Programming In the Model:
A new perspective on scripting in CAD systems

by

Maryam M. Maleki

M.Sc. (Architecture), Shahid Beheshti University, 2006

Dissertation Submitted in Partial Fulfillment
of the requirements for the degree of

Doctor of Philosophy

in the
School of Interactive Arts and Technology
Faculty of Communication, Art and Technology

APPROVAL

Name:	Maryam M. Maleki
Degree:	Doctor of Philosophy
Title of Thesis:	Programming In the Model: A new perspective on scripting in CAD systems

Examining Committee:

Dr. Bernhard Riecke
Assistant Professor
Chair

Dr. Robert Woodbury
Professor
Senior Supervisor

Dr. Halil Erhan
Assistant Professor
Supervisor

Dr. Carman Neustaedter
Assistant Professor
Supervisor

Dr. Ted Kirkpatrick
Associate Professor, Computing Science
Internal Examiner

Dr. Charles Eastman
Professor, Georgia Tech
External Examiner

Date Approved: April 4th, 2014

Ethics Statement

Abstract

Scripting has become an integral part of design work in computer-aided design (CAD), especially with parametric systems. Designers who script face a 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 and a focus group with HCI 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.

Keywords: End-user programming; Computer-aided design; Parametric modeling; Cognitive dimensions of notations framework.

To Roham, for his love, patience, and support.

Acknowledgments

Finishing a Ph.D. is not a one man (or woman) job, and I owe a debt of gratitude to all of those who mentored me, helped me, held my hand, and patiently listened to my complaints during the last few years.

My deepest gratitude goes to my senior supervisor and my mentor Dr Robert Woodbury, to whom I owe my academic success. Rob never stopped teaching me and supporting me, both in school and in life. He gave me freedom to pursue my research interests, while at the same time carefully watched over me and kept me on the right track. I feel extremely lucky to have worked with him and to have my name next to his on this work and many others.

I also would like to thank my supervisors Dr Carman Neustaedter, for his invaluable help with my research methods, and Dr Halil Erhan, for always asking the tough questions and making me think. They constantly pushed me to do better and to aim for excellence.

And many thanks to my examiners Dr Ted Kirkpatrick and Dr Charles Eastman for taking the time to read my thesis and for their excellent questions and great feedback.

Lanz Singbiel and Matt Mercer helped bring my ideas to life in the prototype and endured my iterative design process and the many many changes and recoding it required. The participants of the user study and the focus group selflessly contributed their time and expertise to this research and evaluated the prototype with no expectations. I thank them all.

This project was partially supported by the NSERC Discovery, Postgraduate Scholarships, and Collaborative Research and Development programs; by MITACS Accelerate and by the GRAND Networks of Centres of Excellence, whose support is greatly appreciated.

And finally, I would like to thank my family from the bottom of my heart. My husband Roham stood by me and supported me all these years, so that I can pursue my dream. He deserves an honorary degree for being the spouse of a Ph.D. student. My parents, Parvin and Mohammad, who believed in women's higher education and constantly pushed me since I was six years old to aim high and never give up. And thanks to my sisters, Behnaz and Behdis, for their love. This degree belongs to all of you.

Contents

List of Tables

List of Figures

Chapter 1

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[1], in order to have the freedom to explore unconventional and complex forms. Aish (2003) categorizes architectural design into two categories: conventional and exploratory. The majority of architectural design and construction work is in the first category. The conventional CAD systems that support this category are either low level (line, arc,...) or high level (wall, window,...) and work well for conventional designers. Exploratory design though, challenges conventions by introducing new forms and geometry. 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. In order to do that, designers must have necessary programming skills. 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 (Woodbury, 2010). In recent years, parametric systems have become widespread in CAD practice, yet remain difficult to use and learn. Indeed, several recent

[1]In computer science, the word *scripting* is used to refer to a specialized form of programming. There are characteristics that define a scripting language from a programming language, such as ease of use and loose structure (Morin & Brown, 1999; Loui, 2008). I use *scripting* and *programming* interchangeably in this document to refer to the code that is used in CAD systems.

books focus on enabling more effective use of such systems (Woodbury, 2010; Jabi, 2012) and there are international workshops on fostering advanced communities of practice (Peters & Peters, 2013).

The move from direct manipulation of the model in the GUI to writing a computer program presents many challenges and barriers, some of which are specific to CAD, but most of which are shared with other end-user programmers. And end-user programmer is a domain expert who writes code as part of the job and not for the sake of programming (Nardi, 1993; Jones, 1995; Myers et al., 2007). Programming on its own is a difficult task, because it requires problem solving skills, algorithmic thinking, and using the syntax of the language to express one's intentions to the computer (Lieberman & Fry, 1995; Pane et al., 2002; Woodbury, 2010). In addition to these difficulties, there are barriers and problems specific to end-user programming. The fact that the focus of end-user programmers is on the task, and the goal is to perform said task, makes programming only a tool for solving a problem in their work. As a tool, programming should not steal the focus from the task and should not present additional challenges to that of the task itself. This is not an easy goal to achieve in designing an end-user programming environment. There is a body of research dedicated to end-user programming with the goal of discovering and solving these barriers and making end-users' programs more reliable.

As an architect and an HCI researcher, I am strongly motivated to bring findings of the HCI research in end-user programming to parametric CAD systems, in order to support architects and engineers in their transition from modeling to scripting. I started with the following questions.

- How can we reduce the fear of coding among designers and encourage them to use it in their design process?
- Can scripting come out of its hidden place to the forefront of the CAD interface?
- Does the programming notation have to be radically different from the modeling notation?
- Why have visual programming languages gained so much popularity among architects?
- Which is more important: the scripting language or the way it is presented to the end-user programmer?

- Are designers who use scripting in their work different from the rest? Or would others use scripting too, given the right scripting language and environment?

The underlying goal of all of this is for computer programming to some day become a mainstream design tool in CAD systems. To narrow down the scope and focus my research, I formed the following research questions.

- *RQ 1:* What makes scripting/programming a difficult task? How can we reduce this difficulty in CAD systems?
- *RQ 2:* What characteristics in a CAD scripting environment help reduce designers reluctance to use scripting?
- *RQ 3:* What are the barriers designers face when they program in CAD? How can we reduce these barriers?
- *RQ 4:* How can we enable designers to transfer and use their skills and expertise in modeling to scripting?

Programming In the Model is my humble attempt to answer some of these questions. My ideas are yet to be tested in the field, with different types of users and in real-world design tasks. But it is encouraging to see CAD developers starting to move in the same direction.

In this document, I first present a background and literature review of the computer-aided design domain, followed by the computer programming and end-user programming domains. I then explain my design rationale and methodology for Programming In the Model, with a full description of its features and my prototype system.

I also present the procedures, analyses and results of the user study and the focus group evaluations of Programming In the Model, concluding with a discussion of the results and the lessons I learned from this project.

Chapter 2

Background and related work

In this chapter, I present background information about the domain of Computer-Aided Design and introduce domain concepts and words that I will use throughout this document. Then I review the programming, end-user programming, and HCI literature and the related works.

2.1 Computer-aided design

Computer-aided design (CAD) refers to the use of computer technology in designing, drafting, and manufacturing of a product such as an engine, or part or all of a building, by architects, industrial designers, and civil, mechanical, and electrical engineers.

Conventional CAD systems follow the model of paper and pencil of design, in which the designer adds and removes marks on paper (Woodbury, 2010). AutoCAD is a well-known example of such systems. In such systems, the designer creates an object by specifying its location and size, by entering values in a command line entry or using the snap feature to use other objects' relative location or size. The snap feature takes an existing object's location or size and passes it on to the object being created. But this does not make a lasting relationship between these objects, in that the second object only gets the current values and sticks with them, even if the first object changes in location or size.

In contrast, designing in a *parametric modeling system* requires the designer to create new objects in relation to other objects. These are lasting relationships that constantly pass the data from one object to another and update the model. For example, in order to create a line on a set of two points, the designer assigns the points as start point and end point of the line. This is similar to snapping the line to these two points in a conventional CAD system, but different in that there is now a dependency between the line and the points. The location and size of the line is now determined by the points and will change when they move.

