B. Dave, A. I. Li, N. Gu, H.-J. Park (eds.), *New Frontiers: Proceedings of the 15th International Conference on Computer-Aided Architectural Design Research in Asia CAADRIA 2010*, 117–125.

PROGRAMMING IN THE MODEL

Combining task and tool in computer-aided design

MARYAM M. MALEKI and ROBERT F. WOODBURY
School of Interactive Arts and Technology, Simon Fraser University, Surrey, BC, Canada
mmaleki@sfu.ca; robw@sfu.ca

Abstract. Programming takes designers away from typical domain- and task-based interfaces such as three-dimensional modellers. It thus imposes additional cognitive load on the already challenging design task. Programming in the model is a system design strategy that embeds the act of programming in a 3D CAD model. This paper presents the argument for programming in the model and two user interface constructs that support such programming.

Keywords. End-user programming; scripting; visual programming; direct manipulation.

1. Introduction

Most computer-aided design (CAD) systems for buildings provide designers with powerful graphical user interfaces that satisfy the requirements of conventional designs, but sometimes this is not enough for designers who want to explore unconventional design ideas or to work more efficiently. In such cases, designers need to use computer programming to have more capabilities and freedom to explore (Aish, 2003). However, the shift from a three-dimensional model to a programming environment presents cognitive challenges. Designers who program in CAD systems face the same difficulties that other end-user programmers face in other domains. In this ongoing project we address this issue and propose programming in the model as a way of using designers' spatial and visual abilities to bring programming and design tasks into closer proximity. In this paper we describe the problem of end-user programming in CAD and present some of the findings from the literature. Then

we introduce programming in the model as well as two example aspects of such programming.

2. The problem

Computer programming can improve designers' efficiency by allowing them to program iterative operations, making modules for future use, and helping others by sharing code. It also helps them with their design task by giving them freedom from the limitations of the interface and allowing them to explore unconventional and more complex forms, as well as accommodating change in the design process and enabling rapid exploration of variations and alternatives (Woodbury, 2010). In design, the word "scripting" often replaces "programming." The difference lies only in reception: "scripting" seems less foreboding than "programming." There is no meaningful technical difference between the two terms.

Users of computer-aided design systems have at least one special skill in common: they can understand and relate to a 2D or 3D model very easily. They are experienced and comfortable with the model and it makes sense to them. In working with CAD systems, their purpose usually is to create a model, for example a 2-dimensional drawing of a house or a 3-dimensional model of an engine or a skyscraper.

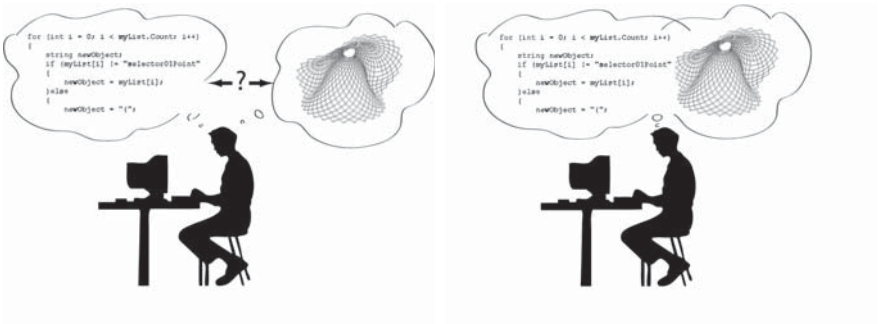
Designers may be capable with the model itself, but most are not as comfortable with programming. Some designers struggle to understand the underlying structure of a programming language and the structure of the model. Others have a hard time making sense of the syntax of the program.

Most such designers are amateur programmers acting as end-users, not system developers – they have relatively little formal education in computer science or programming. End-user programmers are people who are professional in domains other than programming who need to write programs in the context of their expertise and as part of their job in order to get a task done. End-user programmers may use any programming language independently or as part of a software depending on the nature of their job, such as spreadsheets for accountants, SolidWorks for mechanical engineers, and AutoCAD for architects.

For an end-user programmer, the focus is on the task in hand and not programming, and the goal is mostly to do the task, rather than to produce efficient, reusable code. Programming is only a tool for making algorithms to perform repetitive tasks quickly and more efficiently or to do things that are hard or impossible to do in the graphical user interface.

