-Aided Design (CAD) systems to design and represent buildings and products. Typical CAD systems present a two or three dimensional model of the design to the users and allow them to directly manipulate objects in the design using the graphical user interface (GUI). Sometimes though, designers reach the limits of the CAD GUI, which they can only overcome by writing programs or scripts in order to have the freedom to explore unconventional and complex forms [1]. This is especially true in parametric CAD systems when designers design by manipulating the underlying data structure of the geometric model and by defining relationships among geometric components [24]. In recent years, parametric systems have become widespread in CAD practice, yet remain difficult to use and learn. Indeed, several recent books focus on enabling more effective use of such systems [24, 12] and there are international workshops on fostering advanced communities of practice [18].

These domain experts who write programs become *end-user programmers* [7]. They face a challenge when they move away from the CAD 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 to hand [15, 24]. Because programming is just another tool for them, they weigh the perceived cost and risk of learning and using programming against the benefit that it brings to their work. In the attention investment model [4], if a system looks too hard and the attention cost of learning and using it seems too high, end-users will balk, which means they will not benefit from using it in their work. Further, end-user programmers often interpret barriers to learning and use as insurmountable [14] and have what has been described in the CAD domain as 'fear of code' [19]. Note well the two terms 'learning' and 'use': both apply. A person learning a system aims to discover how to achieve tasks in the system's notation. A person using a system aims to have the appropriate notation available with minimal distraction from the task to hand. Attention investment and barriers apply to both learning and use.

The goal of our research is to reduce the perceived cost/benefit ratio and barriers to programming in CAD for designers. Our main strategy, which we call *Programming in the Model (PIM)* is to break down the learning and use process into small, accessible and reorderable steps; and situate these within or relate them to the CAD GUI. At the outset, we did not know, nor did the literature provide, features that might meet these goals. Thus we undertook an iterative design and prototyping process in which we had regular reviews with experts in parametric CAD. The result is the PIM system, which has features such as a live interface with side-by-side model and script windows with fine-grained realtime updating, fragments of the script and dependency representation locally available in the model, and user-customizable preview (lookahead into the future model) with user actions. We imagine these techniques to be applied to existing CAD systems to break the steep learning and use curve of scripting into small steps and to bring programming and modeling tasks 'closer together.' By doing so we hope to reduce the fear of code in designers and make programming a tool in their design process.

Within this design space, we have not found any studies in the literature that show evaluations of such features to understand user reactions to them. To address this, we conducted a qualitative study of the PIM system. The goal of the study was to assess different programming in the model features to understand how users interact with them, which features create pos-

itive experiences, which features are challenging to use, how features might be improved, and, overall, which features may benefit other fully functional CAD systems. Our results illustrate the importance of multi-directional live scripting where users are able to easily move between scripting and working with the model in a fast and efficient manner. We also learned that localizing fragments of one view in others is valuable, but requires careful design to ensure good user experience. Lastly we learned that preview has real potential as an error prevention and debugging tool. The implication is that commercially-available CAD systems should strongly consider incorporating immediate liveness, if not what we term 'extreme liveness,' into their interfaces. When it comes to other features, designers should consider preview and localization features that build on our study results.

In what follows, we briefly present background research and related work and explain Programming In the Model. Next, we explain the results of the study in more detail and discuss design implications for CAD systems.

BACKGROUND AND RELATED WORK

According to Blackwell [4], loss of direct manipulation and introduction of a new notation are two main features of programming tasks. In CAD, direct manipulation is the primary method of interaction with the model. 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. Modelling notation consists of line, curve, surface and solid components, move, rotate, copy and scale operations, and dimensions and materials. The scripting language has a very different notation that may include functions and loops and conditionals, classes and instances, arguments, variables and types, and commas, semicolons, and brackets.

During a modelling task, the user interacts with the model to create and modify objects and their relationships with other objects. During programming however, they have to focus their attention on a new window and interact with programming elements instead of the model that is the object of their design. As a result, they lose the benefits of direct manipulation including immediate visual feedback to their actions [20] and the sense of directness between their thoughts and the actions of the system [11].

Dertouzos [10] and later Myers [17] introduce the concept of a *gentle slope system.* In such systems, to do a customization, users only needs to learn a small number of features. In other words, they must climb only a small step to move forward. Some systems require a huge amount of learning before users are able to get a task done. Often, they hit a wall (or barrier) that they need to surmount before they can continue (See Figure 1). For example, spreadsheets are relatively easy to use, up until the point when the users open VBA (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. Ko and Myers [14] argue that such barriers can effectively be insurmountable and present six barrier types: design, selection, coordination, use, understanding and information.

Attention investment, gentle-slope systems and barriers are useful analytic ideas, but do not provide guidance for design. For this we turn to exemplary systems. Multi-view, multi-level interaction is an established principle in parametric CAD [3], with a focus more on how each view contributes to design, rather than on the impacts of having multiple views. In contrast, some system developers argue that designers need immediate connection to what they are creating [22], and programming environments can facilitate or take away that connection. According to Victor [23], *immediate visual feedback* to programming actions shows their effect on the design and letts the designers react to them right away. He also recommends starting with something concrete that is completely understood before generalizing it, and making everything transparent, from the flow of the program to the state of variables to make a more *learnable programming* environment for designers.

Khan Academy has recently launched a programming interface for beginners that has a one-way liveness from code to graphics [13]. A line of code is immediately rendered when it is completed and sliders in the coding interface are used to change numeric values, with immediate results displayed visually on the final design. Autodesk's DesignScript [2] has also implemented similar one-way liveness from the script to the CAD model.