Parametric modeling supports design exploration by automatically updating the model when the designer makes a change. For example, changing the floor to ceiling height in a conventional system requires the designer to manually update all other components, such as windows and facade elements, whose size or location depend on that height (Schmidt & Wagner, 2004). But a parametric system propagates that change of height throughout the model and updates those other components automatically, resulting in less repetitive and manual work for the designer.

The task of setting up the dependencies and constraints requires a special kind of design thinking that does not come naturally to everybody (Woodbury, 2010). According to Aish & Woodbury (2005), parameterizing design makes it more complex for designers as they have to focus on both the design and the underlying structure and relationship between objects. Also, sometimes the types of relationships that designers need are more than what is possible for system developers to predict and pre-make for them. Those cases require programming or scripting to set up more complex relationships.

In addition to empowering unconventional and exploratory design, programming improves efficiency by allowing the designers to program iterative operation (Burry, 1997), making modules for future use (Woodbury, 2010), and helping others by sharing code (Gantt & Nardi, Gantt & Nardi).

There are several different types of parametric systems. The two most notable types are *constraint systems*, which solve a set of constraints, and *propagation systems*, in which the system computes from known to unknown in a directed graph (Aish & Woodbury, 2005; Hoffmann & Joan-Arinyo, 2005; Woodbury & Burrow, 2006). I refer you to Woodbury's (2010) *Elements of Parametric Design* for a clear and concise description of these systems. I use the propagation paradigm in my descriptions (and in the prototype, as I will explain later) because of its simplicity, reliability, and speed.

In the next section, I define domain concepts and terms that I use in this document, for the readers unfamiliar with CAD and parametric modeling.

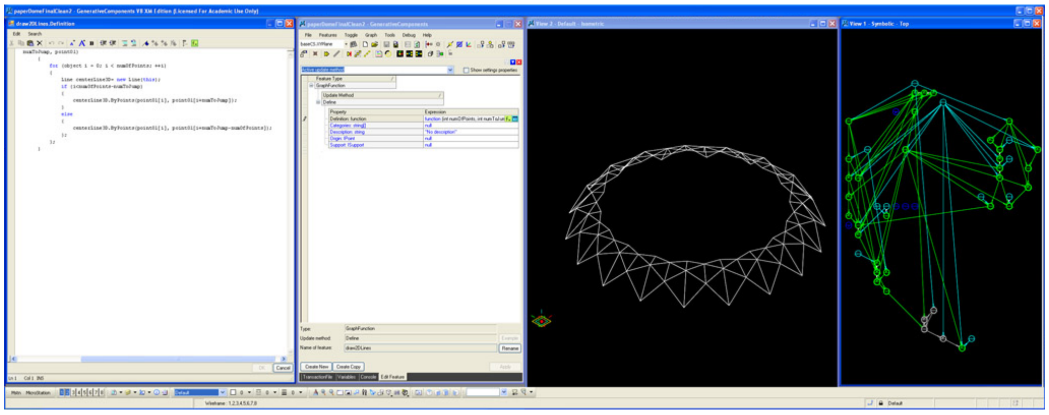
Definitions

For a designer using a CAD system, the primary accessing scheme is the location, and the design task is embedded in geometry. Therefore an essential role of any CAD system is representing the model to the designer (Figure 2.1). The ***model view*** is an important part of the graphical user interface (GUI) of any CAD system. This is a window that represents the model in the space and allows the designer to view and evaluate it from different angles and with different resolutions. In most CAD system, the user can directly manipulate the model in the model view and add or edit components, using absolute location in the space or relative to other components. Toolbars in the GUI provide buttons for creating and editing actions, while command-line interaction is sometimes available, if preferred by users.

In a propagation CAD system, each object can be identified as a ***node***, with data flowing from one node to another through a ***link***, depending on how they are related. The node-link diagram that graphically represents this ***relationship*** is called the ***graph view***. For example, a model with a point located on a line has two nodes in its graph, one for the point and one for the line. A directed link goes from the line node to the point node to show the direction of data flow. Depending on how the line itself was created, there could be other nodes and links in the graph as well.

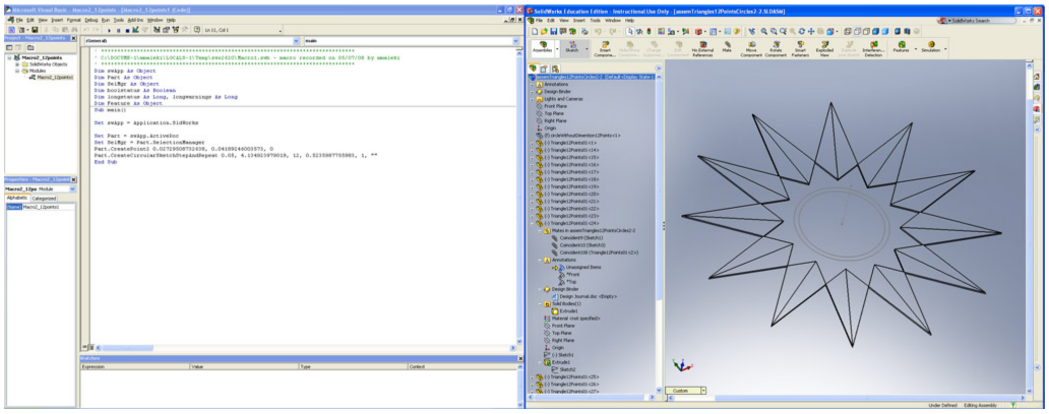
Each object in the model has ***properties*** that define it. For example, a point that is located on a line has a *name* property (a unique string), a *line* property that identifies which line it is located on, and a *parameter* or T property that defines its location on the line. When a line (for example line01) is selected as its *line* input, the point becomes ***dependant*** on line01. The *parameter* value can be an independent number (such as 0.4), a variable (such as myT), which makes the point dependant on myT variable, or an expression (such as point02.T*0.1), which makes the point dependant on point02.

Each object also has an ***update algorithm*** or ***update method*** that determines how and with what types of properties it is created. For example, a point can be created by defining its location on a line, in which case it needs a line and a parameter property, or by its coordinates in a coordinate system, in which case it needs a coordinate system and a set of X, Y, and Z coordinates. Update algorithms are explicitly defined in some parametric systems, but are implicit within the objects in other systems.



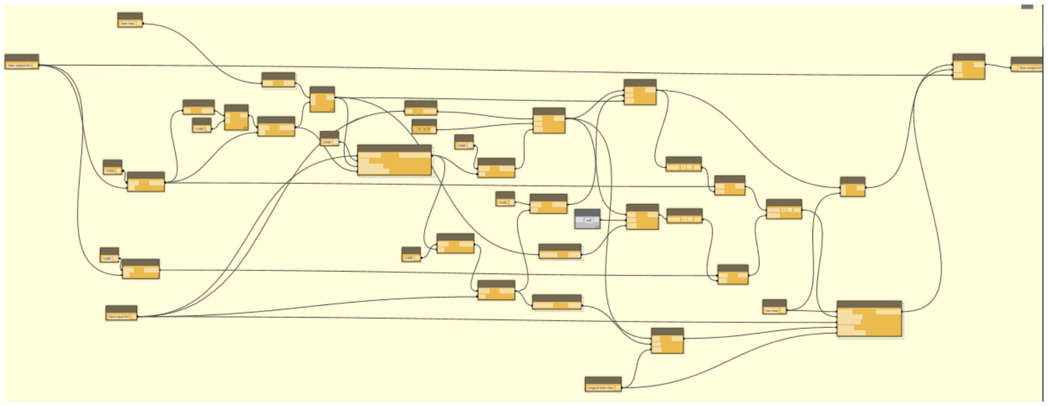
```
numToJump, point01)
    {
        for (object i = 0; i < numOfPoints; ++i)
        {
            Line centerLine3D= new Line(this);
            if (i<numOfPoints-numToJump)
            {
                centerline3D.ByPoints(point01[i], point01[i+numToJump]);
            }
            else
            {
                centerline3D.ByPoints(point01[i], point01[i+numToJump-numOfPoints]);
            };
        };
    }
```