Literature shows that end-user programmers find difficulty when programming is separate or different from their main task (design, analysis, etc.). To

write or edit code in existing CAD systems (e.g., Generative Components, Grasshopper, Rhino script), the user has to open the scripting / visual programming window as a separate program with a completely different interface and functionality. During the process of writing the code, the user goes back and forth between the model and program in order to find the right objects and operations required for the task and also to test the code and see the result. The shift in mental activity from objects and forms to code and algorithm occurs repeatedly. Separation of design and program requires switching between tasks, with consequent loss of both focus and efficiency (see figure 1).



A: Programming task is usually separate from the design task.

B: Bringing programming closer to the design of the model.

Figure 1 . Computer programming and design in CAD.

3. Background

Kelleher and Pausch (2005) name *difficulty of code* as one of the challenges of programming languages for non-programmers that causes confusion and syntax errors. This issue can be addressed by simplifying entering the code or finding alternatives to typing code, e.g., constructing code by using graphical objects or interface buttons, and by demonstrating the action in the interface.

Green and Petre (1996) introduced the cognitive dimensions frameworks for evaluating visual programming languages. They consider *closeness of mapping* to be one of the important criteria in the efficacy of an end-user programming environment. They argue that the closer the programming world is to the problem world, the easier it is for users to mentally map between them. It's much harder to achieve this closeness in textual programming languages. Pane and Myers (1996) emphasise the importance of matching between the system and the real world by keeping the programming language consistent with user's external knowledge, in order to reduce novice's learning load and avoid confusion.

Spreadsheets are good examples of end-user programming environments that bring programming directly to the problem domain. In spreadsheets, the problem domain is the task being done, which is typically amenable to representation as a table or array. The representation is a table of cells filled with values and formulae. User can write programming statements directly into the cells and see immediate feedback in the form of numbers or other values. However, like most end-user programming systems, Excel's task mapping becomes much less clear when a user engages scripting (Visual Basic for Applications (VBA)).

There is a large body of research that addresses the issues of end-user programming and programming in general. Concepts such as *direct manipulation* and *visual programming* are particularly applicable to the issues of end-user programming in CAD. Two aspects of direct manipulation (Shneiderman, 1983) are reducing the distance between user's thoughts and the system and giving them the sense of directly manipulating the objects and not through the program and computer (Hutchins et al., 1985). Visual programming (the direct manipulation of programs) has appeared in CAD systems interfaces (e.g., Rhino's Grasshopper). In this technique, shapes and their connections represent programming constructs and the flow of data or control. Although visual programming may seem to be a good way of using designers' visual abilities, it still separates the task of programming from the task of designing the model and suffers scaling problems as models grow.

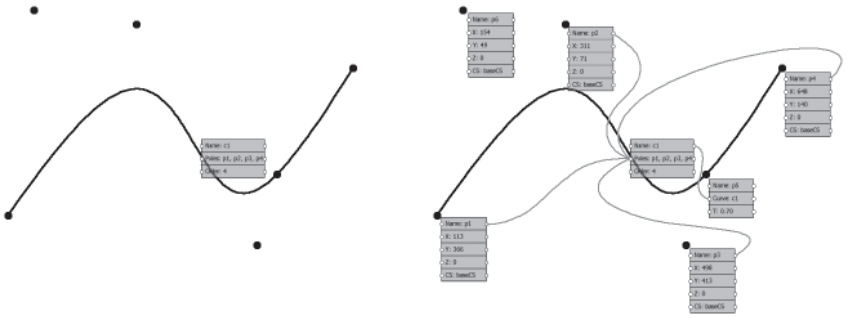
4. Programming in the model

In order to use designers' spatial and visual abilities in programming we suggest contextualising programming concepts and subtasks directly in the three-dimensional model. Relationship between objects are assigned and modified in the model view. Object properties are accessed directly in the model view by clicking on the objects. Programming constructs such as functions and loops are created in the model directly where they are needed and other objects in the model and their properties can be used as arguments.

Programming in the model also uses aspects of visual programming (e.g., Rhino's Grasshopper) but is different in that the goal is not just to make programming visual, but to bring it closer to the model. We use both text and visual elements in the model to program. In order to avoid scaling problems of visual programming for larger models, we give the user access to the underlying textual program at all time. Every programming task performed in the model is reflected in the textual language and vice versa. Small pieces of a program can be created in the model and larger pieces and modification can be done in the scripting window.

To make the concept of programming in the model clearer, we explain how it supports two types of tasks commonly performed in CAD: (1) accessing and modifying object properties and dependencies, and (2) making lists or series of objects.