Liveness, when implemented, is largely one-way, typically from the more to the less symbolic interface. Further, we do not know of any studies that evaluate liveness in CAD systems. An example of a two-way live interface is Adobe Dreamweaver, in which the screen can be split between code and result, and visual elements of the webpage can be modified in both the HTML code and in the visual representation of the page. However, more advanced edits such as Javascript code do not work in this way.

Visual programming represents a nearly orthogonal approach to our learning and use problems. It has had some success in breaking programming barriers in CAD (e.g. Rhino's Grasshopper), particularly because of how its visual notation 'speaks' to CAD users who are mostly architects and engineers with strong visual and spatial skills [5]. It also allows a more direct manipulation of the program [6]. Although visual programming appears to be easier to use and understand for designers, it suffers from some of the same problems as textual programming, including the different notation it uses (node-link) compared to the modeling notation and its lack of direct manipulation of the model (direct manipulation only happens on the nodes and links.) In addition, visual programming implementations typically do not give designers all the capabilities of scripting and thus limit design exploration [8].

Terry's *side views* [21] present a preview of commands that enables users to experiment without modifying the original data and compare and contrast multiple states of the data during alternative generation. He explores the effectiveness of

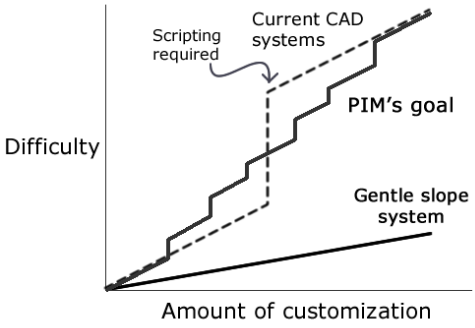


Figure 1: Gentle slope systems

side views in text and image manipulation interfaces. While valuable, more generalized versions of preview tools in CAD scripting interfaces are largely absent in the field.

Together, the related work and our initial design studies (see below) provide a backdrop for understanding why we have designed particular features into PIM. It also illustrates how some features that we have incorporated within PIM are beginning to appear in other systems, albeit in a limited fashion. Most importantly, what we do not see in the literature is a detailed study that explores the variety of interaction techniques that we have included in PIM, which have the potential to reduce the steep learning and use curve found with many CAD systems. Thus, our study builds on the related work to provide a detailed user evaluation of a set of PIM features along with an understanding of how these should best be integrated within existing CAD systems.

PROGRAMMING IN THE MODEL

In this section we first discuss the goals and considerations behind Programming In the Model, then introduce its features in the context of a prototype we call PIM.

Goals and design considerations

The goal of Programming in The Model (PIM) is to create a more gentle slope system by breaking the barrier between modelling and scripting into small steps and by situating these within or relating them to the CAD GUI . It is important to note that PIM is an interaction strategy, not a new programming language. PIM offers ways to improve the scripting environments of existing CAD systems in relation to their modeling interface. PIM's goal is not to reach the perfect shallow curve in Figure 1, labeled as *Gentle Slope System.* We accept the level of difficulty of these scripting languages and only claim to break the difficulty curve into smaller steps to make it easier for end-user programmers to learn and use these languages (Figure 1). (For the same reason, our design for PIM does not extend to the choice of programming language and common IDE support tools, including syntax-directed editing, syntax error prevention, etc.)

Additionally, we hope to lower the perceived cost/benefit ratio of scripting in PIM compared to traditional scripting languages, 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.

We developed PIM through an iterative design–prototyping–evaluation process, starting in 2009. Early prototypes were done within existing parametric tools and demonstrated at industry workshops such as Smartgeometry and to local domain experts. These convinced us of a strategy that includes the textual script as a representation equal to others (e.g., model and dependency graph), rather than, for instance, building a new visual metaphor for scripting. We heard, again and again, that the script is the most powerful view to which people would ultimately turn to solve hard problems and through which people would advance to full programming environments such as Java or C#. Through these prototypes, we developed the following design considerations for the PIM environment: (a) supporting direct manipulation of the model during programming; (b) supporting users' transition from modeling notation to programming by strongly relating modeling and coding acts; (c) enabling fine-grained and incremental programming; and (d) predicting the effects of changes before committing to them. Maleki et. al. [16] published a full description of PIM system features. The focus of this paper is on presenting its evaluation.

PIM features

PIM is a 2-dimensional parametric CAD prototype with a limited number of geometric objects (coordinate systems, points, lines, and vectors). PIM's interface consists of a modeling window that shows the 2D model, a graph window that displays the parametric structure of the model in a node-link diagram, a script window, and a number of tabs that house non-geometric components of the design. The goal of creating the prototype was not to make a new, independent CAD system, but to have a platform to implement and demonstrate our ideas in a realistic setting. We implemented a subset of all desired PIM features in the working prototype. We produced a video demo of PIM, which along with the prototype, presents a complete picture of PIM features for the user study (see attached video figure).

Unlike most CAD systems that block access to the model during scripting, PIM keeps active all representations of the design side-by-side, including the model view, the dependency graph, and the script window. Further, aspects of one view can be shown and edited directly in others. All views are concurrent and interactive, so the designer can equally access the model and the graph during scripting. To help navigation through these representations, PIM offers ***brushing and highlighting*** of the data in the script, graph and model (Figure 2). Holding the mouse over a line of script highlights the text, the object(s) the line describes in the model, and the node(s) representing them in the graph. The live interface of PIM gives the designers ***immediate feedback*** for their actions in the script window by updating the model and graph concurrently with the script. Comparing to the blind working conditions of traditional CAD scripting environments, realtime of updating of the model allows the designers to see and evaluate the effect of that action on the model right away without having to leave the script window.