(a) GenerativeComponents. Windows from left to right: script view, property manager, model view, graph view



```
' ******************************************************************************
' C:\DOCUME~1\mmaleki\LOCALS~1\Temp\swx2620\Macro1.swb - macro recorded on 05/07/08 by mmaleki
' ******************************************************************************
Dim swApp As Object
Dim Part As Object
Dim SelMgr As Object
Dim boolstatus As Boolean
Dim longstatus As Long, longwarnings As Long
Dim Feature As Object
Sub main()

Set swApp = Application.SldWorks

Set Part = swApp.ActiveDoc
Set SelMgr = Part.SelectionManager
Part.CreatePoint2 0.02729508732638, 0.04189246003573, 0
Part.CreateCircularSketchStepAndRepeat 0.05, 4.134923979019, 12, 0.5235987755983, 1, ""
End Sub
```

(b) SolidWorks interface on the right, including property manager bar and the model view, with Microsoft Visual Basic window on the left



(c) Dynamo. A parametric system with a visual programming language

Figure 2.1: Parametric CAD systems

Most CAD systems have built-in programming languages, and almost all allow the use of a general purpose programming language for further customization. These programs are displayed in a window we call the ***scripting window***[1]. As I will discuss later in this chapter, there are ***textual programming*** languages, with text as their notation, and ***visual programming*** languages, which use other notations such as diagrams. Bentley's Generative Components is an example of a parametric CAD system with a textual scripting language (GCScript) and a textual general purpose programming language (C#). Solidworks, although not a propagation system, has a textual general purpose language (Visual Basic for Applications (VBA)) and no built-in scripting language. Autodesk's Dynamo (an add-on to Vasari and Revit) and McNeel's Grasshopper (an add-on to Rhino3D) are examples of visual programming languages used in parametric modeling. Figure 2.1 shows the interface of three of these systems.

2.2 Computer programming

Computer programming is the act of extending or changing a system's functionality (Van-Roy & Haridi, 2004), by creating a sequence of instructions to enable the computer to do something. There is usually a problem that needs to be solved, a set of requirements that need to be met, or a goal to be achieved. Therefore, programming requires problem solving skills. In addition, the solution has to be described as a set of instructions for the machine to follow. So a programmer must be able to think algorithmically. These two skills make programming a hard task (Woodbury, 2010).

After one acquires these skills, one still has to know how to communicate the instructions with the computer (Pane et al., 2002). The syntax of the programming language and the support provided by the programming environment play an important role in this aspect of the programming task. Lieberman & Fry (1995) state that what makes programming cognitively difficult is that the programmer must imagine the dynamic process of execution while he or she is constructing the static description. Rogalski & Samurcay (1990) define the complexity of a programming language for learners as a function of the distance of the control and the variety of virtual entities, such as variables and memory.

[1]If I want to be precise, I should distinguish between a scripting window (for a built-in scripting language) and a programming window (for a general purpose programming language). But as mentioned before, I use scripting and programming interchangeably in this document.

There are certain characteristics that make programming different from other forms of computer usage. For example, we never refer to our work in a word processor as programming. In such tasks we work directly on what we are creating, for example, we directly manipulate the text to achieve the format that we want. In a programming task, we write instructions for the computer to do something at a future time to achieve a goal. So we manipulate the object through a different notation. Blackwell (2002) summarizes the common features of programming as (a) loss of the benefits of direct manipulation, and (b) introduction of notational elements to represent abstraction. Both of these features have significant effects on cognitive aspects of programming tasks.

2.2.1 Direct manipulation

Shneiderman (1983) describes continuous representation of the object of interest, physical actions instead of complex syntax, easily reversible operations, and rapid visual feedback on the object of interest for users' actions, as some of the characteristics of direct manipulation interfaces. He compares direct manipulation with driving a car, in which common, well-known actions such as rotating the steering wheel is enough to change the direction of the car and result in immediate visual feedback by changing the scene outside the windows. Direct manipulation results in the closeness of the problem domain (object of interest) to the command action. Some of the benefits of direct manipulation are

- Novices can learn basic functionality quickly, usually through a demonstration by a more experienced user.
- Experts can work very rapidly creating new features or functions.
- Error messages are rarely needed.
- Users can easily see if their action is taking them closer to their goal, and if not, they can easily reverse it.
- Users feel confident, because they feel in control and can predict system's response.

Hutchins et al. (1985) take a comprehensive look at the cognitive aspects of direct manipulation. Their assumption is that the directness of the interface results in less cognitive load for the user. They name *distance* and *engagement* as two aspects of directness. The former refers to the distance between the user's thoughts and the requirements of the system. In other words the translation between the user's thoughts and goals and the system's actions is straightforward, both in the input and the output level. The latter refers to the qualitative feeling for the user that (s)he is directly manipulating the objects, not through the program or the computer.

The intentions of the user somehow gets translated into the low-level language of the machine. This is referred to by *gulf of execution* (Hutchins et al., 1985). When most of the burden of covering the gulf of execution is left to the user, the sense of distance becomes larger. Environments with features such as syntax-directed editing reduce this distance (Lieberman & Fry, 1995).

The output of the program may also be distant from the intentions of the user. The *Gulf of evaluation* (Hutchins et al., 1985) refers to this distance and how much the programmer has to imagine the result to compare it with the intent. Debugging tools and visualization of results can be effective in bridging this gap (Lieberman & Fry, 1995).

Not all of these are necessarily possible to achieve in a programming system. Continuous representation of the result may not be possible, because the result may take place in the future or all the data may not be available at this time for the program to run. Effects of an action are abstract in programming, contrary to other forms of actions that produce immediate, concrete result. And the effects may be spread over time or space (Blackwell, 2002). Without direct manipulation, the benefits described above are no longer available to the users.

2.2.2 Programming notations

Each programming system has a programming notation. This notation can be text, diagrams, gesture, and so on. A notation is a way of representing the state of the world, and the correspondence between the representation and the world is based on convention. One notation may be better for one type of task than another, and they each have advantages and disadvantages. Choosing one notation over another for a programming system is a matter of trade-offs. The cognitive dimensions of notations offers a framework for evaluating programming notations.

2.2.3 Cognitive Dimensions of Notations framework

The cognitive dimensions of notations framework (Green, 1989) provides a vocabulary for discussing design problems, developed by cognitive psychology researchers for system developers. The framework was originally put forward by Thomas Green as a tool for evaluating usability of information artifacts, from a telephone to a programming language. The purpose of developing this framework is to give names to concepts that designers have noticed and trade-offs they make in designing a system and to allow them to discuss them and their consequences. Here are the original cognitive dimensions, adapted from Green (2000).

- Visibility: Ability to View Components Easily.
  Systems that bury information in encapsulations reduce visibility. Since examples are important for problem-solving, such systems are to be deprecated for exploratory activities; likewise, if consistency of transcription is to be maintained, high visibility may be needed. An important component is juxtaposability, the ability to see any two portions of the program on screen side-by-side at the same time.

- Hidden Dependencies: Important Links between Entities Are Not Visible.
  If one entity cites another entity, which in turn cites a third, changing the value of the third entity may have unexpected repercussions. Examples: cells of spreadsheets; style definitions in Word; complex class hierarchies; HTML links.

- Role-Expressiveness: The Purpose of an Entity Is Readily Inferred.
  How easy is it to answer the question "What is this bit for?" How easy is it to "read" the programming language? A role-expressive notation makes it easy to discover why the programmer or composer has built the structure in a particular way.

- Abstraction: Types and Availability of Abstraction Mechanisms.
  Abstractions (redefinitions) change the underlying notation. Macros, data structures, global find-and-replace commands, quick-dial telephone codes, and word-processor styles are all abstractions. Some are persistent, some are transient. Systems that allow many abstractions are potentially difficult to learn.

- Secondary Notation: Extra Information in Means Other Than Formal Syntax.
  Users often need to record things that have not been anticipated by the notation designer. Rather than anticipating every possible user requirement, many systems

support secondary notations that can be used however the user likes. One example is comments in a programming language, another is the use of colors or format choices to indicate information additional to the content of the text.

- Closeness of Mapping: Closeness of Representation to Domain.
  How closely related is the notation (the programming world) to the domain it is describing (the problem world)?

- Consistency: Similar Semantics Are Expressed in Similar Syntactic Forms.
  Users often infer the structure of information artifacts from patterns in notation. If similar information is obscured by presenting it in different ways, usability is compromised.