Model objects (points, curves, solids) each have a collection of properties, the specifics of which depend on the creation method used. For instance, a point created by defining Cartesian coordinates may have a list of properties including coordinate system and X, Y, and Z translations, whereas a point that is located on a curve has no coordinate system but has additional properties such as the curve and the point's position on the curve given by a parameter *T*. In programming in the model, these properties are accessible and modifiable in the model. Short or extended lists of properties may be displayed based on the user's demand for one or some of the objects in the model. Figure 2 (A) shows a model with visible object properties. The level of visibility of property information boxes is adjustable by the user by changing their transparency.



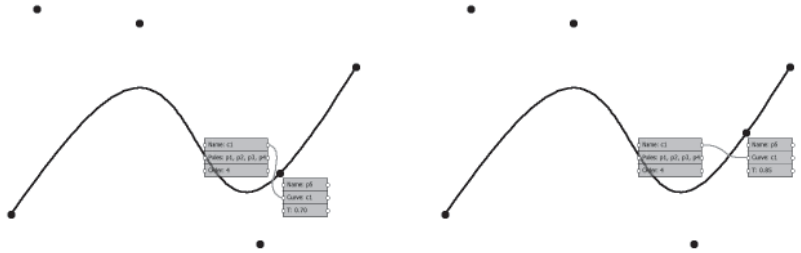
A: Object properties *B: Relationship between objects*

Figure 2 . Properties and dependencies represented in the model

In parametric modeling, an object may be dependent on other objects or variables if other objects or variables are used to determine the value of its properties. In programming in the model, these dependencies are shown by links connecting object properties to each other. For each object, inputs come from the left side and outputs exit from the right. This is similar to Generative Components' symbolic view and Grasshopper's graphic representation of the model. In fact, they represent the same information. The main difference is that the graph is embedded in the model. There is no need for a different window, therefore no need for the user to go back and forth between windows to get information about object relationships by trying to map the graph to the model. Figure 2 (B) shows dependencies between a curve and its poles by links connecting the poles to the left side (representing inputs) of the *Poles* field of the curve's properties. It also shows its relationship with a point on the

curve by a link connecting the right side (representing outputs) of the curve's property box to the point.

Object properties are editable in the model and the change immediately appears in the model. In figure 3 *point05* is located on the curve and its position is determined by a parameter *T* that has a range between zero and one. In this figure the user changes *T* from 0.70 to 0.85 by editing the field named *T* in the point's property list in the model. The change in *T* moves the point closer to the end of the curve immediately after she hits *Enter.*



A: T = 0.70 *B: T = 0.85*

Figure 3. Changing object properties in the model.

The user can change the dependencies in the model as well. By grabbing one end of a link, she can detach it from an object and attach it to another. Figure 4 demonstrates this action. In the figure on the left, *point02* is the second pole of the curve, so the curve is dependent on it. This dependency is shown by a link that connects *point02* to the *Poles* field of the curve properties. By grabbing the end of the link that is attached to *point02*, the user can move it to another point (in this case *point06*) to change the pole. The shape of the curve changes based on the new pole to reflect the change and the property list is updated accordingly.

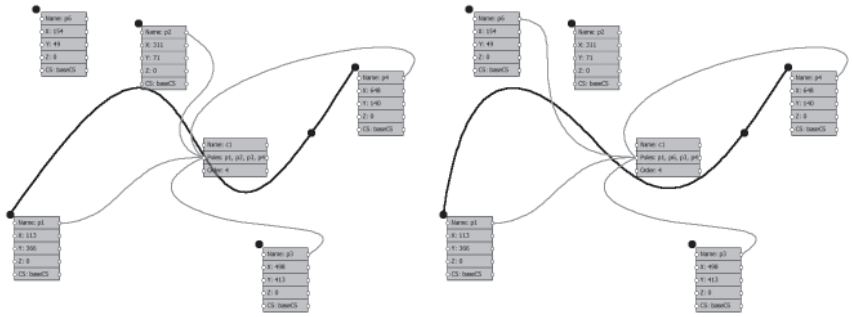
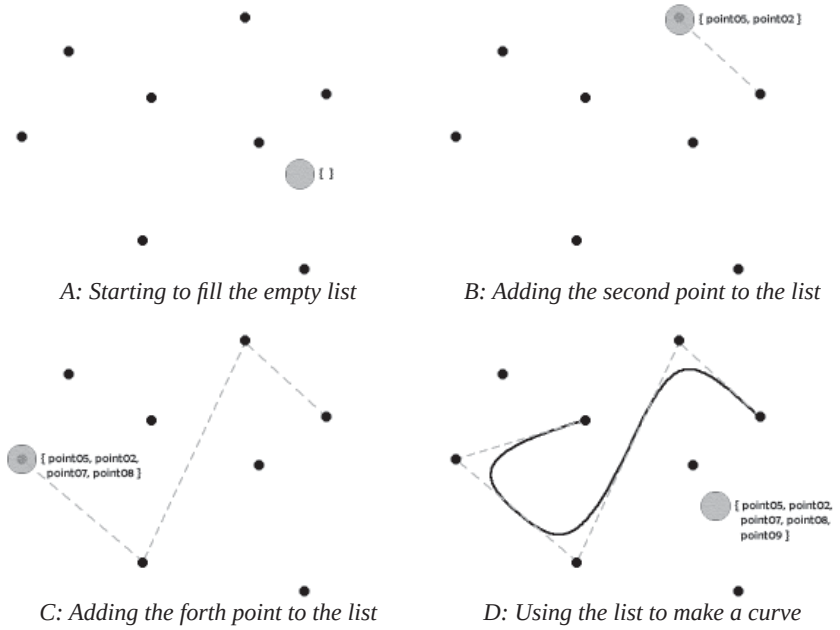