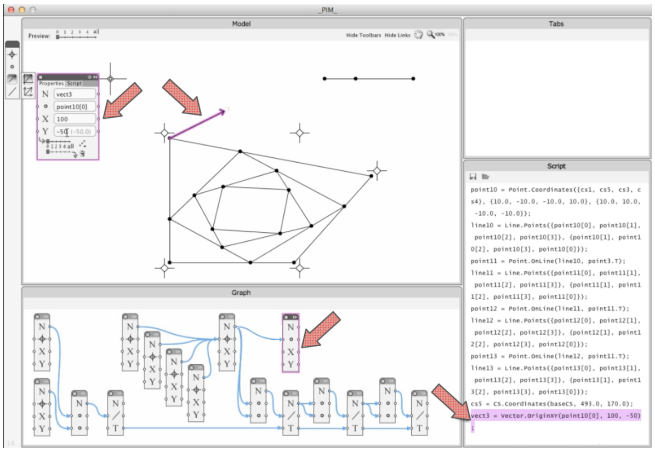
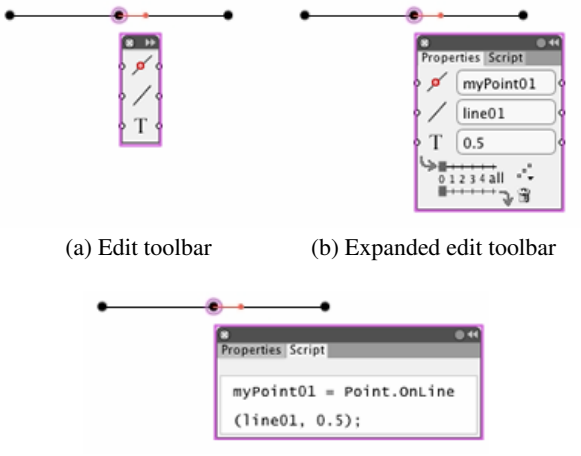


Figure 2: PIM is a live interface with all representations reflecting the action in realtime. Brushing and highlighting brings the object into focus in all windows. Edits in any window are immediately reflected in all others.

Immediate feedback in PIM goes both ways (Figure 2). Actions performed by direct manipulation of the model, using the CAD GUI, are reflected in the script concurrently. The ***real-time script generation*** means that (1) users can learn the syntax by observing the script that the system generates, and (2) they can create code by modeling and the system will translate their actions into code. The model-to-code liveness can be confused with macros (as was by a study participant), but there is a major difference. When a novice creates a macro, the goal is not to do or learn programming. It is to automate a repetitive task. If the user attempts to edit the macro code, (s)he faces a piece of program in an unfamiliar language with almost no legible mapping between the syntax and the domain (for example Visual Basic for Applications code and a Solidworks model). PIM's bidirectional liveness not only generates the code from the user's modeling action (as done in macros), but it presents this translation from one notation to another to the user immediately. This changes the way we write ***functions*** or any block of code in CAD. At any time during this process, we can choose to work in the script, the model or the graph and the other two representations are updated in realtime.

Now these features help the designer when it is necessary to work in the script. But switching back and forth between these windows creates cognitive load for the designer and takes away the focus from the model, where the design is taking place. PIM's approach is to ***localize access,*** that is, ***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 [3], and an editable copy of the script that creates that object (Figure 3). A user has the option to perform as much of the task as possible in any one these representations.

In most CAD systems, in order to explore the relationships between an object and the rest of the model, the designer has to highlight the object and find it in the graph, follow the



(a) Edit toolbar (b) Expanded edit toolbar

(c) Script tab in the model

Figure 3: Localized information about an object appears next to it in the model. This information may convey name and type of the object, inputs, corresponding scripts, and dependencies.

links and find what object(s) are upstream or downstream, then highlight and find those objects back in the model. PIM gives the designer the option to view these ***dependencies in the model view*** (Figure 4). 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. The graph window still exists and helps to get a sense of the whole dependency structure or to do more complex tasks.

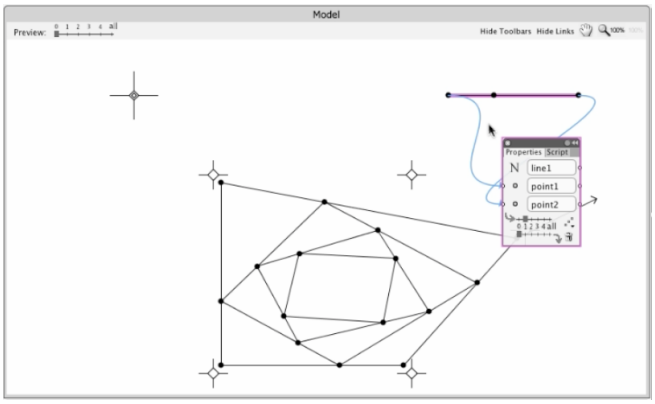


Figure 4: Dependencies in the model directly connect objects with their upstream and downstream objects.