- Hard Mental Operations: High Demand on Cognitive Resources.
  A notation can make things complex or difficult to work out in your head, by making inordinate demands on working memory, or requiring deeply nested goal structures.

- Viscosity : Resistance to Change.
  A viscous system needs many user actions to accomplish one goal. Changing all headings to upper-case may need one action per heading. (Environments containing suitable abstractions can reduce viscosity.) The authors distinguish repetition viscosity, many actions of the same type, from knock-on viscosity, where further actions are required to restore consistency.

- Premature Commitment : Constraints on the Order of Doing Things.
  Do programmers have to make decisions before they have the information they need? Examples: being forced to declare identifiers too soon; choosing a search path down a decision tree; having to select your cutlery before you choose your food.

- Error-Proneness: The Notation Invites Mistakes and the System Gives Little Protection.
  Enough is known about the cognitive psychology of slips and errors to predict that certain notations will invite them. Prevention (e.g. check digits, declarations of identifiers, etc) can redeem the problem.

- Diffuseness: Verbosity of Language.

Some notations can be annoyingly long-winded, or occupy too much valuable real-estate within a display area. Big icons and long words reduce the available working area.

- Provisionality: Degree of Commitment to Actions or Marks.
  Is it possible to make provisional actions, such as recording potential design options, sketching, or playing what-if games?

- Progressive Evaluation: Work-to-Date Can Be Checked at Any Time.
  Evaluation is an important part of the design process, and notational systems can facilitate evaluation by allowing users to stop in the middle of a task to check work so far, find out how much progress has been made, or check what stage in the work they are up to.

In order to use the framework, we need to ask whether the system supports the user's intended activities. Green (2000) categorizes user activities as incrementation, transcription, modification, exploratory design, searching, and exploratory understanding. Each generic activity requires only some of the dimensions. For example, in a system designed for exploratory design, viscosity (resistance to change) and premature commitment have to be low, and visibility (of system components) and role-expressiveness (what each entity does) must be high to support designers changes, from detailed tweaks to fundamental changes. Improving the system in one dimension may have a reverse effect on another dimension. These are *trade-offs* that designers choose to make, based on the requirements of the system and the type of tasks it will support. For example, increasing the visibility and role-expressiveness of a system by displaying more items on the screen, may result in a system that is very diffuse.

Other researchers have suggested new dimensions to be added to the framework, including the following (Blackwell et al., 2001).

- Creative Ambiguity: The extent to which a notation encourages or enables the user to see something different when looking at it a second time.

- Specificity: The notation uses elements that have a limited number of potential meanings rather than a wide range of conventional uses.

- Detail in Context: It is possible to see how elements relate to others within the same notational layer, and it is possible to move between them with sensible transitions.

- Indexing: The notation includes elements to help the user find specific parts.
- Synopsis: The notation provides an understanding of the whole when you 'stand back and look'.
- Free Rides: New information is generated as a result of following the notational rules.
- Useful Awkwardness: Its not always good to be able to do things easily. Awkward interfaces can force the user to reflect on the task, with an overall gain in efficiency.
- Unevenness: Because things are easy to do, the system pushes your ideas in a certain direction.
- Lability: The notation changes shape easily.
- Permissiveness: The notation allows several different ways of doing things.

2.3 End-user programming

A *professional programmer* is someone whose primary job function is to write or maintain software. A *novice programmer* is someone who is learning to program. An *end-user programmer* however, is someone who writes programs only to achieve their main goal, which is not programming in itself, but something else such as accounting, engineering, designing and analyzing buildings, and music composition (Nardi, 1993; Jones, 1995; Myers et al., 2007). Although most end-user programmers use special-purpose languages specific to their domain, some use general-purpose languages such as Java or C (Myers et al., 2006).

Most programs today are written by end-user programmers, not professional programmers (Ko et al., 2011). In fact, the number of end-user programmers is much higher than professional programmers, roughly 55 million to 3 million in 2012 (Scaffidi et al., 2005). Where a programmer's goal is to produce code for others to use, code reuse is not a high priority for end-user programmers. For example, a teacher may program a spreadsheet to calculate her grades, or an architect may write code to create a complex geometric model. Sometimes in an organization, one employee becomes the go-to person for others when they need to program their application (Gantt & Nardi, Gantt & Nardi). This person gradually leaves the end-user programming category, because the goal of his job now is to produce reusable code for others to use. Nardi & Miller (1991) identify three types of software expertise in organizations. *End users* are professionals in their domain with little knowledge or interest in programming. *Local developers* are knowledgeable in both domain and programming and play the role of tutors and developers for the end users. *Programmers* are those with professional training in computing and support both end users and local developers.

2.3.1 End-user programming barriers

Ko et al. (2004) (also (Ko, 2007)) identify six learning barriers that end-user programmers face when learning a new system.

- Design Barriers: *"I don't know what I want the computer to do..."*
  These are inherent cognitive difficulties of a programming problem, regardless of the programming notation used. Examples are conditionals, sorting, and event concurrency.

- Selection Barriers: *"I think I know what I want the computer to do, but I don't know what to use..."*
  These are an environment's facilities for finding what programming interfaces are capable of a particular behaviour.

- Coordination Barriers: *"I think I know what things to use, but I don't know how to make them work together..."*
  These are *"invisible rules"* of the system for combining different language and library interfaces.

- Use Barriers: *"I think I know what to use, but I don't know how to use it..."*
  These barriers happen when the interface obscures 1) in what way something can be used, 2) how to use it, and 3) what effect such uses will have.

- Understanding Barriers: *"I thought I knew how to use this, but it didn't do what I expected..."*
  These are related to the program's external behaviour that obscures what did or did not happen at compile or run time.

- Information Barriers: *"I think I know why it didn't do what I expected, but I don't know how to fix it..."*

These barriers occur when it is difficult to find information about the program's internal behaviour, such as variable values and what calls what.

In a series of studies of professional and end-user programmers, Ko & Myers (2005b,a) discovered several contributing factors to the programming problems.

- What end users want their programs to do, can sometimes be more difficult than the code they need to write, for example sorting algorithms. Also, sometimes end-users expect the system to do something that is beyond its scope, instead of moving to another, more suitable system.
- Fundamental cognitive biases can cause end users to introduce errors into code. Here are some examples. We tend to collect only enough information needed to make a decision, not the best decision. We tend to favour immediate, less useful feedback over delayed, but more useful ones. We tend to use tools that will help us reach our short-term goals, regardless of their effect on our long-term goals.
- The languages, environments, libraries, and tools used by end users to create their programs can cause significant difficulties for them.
- The code that end users create is usually not fully understood by them, either because they did not write it entirely on their own, or because they do not understand how it is interpreted by the machine.
- The errors in the code that end users create can cause additional errors to occur.

"The fact that end-user programmers are motivated by their domain and not by the merits of producing high-quality, dependable code, means that most of the barriers that end users encounter in the process of writing a program are perceived as distractions." (Ko, 2007)

Facing barriers and distractions require extra attention for the end-user programmers. This is described by Blackwell in the attention investment model.

2.3.2 Attention investment model

The attention investment model (Blackwell, 2002) is an attempt to explain the rationale behind users' decisions when it comes to programming activity. Attention or concentration is much like a currency. Each programming action requires a certain amount of attention currency from the user, called cost, to get it done. In return, it saves the user some attention units in the future, called pay-off, as a result of automation or external reward. However, it is probable that no pay-off will occur or more cost will be necessary. This is called risk. The user makes a quick analysis of the cost, pay-off and risks involved in a task and decides whether to use direct manipulation of GUI or write a program. At any time during the task user may realize that it is getting too costly or risky and change his/her mind.

Blackwell & Burnett (2002) use this model in designing Form/3, an end-user programming add-on to Excel. In designing new features in Form/3, they always refer to the attention costs before making a design decision. For example, they argue that a new feature should look and work in a way that allows the users to reapply what they already know about working with Excel, instead of making attention investment in learning new things. Secondly, they acknowledge that the users should be free to make their own decisions about how and where to invest their attention. The system should not force them to choose options upfront before they have had a chance to work with them and make their own cost/benefit evaluations.

2.3.3 Gentle Slope Systems

MacLean et al. (1990) first introduced the tailorability of a system in a diagram, called the tailorability mountain (Figure 2.2). The diagram shows the amount of skills required to achieve a degree of tailoring power in using a system.