Figure 4 . Changing relationships in the model

The second example demonstrates how lists can be made in programming in the model. In parametric modeling, sometimes it is necessary to provide a list of objects as an input for other objects or a function. The syntax of a list is different in each programming language and can be confusing for users. For example in Generative Components, a list is defined by two curly brackets and object names are separated by comas, e.g., {*point01, point02, point03*}. Missing any of these characters results in syntax errors. Some systems provide the ability to select the objects in the model to be added to the list. But there is usually no feedback in the model on the current status of the list and the syntax needs to be checked for possible errors.

In programming in the model, lists are objects that can be placed in the model. The user can move a list around, select objects to be added to the list, see the textual representation of the list, and also keep track of the list in the model through the visual feedback in the model. In figure 5 a list of five points is made in the model and used for creating a curve. As points are being added to the list, the textual list is created with the correct syntax, without the need for the user to constantly check for syntax errors. At any time during this process and afterwards, the user can modify the list by typing the name of the points directly in the textual representation next to the list object. This ability



A: Starting to fill the empty list *B: Adding the second point to the list*

C: Adding the forth point to the list *D: Using the list to make a curve*

Figure 5 . The process of making a list of five points to create a curve by poles

gives the user the option to choose between multiple methods of making and editing lists.

5. Discussion and conclusion

As mentioned earlier in the paper, this is a work in progress. We have much more to find and explore. We are currently working on other programming concepts such as loops and conditionals and how to represent them in the model. Also, we are developing a prototype of the system that can be used by designers in future user testing of the project. It is important to mention that programming in the model is not a programming language on its own. Rather it is a way of presenting programming and scripting to designers that can be implemented in any CAD system and adapted to the programming language used in that system. This textual scripting language then works with programming in the model and provides additional editing capabilities when model grows in size and complexity or for designers who choose to write code.

Programming in the model will benefit designers by breaking the initial fear of code that prevents some of them from starting scripting in CAD. This is done by gradually introducing programming concepts in the 2D-3D model and allowing direct manipulation of the program in the model. Designers who choose to learn scripting along the way can follow the development of the textual program and familiarise themselves with the syntax. Although the program is represented visually in parts, there is a significant difference between programming in the model and visual programming in that the emphasis is on the closeness of program and model and not just visualising it.

Acknowledgements

This research is partially supported by the Canadian NSERC Discovery Grants program, which support is gratefully acknowledged.

References

Aish, R.: 2003, Extensible computational design tools for exploratory architecture, *Architecture in the digital age: design and manufacturing,* Spon Press, New York.

Green, T. R. G. and Petre, M.: 1996, Usability analysis of visual programming environments: a cognitive dimensions' framework, *Journal of visual languages and computing,* **7**(2), 131–174.

Hutchins, E. L., Hollan, J. D. and Norman, D. A.: 1985, Direct manipulation interfaces, *Human–computer interaction,* **1**(4), 311.

Kelleher, C. and Pausch, R.: 2005, Lowering the barriers to programming: a taxonomy of programming environments and languages for novice programmers, *ACM comput. surv.,* **37**(2), 83–137.

Pane, J. F. and Myers, B. A.: 1996, *Usability issues in the design of novice programming systems,* Carnegie Mellon University, School of Computer Science, Pittsburgh, Penn.

Shneiderman, B.: 1983, Direct manipulation: a step beyond programming languages, *Computer*, **16**(8), 57–69.
Woodbury, R. F.: 2010, *Elements of parametric design*, Taylor and Francis, forthcoming.