In parametric modeling, sometimes the change we want to make on an object must be applied several levels up in the dependency tree, which makes it difficult to decide the action that would produce the desired effect downstream. PIM offers a feature called ***preview*** that, if turned on, gives the user a lookahead into the state of the model, especially the objects that are downstream of the object that is being changed (Figure 5). The user can choose to see as many levels downstream

as needed, evaluate the model and only confirm the edit when the intended effect is achieved.

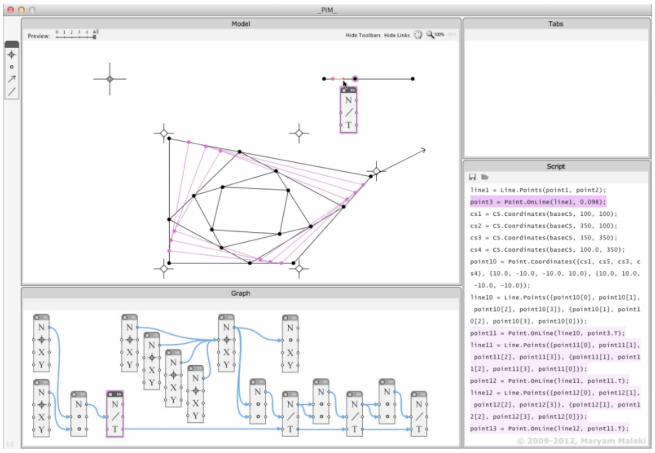


Figure 5: Preview in PIM displays the edit alongside the original model and highlights all the objects that are affected by the change.

USER STUDY

In order to evaluate and improve PIM's design, we conducted a qualitative user study, with both HCI and domain experts as participants. The goal of the study was to qualitatively assess different PIM features to understand how users interact with them, which features create positive experiences, which features are challenging to use, and, overall, which features may benefit fully functional CAD systems.

Participants

All participants including the HCI experts had some experience with parametric CAD systems. They were chosen from CAD users in academia and practice. They all had a design background; had used a CAD graphical user interface as part of their work; had some experience with a parametric CAD system such as Solidworks, Grasshopper, Revit, Cinema 4D, and Generative Components; and had experience with those systems' scripting interfaces. This was important for it meant that our participants could compare the features within PIM to the standard practices of existing CAD tools, as well as imagine how the PIM features may work in such tools if available. Thus, our participant selection provided a means for us to receive valuable critiques of our design choices and directions for future development. We recruited participants until we reached data saturation: that is when we were not observing or hearing anything new. There were 8 male and 4 female participants, all adults over the age of 18. They were recruited through direct email, social media such as LinkedIn and Facebook, and verbal requests. Participants received a small reward (in the form of a gift card) for their time.

Procedure

Before each session, participants read and signed an informed consent form and filled in a pre-session questionnaire about the their age, gender, and prior experience with parametric CAD and computer programming. Each session had four parts.

1. In the first part, the researcher gave a quick demo of PIM and introduced its basic functionalities to the participant. Then the participant took over and performed small tasks designed to familiarize them with the new interface.

2. and 3. In the second and third parts, the researcher gave demos of more advanced features of PIM, including scripting and dependencies, and the participant performed four tasks in each part that used those features. A short video demo was shown at the end of Parts 2 and 3 that covered future PIM features not yet implemented in the prototype. The tasks were similar to common parametric modeling tasks and were chosen so that the user had a chance to examine all available features of the system, but were simplified to accommodate the study time and PIM's current limitations. Tasks involved creating new geometric objects, editing existing objects, and identifying and manipulating relationship between objects, all in the model window, in the script window, or both. The following two tasks are representative of the broader tasks set:

Part 2, Task 1: *"Identify the left most coordinate system in the model, find its syntax in the script window, and change its Y input to 50 in the script."*

Part 3, Task 3: *"Using the dependency links, can you tell what objects will be affected if you edit the vector called vect01, without actually editing it?"*

During Parts 1 to 3, participants were asked to think aloud and tell us what they were doing, why they were doing it, and difficulties and uncertainties they encountered. We encouraged participants to ask questions and told them that they were not being tested for their skills or speed with which they learned the features and performed the tasks.

4. After Parts 1 to 3, participants engaged in an open-ended interview with the researcher about PIM features in general and in relation to their work. The researcher asked questions about different PIM features and discussed any unusual or interesting event that happened during the tasks. Throughout the study, our focus was on PIM features and how they affected the participants' experience with the system as opposed to common usability concerns.

Overall, each study session took about two hours, with a total of 10 minutes for introduction and background questionnaire, 20 minutes in total for demos (live and video) split in three parts, 60 minutes for tasks split in three parts, and 30 minutes for a closing interview.

Study method rationale

The target users of CAD scripting environments are typically intermediate and expert users of the system who want to extend the capabilities of the system and customize it through scripting [9]. Because PIM is a new system, there are no

users who are experts with it. As a result, all of our participants started out as PIM novices. In order to scaffold participants in to users that could be considered intermediate or expert users of the system, more in line with typical users of CAD systems, we gave them short demonstrations of the functionality in the PIM system. We divided the demonstration into three separate parts followed by relevant tasks, in order to keep the users engaged and avoid losing their interest and attention, which might be the case with a single long demonstration. An alternative approach of letting users build up their expertise and understanding on their own without demonstrations would have been largely impractical in a study situation; building up one's expertise in a system like PIM could easily take days or weeks.