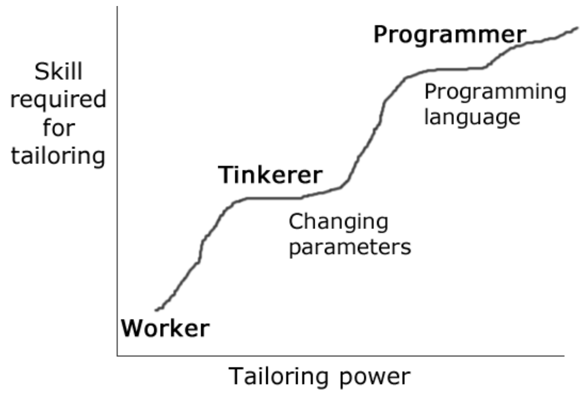


Figure 2.2: The tailorability mountain and its inhabitants (Adapted from: MacLean et al. (1990))

Lieberman et al. (2006) identify two types of end-user activities:

- *Parametrization or customization:* Activities that allow the user to choose between alternative behaviours already available in the system
- *Program creation and modification:* Activities that require creating a software artifact from scratch or modify an existing one, including macros, scripting, visual programming, and programming by example.

End-user programming involves mostly the second set of activities. It is best to aim for a *"gentle slope"* for going from the first set to the second set of activities.

Dertouzos (1992) and later Myers et al. (1992) introduce the concept of a *gentle slope system* (See Figure 2.3). In such systems, it is easy to make small customizations, while more complicated ones only involve proportional amount of complexity and difficulty. In other words, to do a customization, the user only needs to learn a small number of features, and is not faced with a jump in complexity in order to achieve more tailoring power.

Some systems require a huge amount of learning before the user is able to get a task done. Often, the user hits a wall that (s)he needs to climb, before (s)he can continue. As shown in Figure 2.3, spreadsheets are relatively easy to use up until the point when the user opens VBA to write a piece of code. That is when (s)he faces a steep learning curve because of the new programming language and lack of direct access to the spreadsheet interface.

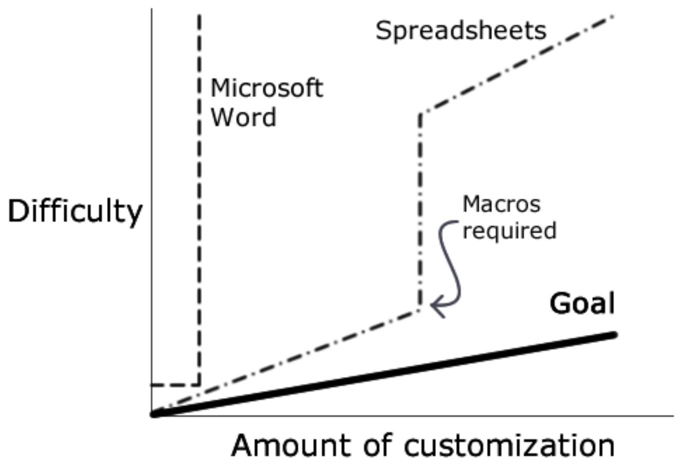


Figure 2.3: Gentle slope systems (Adapted from Myers et al. (1992))

2.4 Research threads

There has been a lot of research on programming environments and notations. End-user programming (EUP) has benefited from both the general purpose programming research and focused EUP research. In this section, I review some of these research threads that are relevant to my research.

2.4.1 Visual programming

Burnett (1999) defines visual programming as programming with more than one dimension used to convey meaning. Spatial relationships, multidimensional objects, and time are examples of these additional dimensions. The goal of visual programming is not to eliminate text, but to make programming more understandable for users and to improve the correctness and speed of their work. This is done by providing simplicity of concepts, concreteness of process, explicit depiction of relationships, and immediate visual feedback.

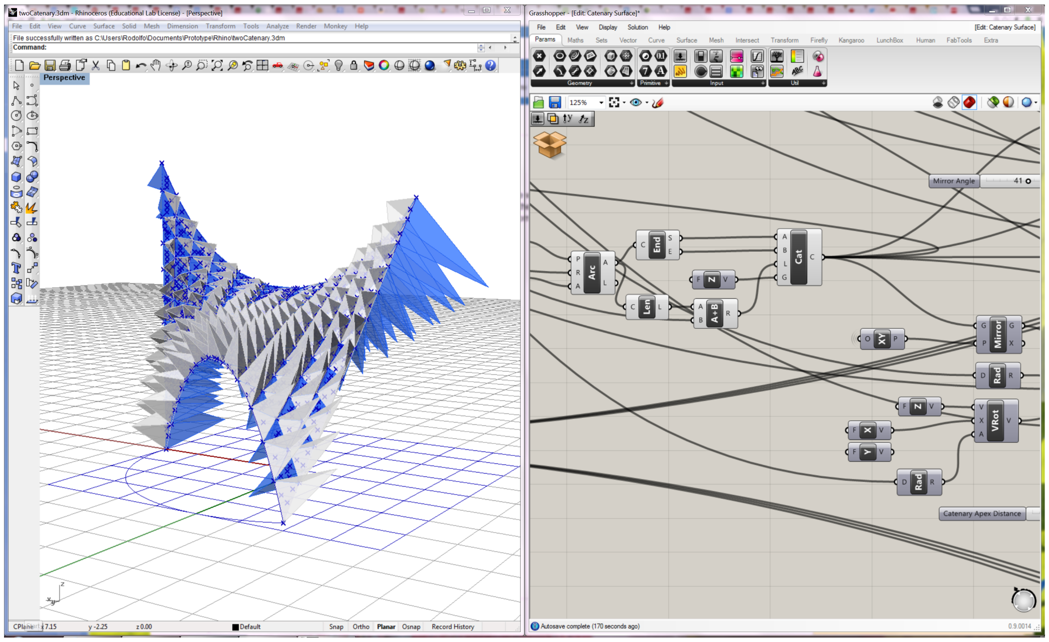
Visual programming has its own issues and has caused heated arguments among researchers about its usability. One of its biggest problems is scalability. The scaling up problem comprises representations issues, programming language issues, and issues that are beyond coding (Burnett et al., 1995). Another issue of visual programming is whether a given graphical notation has the same meaning for all users or do experts and novices read it differently (Petre, 1995). The evidence against visual programming also includes the match-mismatch of notation and task. One type of notation may work for one type of task and not others. Also, individual differences between users play an important role in the success of a visual programming language, as some people are better in comprehending visuals than others (Whitley, 1997).

There has been much research done on visual programming languages (VPLs) and environments (VPEs). This has resulted in several different paradigms and styles of visual programming. We can classify VPLs based on their programming paradigm. Some of the most important VPL paradigms are constraint-based VPLs, data-flow VPLs, form-based and spreadsheet-based VPLs, imperative VPLs, and rule-based VPLs (Burnett & Baker, 1994). Costagolia et al. (2002) categorize VPLs based on the type of graphical notation they use. The *Graph* class of VPLs are those that consist of boxes and arrows connecting them. The *Plex* class covers the ones with flowchart style graphs with objects that have predefined number of attaching points. The *Box* class is mostly used for geographical systems and consists of boxes that represent objects and spatial rules such as inclusion and intersection that represent queries. The *Iconic* class of VPLs uses icons for objects. And finally the *Hybrid* class uses features of two or more of above classes.