Our study was purposely qualitative and observational in nature because we wanted to explore how users interacted with features within PIM and what they thought of the features. We were interested (dis)confirming the features and their relative perceived importance in order to refine and develop PIM. Thus, we were not interested in task performance, completion times, and counts of user errors that one might quantitatively assess. Instead, we wanted detailed observations and thick descriptions that come from an exploratory qualitative study where we were able to observe users' reactions to PIM features, as well as ask HCI experts to evaluate the usability of these ideas. We also purposefully did not conduct a comparative study with PIM against other CAD systems. PIM is a prototype system and not a fully functional CAD system; thus, such a comparison study would have generated an 'apples to oranges' comparison.

Data collection and analysis

The sessions were voice recorded with participants' permission. During the tasks they performed with the prototype, the computer screen was recorded using software for future analysis of users' actions. The researcher(s) observed the sessions and took notes. The data that we collected in the study included a written pre-task background questionnaire, audio (voice) / video (screen capture) files of the demo/task, an audio file of the post-task open ended interview (discussion). We used NVivo10 software to sort, transcribe, and analyze the data. In the first round of data analysis, we used open, axial, and selective coding from grounded theory. With no predefined codes, we went over the data and let it speak to us. Categorizing codes were quickly formed, including participant classifications, nature of the data (statement or event), type of data (positive, neutral, negative), and PIM feature(s) to which it relates.

We tagged all statements spoken by the participants and every event we observed during the tasks as positive, negative or neutral. A *positive* tag indicates a statement that showed a positive emotion toward a PIM feature, for example *"It is very helpful that I can see the object highlighted in the model."*. A *negative* tag indicates an event that showed a problem or challenge, for example when the participant failed to identify the right object in the model, or a statement with negative emotion such as *"I am not so sure that preview will work well in the large models that I work with."* Events and statements that were neither positive nor negative were tagged as neutral.

RESULTS

After studying the codes that had emerged in the analysis, we grouped them into higher level themes. These themes describe the reactions that participants had to PIM's design features (both positive and negative). We explain them in detail under the headings of *search-navigation*, *learning*, *fear of code*, *mental model*, *errors*, *trial and error*, and *complexity*.

Search-navigation

One of the barriers 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. Brushing and highlighting was used widely by the participants to search and navigate within each window and also across multiple windows, especially to locate the object of interest in the script when it was known to them in the model. Participants acknowledged the benefits of instantaneous brushing between the model, the graph, and especially the script window, something that they had not seen in any other CAD system. Of course, highlighting works because of the liveness of the interface. One participant mentioned her past experience, trying to find things:

"It is an interesting search method, comparing to what I use in GenerativeComponents in a huge model. To find something, I had to search and type it in and find it. But here I can just go over it and it highlights. It makes it easy." (P2)

Sometimes the thing that needed to be found was not an object, but a relationship. For example, the user wanted to know which coordinate system a particular object was in. Displaying dependencies in the model helped the users find that information easily without having to leave the model and go to the graph, follow the relationships, find the coordinate system, then find it back in the model. One participant explained:

"[Displaying dependencies in the model] is useful for sure. Just because you don't have to look from [model] object to node, [node] to node, back to object. It's direct." (P6)

As one might expect, dependencies were not always understandable by users. At times links showing the dependencies were obscured by model objects. In these cases, participants did not see the links and missed the connections between objects. However, having parts of the script that related to an object available in the model window was found to be another way of navigating the massive amount of data in a parametric CAD interface.

"It is interesting how code is tagged to objects. ... And it is sorted in the 3D space by the virtue of being tagged to [model] objects." (P6)

Learning

Participants found scripting in PIM was easier to learn than in other CAD systems. This was evident to them and to us after the first few scripting tasks that they performed using the prototype. Some participants did not even wait for us to show them how to use the system. Instead, they intuitively started

playing around with the script. By looking at the data we can see that the feature that related the most to learning was the liveness, especially the model to script liveness (or auto script generation). One participant observed how the code reflected the action of creating a vector in the model:

"The name of the object is what immediately gets my attention, because it's been repeated in the code. ... Then the argument point10[0] is repeated in the script. That's perfect that it shows up in both places." (P3)

He then emphasized the importance of *extreme liveness* or liveness to the smallest action possible in how the users connected the two representations:

"I think if they type everything in the edit toolbar and enter and all of a sudden a line of code appears with all the arguments, that short lapse of time is enough for them to lose the connection. But the fact that they are inputting the values and the window is still open and the inputs update in the script, makes a huge difference. So If you don't see the line of code when you are entering the inputs, then you close the window, you are not even thinking about it anymore. You move on to the next task. But here you are in the middle of the task and you are seeing the code in realtime, it gives you the opportunity to consider what you are doing in code form." (P3)

Participants also commented on how they learned from examples and how, in PIM, they were able to learn from the examples of their own modeling work. One participant said that the liveness of PIM is revealing, in the sense that it shows the users that what they do during modeling is not much different from what they would do during scripting.

"They start to get a sense of functions and relationships... just by manipulating geometry the way they already are. By having [the script] construct itself while they are modeling, they realize that those are the relationships they are making, but they never think of it that way. This is revealing." (P6)

A challenge that PIM did not manage to overcome was one particular participant who even after performing the tasks and working with the system still believed that scripting was not a necessary tool in design. In fact, he was opposed to it as a design tool. This shows that PIM is only beneficial for users who are willing to start programming in the first place.

Fear of code

Most of the participants stated that PIM's live scripting was would help reduce the general fear of code among designers. They referred to the traditional script window as *'black box'* and the code as *'nuclear physics'* to describe how unfamiliar and scary it seemed to non-programmers.

"If they always see the script updated as they work, they'll start to pick things up. Most of the designers I work with don't script. So when I say to them that we can write a couple of lines of code to do this easier, it's like a black box for them, like you are talking nuclear physics! So this ... would open the minds of the designers who just don't want to go there, because they are afraid of what they don't know." (P4)

Another participant pointed out that by having the script window open all the time and update continuously, users saw that the script window does the same thing as the model window:

"People say 'I know it makes my work easier, but I don't want to go into the black box and learn it.' But if they see that it's not that bad. You are already drawing lines all day and you can see that one line in the model is one line in the script. So you'll start to see how simple it is to input a few things." (P4)

Mental model

Scripting in CAD does not provide a close mapping between the domain world and the computer world. One is the world of shapes and forms and the other comprises syntax and algorithms. We received many positive comments about the difference between modeling, dataflow and scripting in terms of the users' mental model.

"Graph nodes and script lines are too abstract for designers and may not even mean anything. But as soon as they connect them with the model, it makes sense and things start to have meaning. The model makes the code more legible, and the model and the code together make the graph more legible. Scripting on its own is pretty bad. No one can figure out the structure of the system from code alone. But as soon as you add the model and graph, they think that the code is not really that bad. The first thing they need to do is to see all three together, then start from the one that is most familiar to them, which for most designers is the model." (P3)

Participants noticed that PIM's side-by-side windows helped them connect the abstract script to the more concrete model:

"You are taking two tools or two modes of description that are usually separate and putting them together. In this interface, this [scripting] becomes a lot like drawing and drawing becomes like scripting. And people from each camp can see the value of the other form." (P6)

While trying to create a vector in the script, one participant suddenly noticed that the model was reflecting his moves in the script. He then talked aloud and said that had he paid attention to the model and not just the script, he would have realized that it was much easier than coding alone.

One participant who identified himself as a *visual* person, still did not understand the value of scripting, but was thrilled to have access to the script in the model:

"Just to be able to work directly on the model sounds so much better to me than anything to do with the script. ... It's so much easier for me to think in the model or graph view. So anything that takes me away from the script really helps. I think the main reason is that usually script is very far away from the model. ... being able to do it directly here in the model is something I like. I appreciate that you can also do it here [script]. So if I work with this system a long time and get familiar with the script, then I'd start using the script a lot more. But initially having everything work here [in the model] makes much more sense to me." (P7)

Two participants were completely opposed to scripting as a way of modeling and were not convinced that PIM could help

with that. They criticized the use of syntax as what they called *machine language* and called for other ways of communicating with the system that were closer to the natural language.

Error prevention/detection

The immediate feedback that users received in the model for scripting actions, allowed for quick evaluation of the results, detection of errors on the spot, and fixing them before moving on to the next tasks. Participants acknowledged this and compared it to traditional scripting when feedback is only available after compiling a body of code.

"The feedback level allows a lot of close control. So you make specific decision as you go along, as opposed to writing a few lines of code and expecting a result, then having to back track to see what went wrong. This is very helpful." (P8)

Highlighting and brushing was also said to draw attention to the feedback that the system was giving them in all three windows in the interface. They also pointed out the effect it could have on non-programmers' experience with scripting when errors were easy to detect and fix.

"Errors are what makes programming scary for non-programmers. Because this is live, you write a little bit of code and you see the result. So it means that if there is an error, you can trace it back." (P11)

On the other hand, and to our disappointment, PIM caused some errors during the tasks. One particular type of error was in identifying relationship between objects. A few of the participants got confused with the dependency links in the model and made mistakes following them to the upstream or downstream objects. Another challenge they faced was connecting the edit toolbars to their corresponding model object, especially when they moved the toolbar away from the model. Both problems seem to be more about usability than concept and we believe there are simple ways to address them.

Trial and error/ Design exploration

The lookahead feature in PIM was a new tool for the participants that helped them explore scenarios quickly and revert back to the original state if necessary. Some raised the question of previewing large models, which we return to when we talk about complexity. Others asked whether the preview state can be saved as an iteration or alternative.

"Does it save that history? Does it revert back to its original location if I want to undo? I guess it's more history than undo. I may want to keep the information there, like a button in grey that keeps the information about what's been done recently, like steps that I can go back in time." (P10)

Preview at this stage is a means to evaluate how the model will be affected by the action that is being performed, especially in the script, rather than a history of users' actions. They also requested more selective preview, not only selecting downstream levels, but also selecting one or more objects in the model and previewing those only.

Complexity/ scalability

A large number of participants felt that, while PIM worked well for simple models, it would face challenges when models grew complex. Thus, they felt it would have issues 'scaling up.' For example, they did not feel it would work well if the model became 'too heavy' for the machine, if there were too many objects on the screen, if there was not enough screen real estate for all the visual elements of PIM to be display, or if the model was three-dimensional model.

"I think the dependencies can be a little confusing with the links. Here you are looking at a very simple line and point model, but if you have a complex 3D volume and one component may affect all of these components." (P4)

They also questioned us on how we would handle certain aspects of the features in real model, for exmple, which object will carry a script in the script tab when it defines multiple objects, or how we would highlight an object in the script when it appears in multiple places.

"It gets tricky once the code [in script tab] represents multiple objects. which object do you tag that to? When it's property of an object, it makes sense to attach it to that object. But when it deals with relationship between a, b, and c, then it makes more sense to put it in the script window." (P6)

On the other hand, they foresaw how some of these features could assist them in complex models. Highlighting and displaying dependencies in the model could help when the graph gets too visually cluttered and/or unorganized.