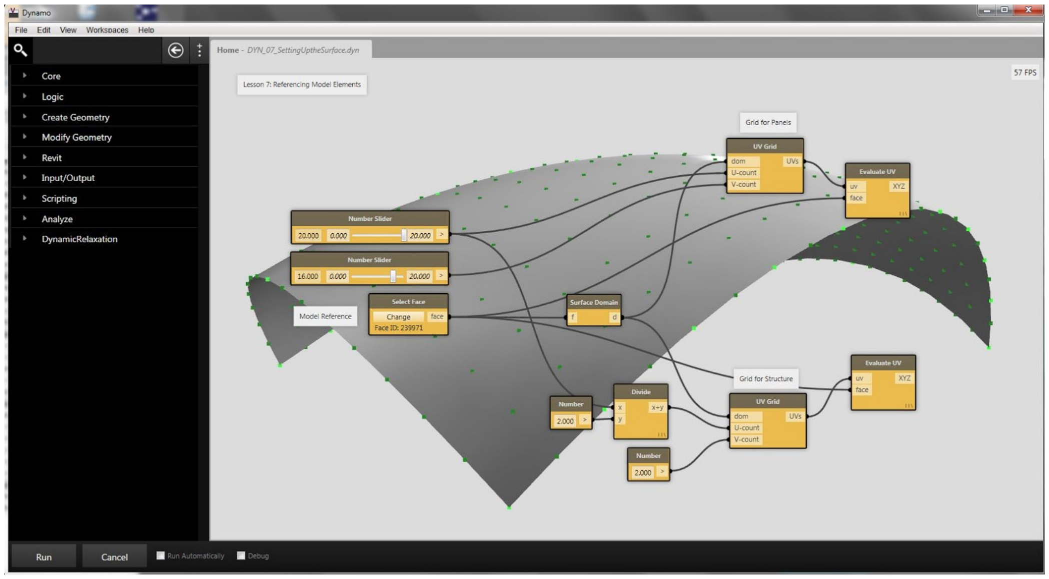
Data-flow visual programming languages (DFVPLs) are among the most popular VPL paradigms. Data-flow languages are based upon the notion of data flowing from an entity to another (Davis & Keller, 1982). These languages are represented by directed graphs. The nodes of a data-flow graph represent primitive instructions such as arithmetic or comparison operations. The arcs shows the data dependency between the nodes (Kosinski, 1973). When a node's arguments become available, it becomes firable. When a node is fired, the operation inside the node is applied to its inputs and the result or output is sent to other nodes that need it. Nodes with no incoming dependencies can be executed independently and therefore concurrently (Davis & Keller, 1982).

Data-flow languages had been around for a while but the availability of cheap graphical hardware in 1990s caused advances in data-flow visual language design (Johnston et al., 2004). As visual programming was gaining popularity, the data-flow paradigm seemed to be represented very well graphically. In addition, as pointed out by Baroth & Hartsough (1995), research shows that most developers think in terms of data-flow, so DFVPLs remove the paradigm shift that they experience in textual programming languages. One of the most important reasons for the success of DFVPLs in the industry was that it was easy for non-programmers especially clients to follow the data-flow diagram and communicate with the developer about the code (Baroth & Hartsough, 1995). In general, DFVPLs tend to blur the distinction between the language and the environment, as well as the separation between design, coding and testing of the code. This blurring provides a tool for rapid prototyping (Johnston et al., 2004).

Visual programming has found its way into CAD systems and has recently become quite popular (Figure 2.4). Grasshopper is a visual programming environment for McNeel's Rhino3D (McNeel, 2010). Similarly, Dynamo is a VPE for Autodesk's Vasari and Revit (Autodesk, 2013). The visual programming environment in Grasshopper is a separate window from the model view. Dynamo has the option of displaying the model in the background of the visual programming window. In both systems, when the user modifies the program, the 3D model updates dynamically, but when (s)he manipulates the model, using Rhino or Revit's commands and toolbars, Grasshopper and Dynamo do not reflect that change.



(a) The Grasshopper interface on the right, with the Rhino model view on the left (Credit: Rodolfo Sanchez)



(b) The Dynamo interface with the 3D model in the background (Credit: Nathan Miller)

Figure 2.4: Visual programming in CAD systems

2.4.2 Programming by demonstration

"If the user knows how to perform a task on the computer, that should be sufficient to create a program to do the task." (Cypher & Halbert, 1993, p. 1). That is the idea behind Programming By Demonstration (PBD) research. The computer is supposed to *"watch what I do"* and create a program from it. Macros are basic implementation of "watch what I do". The user starts the recording, performs a sequence of actions, then stops the recording. The macro when played, performs the same sequence of actions. The limitation of macro recorders is that they are too literal. They repeat the exact mouse and keyboard strokes. So if the user needs to perform a similar action on different datasets for example, the macro would not be useful.

Programming by demonstration (or programming by example) has the additional feature of generalizing user's actions. For example, a spreadsheet macro recording of an action performed on cells A1-A20 will always repeat the same action on the same cells. But a PBD program will be applicable to any list of cells, such as A1-A100 or on "all employee salaries" data. In other words, the user works on a concrete example, usually using direct manipulation, and then the program is generalized by the computer. An important challenge in programming by demonstration is inferring intent from user's actions. It significantly affects how the program is generalized. There are two types of PBD systems when it comes to generalization, the ones that *guess* and the ones that do not (Myers, 1992). The first group of systems use inferencing to generalize the program. The second group require the user to define the generalizations, therefore ensuring that the program will do what the user intends it to.

Demonstrational interfaces maintain the benefits of direct manipulation interfaces by allowing the user to work on concrete examples. Users can see the result of their actions and fix any problems that may occur. Beginners can create programs even without knowing any programming and experts can work more rapidly using direct manipulation. The problem with PBD interfaces is mainly with the ones that guess, since it is possible that they guess incorrectly. The process of reviewing the inferred program can slow down user's work.

Form/3 is a spreadsheet language that allows functions to be created by direct manipulation of spreadsheets (Burnett et al., 2001). The system then generalizes the concrete example into an abstract function. As claimed by the authors, Form/3 is a gentle slope system that extends the capabilities of spreadsheets without the user leaving the spreadsheet interface. The benefit of PBD in Form/3 is that the user maintains a sense of directness, in that the interaction happens directly with the concrete example. Additionally, having the example provides an immediate feedback to the user, therefore minimizing errors and unintended results.

2.4.3 Natural programming languages

Programming involves translating a mental plan into one that is compatible with the computer (Hoc & Nguyen Xuan, 1990). The closer the computer language is to the programmer's mental plan, the easier the translation will be. Natural programming languages are ones that are close to the way users think about the program (Myers et al., 2004).

Directness has been one of the principles of good user interface design since Shneiderman (1983) introduced the concept of direct manipulation. Nelison's recommendation that user interface should speak the user's language (Nielsen, 1994) and Green's closeness of mapping dimension (Green, 1989) both emphasize the importance of direct mapping between the language and the way people usually think about the problem. This results in *cognitive directness* (Hix, 1993), minimizing the cognitive transformation users have to make.

Pane & Myers (2006) developed HANDS, a natural programming environment for beginner programmers. The HANDS system uses the concept of someone sitting at a table, playing with cards. The cards replace variables and automatically take care of scope, size, naming and so on. HANDS is event-based, which is close to the way people think. Evaluations of HANDS shows a gentle learning slope for children. Lauria et al. (2002) used natural programming as a way for users to communicate and direct robots. Instead of writing a program, the users simply instructed the robot to move in one direction or take a route.

Miller (1981) studied what *'natural'* means for different people by asking them to describe a procedural algorithm. They discovered that what is natural for one group in one context is not necessarily natural for others and in a different context. Natural programming languages are specific to the domain, age group, and background of the users.

2.4.4 Live programming

The phrase *"live programming"* (also known as *"live coding"*) is used in two different contexts. The first one refers to the practice of programming (or coding) live in front of an audience. In the context of a lecture or classroom, the lecturer creates a program on the fly to teach the audience how it is done (Rubin, 2013). In the world of music and video, the artist creates and manipulates the program in front of the audience to produce an improvised piece of music or video (Blackwell & Collins, 2005). The code is usually also projected on the screen as part of the show.

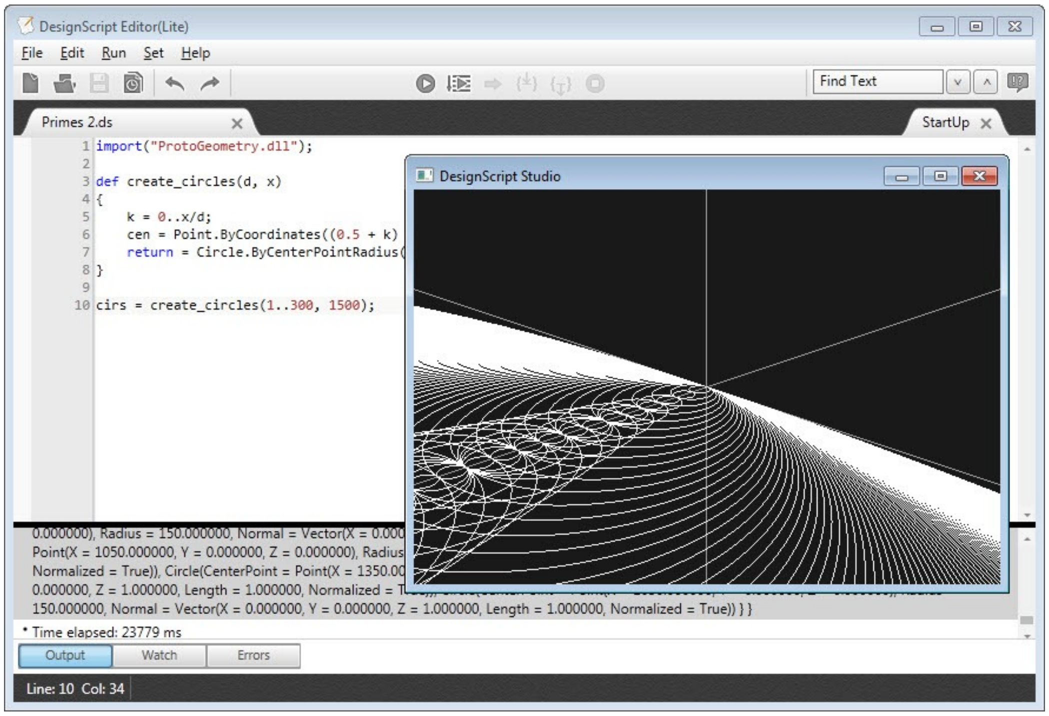
The second use of the phrase "live programming", which is the one I use in this document, refers to programming in an environment that displays the result of the program as soon as possible without the need for the user to run or compile the code. Tanimoto (1990) first coined the term *"liveness"* to indicate the immediacy of automatic semantic feedback during programming. He described four levels of liveness: Level 1 with no feedback, Level 2 with limited, non-automatic feedback, Level 3 with automatic feedback for incremental program edits, and finally Level 4 with level 3 liveness as well as responding to other events such as clock and mouse clicks.

The notion of live coding is beginning to find its way into programming applications (Burnett et al., 1998; Burg et al., 2013). Examples of live coding environments are 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 a one-way liveness from code to graphics. 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 (Aish, 2012) (Figure 2.5) and Rhino's add-on, Yeti (Davis, 2011), are two recently developed live programming environments in CAD systems. They both have a one-way liveness from the script to the CAD model.

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.

There are pros and cons to live coding. A benefit of having immediate feedback to coding actions is that the programmer can immediately see the problems and fix them. In the event of a syntax error, the system would not allow the programmer to continue until the error is fixed. In design domains, designers and creators need immediate connection to what they are creating, and programming environments can facilitate or take away that connection.



```
import("ProtoGeometry.dll");

def create_circles(d, x)
{
    k = 0..x/d;
    cen = Point.ByCoordinates((0.5 + k)
    return = Circle.ByCenterPointRadius(
}

cirs = create_circles(1..300, 1500);
```

```
0.000000), Radius = 150.000000, Normal = Vector(X = 0.000
Point(X = 1050.000000, Y = 0.000000, Z = 0.000000), Radius
Normalized = True)), Circle(CenterPoint = Point(X = 1350.00
0.000000, Z = 1.000000, Length = 1.000000, Normalized = T
150.000000, Normal = Vector(X = 0.000000, Y = 0.000000, Z = 1.000000, Length = 1.000000, Normalized = True)) } }
* Time elapsed: 23779 ms
```

Figure 2.5: DesignScript programming language, running on AutoCAD (Credit: Kean Walmsley)

According to Victor (2012), immediate visual feedback to programming actions shows the effect of those actions to the designers, letting them react to it right away. Live-coding.io (Florit, 2013) is a web-based application that promotes live coding as a way of *"sketching with code"*. On the other hand, live interfaces can be expensive. Burnett et al. (1998) offer several low-cost approaches to implementing a live interface.

2.4.5 Spreadsheets

Spreadsheets are the most widely used programming languages mostly due to their usability and the fact that they do not look like a programming language. They are good examples of end-user programming environments that bring programming directly to the problem domain. In spreadsheets, the problem domain is the cells and numbers in them. Users can write programming statements directly into the cells and see immediate feedback in the form of numbers. Figure 2.6 shows the Trace Precedent feature in an Excel spreadsheet. This is an equivalent of the dependency graph in any dataflow programming language that shows the flow of data in the program. The difference is that in Excel, this graph is displayed in the problem domain using the cells and nodes, instead of in a separate node-link diagram.

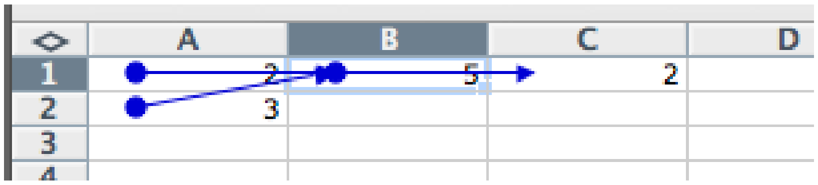
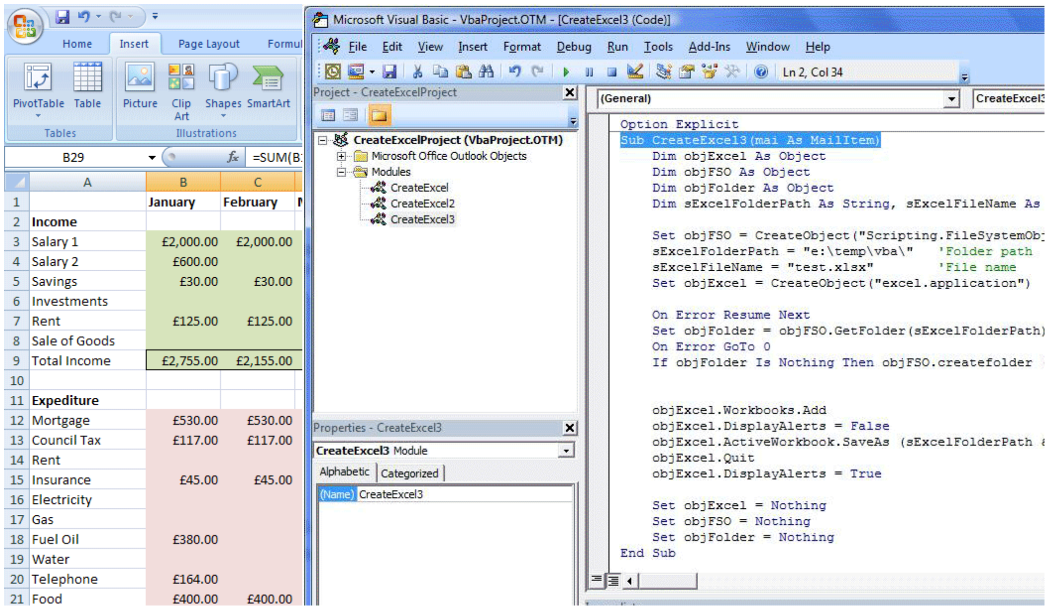


Figure 2.6: Trace dependencies in Excel

However, as most end-user programming systems, Excel fails to keep this closeness of mapping in regards to its use of Visual Basic for Applications (VBA). VBA is a textual programming environment that is used by Excel users to write and edit macros and programs to do more complicated tasks that are not possible in Excel interface. Figure 2.7 shows an Excel spreadsheet on the left and a macro written in VBA on the right. During the process of writing the code, the user goes back and forth between Excel and VBA in order to find the right cells and operations required for the task and also to test the code and see the result.