"With PIM, I don't have to be so concerned with how I organize the graph, because I know I can find what I want later, in a more intelligent way. For example, by using dependencies, I can find and focus on the objects and nodes that are impacted. That matters to me, because I make very large graphs. What this does is highlighting and drawing my attention to what matters and what I am working on." (P3)

Usability issues

We took note of the usability issues when they were observed in the tasks or noted by participants. These arose from our interface design choices and development limitations. The most important usability issues for us were the ones that affected PIM features and how they were perceived by users, among those was the representation of dependency links in the model. We chose a node-link design for displaying dependencies as is common in the graph. However, the links did not work well with the two-dimensional model and caused reduced visibility of both links and model objects. This resulted in participants missing dependency data that was presented to them in the model and became even more pronounced because the graph window was not implemented in the prototype. In other words, participants liked the fact that dependency data was available directly in the model, but did not benefit from it during the tasks because of what we believe to be our choice of visual representation.

A related problem happened with edit toolbars in the model. Some of the participants were confused or annoyed by the fact that sometimes links were hidden under tool bars. In these

cases, they moved toolbars around until they found a place where links and toolbars were more organized and overlapping did not occur. However, as soon as they opened new toolbars or displayed more links, the same problem reoccurred. Thus, floating toolbars and a large amount of links within the interface was clearly a design issue.

Smaller usability related comments included requests for sliders for numeric inputs, for more support in the script (syntax-directed editing), and for more data in the edit toolbar (such as type and update method). We tried to incorporate these suggestions and requests that we deemed effective in improving the usability of the prototype. Other issues remain unresolved, pending more investigation and redesign.

DISCUSSION AND CONCLUSIONS

In this section, we discuss our findings for PIM features and their implications for the Computer-Aided Design field. We focus our discussion around the three main components of Programming in the Model: liveness, localization of information, and lookahead (preview).

Liveness: As the most notable feature of PIM, liveness appeared to have a positive impact on users' experience by creating a mapping between the code and the geometry. Participants in our study were able to easily connect the abstract entities of the script and concrete objects in the model to create a more clear mental model of the scripting language and how it worked. This was because the system was 'live' and highly interactive. For designers who were fearful of programming and hesitant to try it, this simple side-by-side representation brought with it a familiarity with the code and reduced the perceived difficulty of scripting. Given this, we feel the first step in breaking the barriers to scripting in CAD, is to include *liveness* features such as those we implemented in PIM. We also note that, as one participant put it, the important aspect of liveness is in its timing: the mapping is only effective if it is immediate. Thus, the goal is to have *immediate liveness,* meaning that all representations must update concurrently with each action without delay. Participants encouraged us to aim for *extreme liveness*: liveness that includes and shows even the smallest steps of an action. Overall, this appears to be crucial for users to create effective mental connections between the code and the model.

Localization: Superimposing information onto the model view, including object properties, dependencies, and code, was found to be both beneficial and problematic. On one hand, it supports the user in building mental models and in viewing and manipulating the data related to objects locally in the model, where their focus of attention mostly lies. By doing so, the system eliminates the need for the user to switch to other representations to access the data. A strong benefit is that localization of specific aspects, for example, lines of script, appears to aid model comprehension and learning about scripting. On the other hand, the design of such an interface becomes very challenging. One of the challenges is maintaining the visibility of the model itself and avoiding obscuring it with these additional visual elements. Our attempts at preventing clutter in the model view by only displaying non-model elements selectively and temporarily and in a semi-transparent form turned out to be unsuccessful (or at least not enough), as was obvious in the study. Failing to follow dependency links in the model and identify relationships, being frustrated by toolbars and windows covering model objects and constantly moving them around, and losing connection between objects and their respective data were signs that the interface had failed to achieve its goal. These design issues seemed to be the reason for issues in search and navigation and user forecasts of scaling problems.

Lookahead (Preview): As a design exploration tool, lookahead was not very successful. Instead of seeing the purple preview model as an alternate state, participants paid more attention to what objects in the model or lines of code in the syntax were purple. In other words, they used preview to see what was being affected by their action. This has always been a secondary goal of preview, but after seeing the results of the study we now lean more toward recommending *preview as a debugging and error prevention tool* in CAD systems. Preview has the potential to support scripting by highlighting side-effects and preventing errors and unintended results. This is especially important in parametric modeling where objects are related to other objects and hidden dependencies can become problematic, particularly in complex models.

Another challenge was related to scale. It is well-known that visual programming languages do not 'scale up' without losing visibility. What we did was to put part of the visual dataflow on top of the model, which, on one hand, reduced the scalability of the interface as it inherits the visual programming problem in general. On the other hand, localizing only the desired parts of the dataflow reduced user references to the overall dependency model and may actually increase the scale at which dependency information is useful. Future attempts at localization of information on the CAD models must be carefully designed to handle complexity of current CAD models and tested to ensure scalability.

Programming In the Model research attempts to move CAD systems towards more usable and learnable scripting environments. Many attempts have been made to develop easier and more natural programming languages. But, in the CAD domain, there is the potential to use the modeling interface itself as one of the primary interactors with code, and to enable more concrete mental models of the abstract concepts of the script. Our preliminary observations reveal that multi-directional liveness shows real promise in delivering this goal. The implication is that commercially-available CAD systems should strongly consider incorporating immediate liveness, if not what we term 'extreme liveness,' into their interfaces. From a designer's perspective, liveness reduces the distance between design work and the code that is now a necessary part of that work. It should enable better understanding of the parametric design media and thus improved model quality. From a CAD vendor's perspective, liveness is a strategic advantage. The recent history of parametric CAD (at least in building design) can be told as a tale of improving interfaces to the parametric engine, largely through interactions with the dataflow graph. Together, liveness, localization and lookahead open new opportunities for improved (and more

competitive) system designs. The difficulty of making such changes will depend on each vendor's code base. However, systems such as Dynamo (Autodesk), Grasshopper (Rhino) and GenerativeComponents (Bentley) are all increasing the richness of interaction with their diverse models, for which liveness sets an ambitious but reachable goal. At this stage of our research, we are very convinced that liveness is a crucial goal. Other PIM strategies such as localization of data in the model view and previewing the changed model show promise but need more research and careful redesign if they are to be incorporated in commercial CAD systems.

Based on the results of this study, we plan to redesign some of the features such as dependency links and implement the dependency graph window. In addition, we need to push the implementation forward in a way that allows us to conduct studies with more complex tasks and comparative conditions (PIM with and without each feature). Additional features may be discovered and added to the system as well. This iterative process of design, implementation, and evaluation will give us more detailed feedback about these features, therefore enabling us to make recommendations for incorporating these features in commercial CAD systems.

REFERENCES

1. Aish, R. Extensible computational design tools for exploratory architecture. In *Architecture in the digital age : design and manufacturing*. Spon Press, New York, NY, 2003.
2. Aish, R. Designscript: Origins, explanation, illustration. In *Computational Design Modelling*, C. Gengnagel, A. Kilian, N. Palz, and F. Scheurer, Eds. Springer Berlin Heidelberg, Jan. 2012, 1–8.
3. Aish, R., and Woodbury, R. Multi-level interaction in parametric design. In *SmartGraphics, 5th Intl. Symp., SG2005*, A. Butz, B. Fisher, A. Krüger, and P. Oliver, Eds., LNCS 3638, Springer (Frauenwörth Cloister, Germany, August 2005), 151–162.
4. Blackwell, A. First steps in programming: a rationale for attention investment models. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments* (Arlington, VA, USA, 2002), 2–10.
5. Boeykens, S., and Neuckermans, H. Visual programming in architecture, 2009.
6. Burnett, M. Visual programming. *Encyclopedia of Electrical and Electronics Engineering* (1999), 275–283.
7. Burnett, M. M., and Scaffidi, C. End-user development. In *Encyclopedia of Human-Computer Interaction*, M. Soegaard and R. F. Dam, Eds. The Interaction-Design.org Foundation, 2011.
8. Celani, G., and Vaz, C. Cad scripting and visual programming languages for implementing computational design concepts: A comparison from a pedagogical point of view. *International Journal of Architectural Computing 10*, 1 (2012), 121–138.
9. Cooper, A., Reimann, R., Cronin, D., and Cooper, A. *About face 3: the essentials of interaction design*. Wiley Pub., Indianapolis, IN, 2007.
10. Dertouzos, M. ISAT Summer Study: Gentle Slope Systems; making computers easier to use. Presented at Woods Hole, MA, 16 August 1992.
11. Hutchins, E. L., Hollan, J. D., and Norman, D. A. Direct manipulation interfaces. *Human-Computer Interaction 1*, 4 (1985), 311.
12. Jabi, W. *Parametric Design in Architecture*. Laurence King Publishing, 2012.
13. KhanAcademy. Khan academy computer science, September 2012. Accessed on 11/04/2013 at http://www.khanacademy.org/cs/new.
14. Ko, A., Myers, B., and Aung, H. H. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on* (2004), 199–206.
15. Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M. B., Rothermel, G., Shaw, M., and Wiedenbeck, S. The state of the art in end-user software engineering. *ACM Comput. Surv. 43*, 3 (Apr. 2011), 21:121:44.
16. Maleki, M., and Woodbury, R. Programming in the model – a new scripting interface for parametric CAD systems. In *Proceedings of the 33rd Annual Conference of the Association for Computer Aided Design in Architecture*, ACADIA (Cambridge Ont., 2013), 191–198.
17. Myers, B. A., Smith, D. C., and Horn, B. Report of the end-user programming working group. In *Languages for Developing User Interfaces*, B. A. Myers, Ed. A. K. Peters, Ltd., Natick, MA, USA, 1992, 343–366.
18. Peters, B., and Peters, T., Eds. *Inside Smartgeometry*. John Wiley & Sons, 2013.
19. Senske, N. Fear of code: an approach to integrating computation with architectural design. Thesis, Massachusetts Institute of Technology, May 2005.
20. Shneiderman, B. Direct manipulation: A step beyond programming languages. *Computer 16*, 8 (1983), 57–69.
21. Terry, M., and Mynatt, E. D. Side views: persistent, on-demand previews for open-ended tasks. In *Proceedings of the 15th annual ACM symposium on User interface software and technology*, UIST '02, ACM (New York, NY, USA, 2002), 71– 80.
22. Victor, B. Inventing on principle, 2012. Accessed on 11/04/2013 at http://worrydream.com/#!/InventingOnPrinciple.
23. Victor, B. Learnable programming, Sept. 2012. Accessed on 11/04/2013 at http://worrydream.com/#!/LearnableProgramming.
24. Woodbury, R. *Elements of Parametric Design*. Taylor and Francis, July 2010.