	A	B	C
1		January	February
2	Income		
3	Salary 1	£2,000.00	£2,000.00
4	Salary 2	£600.00	
5	Savings	£30.00	£30.00
6	Investments		
7	Rent	£125.00	£125.00
8	Sale of Goods		
9	Total Income	£2,755.00	£2,155.00
10			
11	Expediture		
12	Mortgage	£530.00	£530.00
13	Council Tax	£117.00	£117.00
14	Rent		
15	Insurance	£45.00	£45.00
16	Electricity		
17	Gas		
18	Fuel Oil	£380.00	
19	Water		
20	Telephone	£164.00	
21	Food	£400.00	£400.00

```
Option Explicit
Sub CreateExcel3(mai As MailItem)
    Dim objExcel As Object
    Dim objFSO As Object
    Dim objFolder As Object
    Dim sExcelFolderPath As String, sExcelFileName As

    Set objFSO = CreateObject("Scripting.FileSystemObj
    sExcelFolderPath = "e:\temp\vba\"    'Folder path
    sExcelFileName = "test.xlsx"         'File name
    Set objExcel = CreateObject("excel.application")

    On Error Resume Next
    Set objFolder = objFSO.GetFolder(sExcelFolderPath)
    On Error GoTo 0
    If objFolder Is Nothing Then objFSO.createfolder


    objExcel.Workbooks.Add
    objExcel.DisplayAlerts = False
    objExcel.ActiveWorkbook.SaveAs (sExcelFolderPath &
    objExcel.Quit
    objExcel.DisplayAlerts = True

    Set objExcel = Nothing
    Set objFSO = Nothing
    Set objFolder = Nothing
End Sub
```

Figure 2.7: A view of Excel and Visual Basic shows the different notations of these two environments

Jones et al. (2003) argue that existing functions in Visual Basic do not work because of their cost: the programming paradigm is different between Excel and VB, the notation is different, the programming environment is different and more complex in Visual Studio, and debugging is different and less accessible in VB as opposed to Excel's continuous display of values.

As mentioned before, Form/3 is a *"gentle slope language"* based on the spreadsheet paradigm developed by Margaret Burnett and her colleagues (Burnett et al., 2001; Jones et al., 2003). In Form/3 users create functions by directly working in the spreadsheet. The system then generalizes the program into a function. Creators of Form/3 use cognitive dimensions of notations framework and the attention investment model to guide them in the design process. They keep the functions interface consistent with Excel's worksheet, so their target users can use their prior knowledge in working with Form/3. It also decreases the cos/benefit ratio of programming in the attention investment model (Blackwell & Burnett, 2002), because it sends the message to the users that functions are similar to ordinary worksheets and there is little cognitive cost in using them. To minimize hidden dependencies and maximize role expressiveness dimensions, they display dependencies not only in one worksheet like the way Excel does it, but between multiple worksheets in order to show the user which function is called in which cell.

Form/3 does not require premature commitment from the users by asking them to write a function from scratch (Burnett et al., 2002). Rather, they allow them to create a function from an existing formula or a range of cells. This is important because end-user programmers usually decide to create a function after they create a piece of program. The system then makes a guess about the parameters, and the user can correct the guess if necessary. Form/3 uses Excel's multiple windows and tabs to display function sheets and worksheets at the same time, in order to help the user keep track of the changes in the worksheet when editing the functions.

2.4.6 Introductory programming languages and environments

General purpose programming languages are not made with the purpose of teaching programming to beginners. It is much easier for people to learn basic concepts of programming in an environment specifically designed for teaching programming. There are two groups of beginner programmers: those who want to learn programming for its own sake (for example, to pursue a career as a programmer) and those who want to learn to program to accomplish another goal, such as automating their work (end-user programmers). Research in introductory programming languages and environments covers common issues faced by both groups of beginners. With a broad brush, we can divide these problems into syntactic and semantic related problems. Here I present a summary of the solutions and guidelines presented in the introductory programming literature (Nielsen, 1994; Pane & Myers, 1996; McIver & Conway, 1996; Kelleher & Pausch, 2005; Pears et al., 2007).

Beginner programmers have problem translating their intentions into the syntax of the program. Research in this area suggests different methods and techniques to minimize the problem of syntax in introductory languages, including the following.

- Simplifying the programming language by avoiding confusing syntax, punctuation, and redundant constructs.
- Creating programs by direct manipulation, by choosing from menus, and by graphical or physical objects
- Making programming more concrete by using programming by modification of examples (templates).
- Providing multiple ways of doing things and letting the user choose the appropriate one for the task.
- Maintaining internal consistency of the system, so that similar things mean the same throughout the system.
- Choosing appropriate notation for the task, whether it is textual or visual. Not all notations are good for all types of tasks.
- Maintaining naturalness of the language, both in the way people express the program in their natural language and in the way they think about the program.
- Preventing syntax errors by making them detectable by the system. Syntax errors should not cause semantic errors.

Beginner programmers also have difficulty structuring the program and understanding how it works. The following techniques help with semantic problems of beginner programmers.

- Maintaining consistency with the domain world and user's external knowledge.

- Making things visible on the screen to avoid working memory load and minimize hidden dependencies. Choosing an appropriate level of abstraction.
- Highlighting semantically important entities that may be obscured in the interface.
- Minimizing the need for premature commitment by allowing the user to postpone decisions until necessary information is available
- Minimizing viscosity of the system. Beginners do not always know what they want or need to do from the beginning and the system should not demand too much effort to change something in the program.
- Reducing working memory by allowing recognition rather than recall whenever possible.
- Encourage incremental running and testing of the program and give immediate feedback
- Allowing tracking of program execution and state of variables.

Most children's programming environments have successfully adopted these techniques for teaching programming to children. Mini-languages (Brusilovsky et al., 1997) are small and simple programming languages with an actor acting in a mini world. Most mini languages have basic control structures such as conditional execution, looping, and recursion. Some examples of mini-languages are Logo, Karel the Robot, Karel-3D, Marta, Josef the Robot, and Tortoise. These languages have a small syntax and simple semantics. So students can spend their time on learning algorithmic thinking and program design instead of dealing with complicated syntax and semantics. They use engaging and interesting metaphors such as Logos and robots. They have high visibility that helps in exploratory learning of the language.

ALVIS Live! is an example of a novice programming system that uses direct manipulation strategies to familiarize novice programmers with programming concepts. User studies show that ALVIS Live! not only helps novices to better perform programming tasks, but also positively affects their ability to transfer to textual interfaces (Hundhausen et al., 2006).

2.4.7 End-user software engineering

The number of end-user programmers is growing every year. These domain experts may not be professional programmers, but the programs they write, unless they are strictly for fun, can have serious impact on their lives and works. Errors in end-user programs can cause significant damage to their work, for example, incorrect accounting spreadsheets may result in financial loss to the business. The quality and dependability of these programs are the focus of end-user software engineering research (Burnett, 2009; Myers et al., 2009).

Due to the difference in background, motivation, and interest between end-user programmers and professional programmers, it is not enough to give them the same tools and techniques as software engineers. End-user programming systems should support end-users beyond the programming stage, while keeping their specific goals and motivations into consideration (Burnett et al., 2004). For end-user programmers, the code itself holds little value. Rather, it is the goal of the task that is important. Testing the code is not a priority for end-user programmers, so it is unreasonable to expect them to spend as much time and focus on testing as professional programmers do. End-user programming systems should incorporate necessary software engineering activities into end-users' workflow without taking their focus away from their task (Ko et al., 2011). For example, the system can present and verify the success or failure of the program incrementally by giving the end-user iterative feedback.

End-user software engineering tries to bring the following software engineering techniques into end-user programming (Ko et al., 2011).

- What should my program do? – Requirements
- How should my program work? – Design and design specifications
- What can I use to write my program? – Reuse
- Is my program working correctly? – Verification and testing
- Why is my program not working? – Debugging

Another thing to consider is the wide range of people in variety of domains that engage in end-user programming. Taking into account Blackwell's (2002) attention investment model, end-user programmers weigh the cost and risks of the programming activity against its benefits. But different people have different perceptions of costs and risks. For example, a school teacher who writes spreadsheet programs for her class grades may see the above software engineering activities as a huge cost comparing to the benefit of the program for her work, while a scientist may see them as a cost worth the reward of getting accurate results in their experiments. This difference in perception should be taken into account when designing an end-user programming system.

2.5 Summary of chapter

In this chapter, I presented the CAD domain background, as well as the computer science background. Then I reviewed several different research topics that are relevant to my work. In the next chapter, I present my arguments and how it connects the literature with my research problem.

Chapter 3

Design rationale and methodology

The goal of this chapter is to explain the rationale behind the Programming In the Model (PIM) project and to present a brief overview of the design and prototyping process. First, I give a brief overview of Programming In the Model, in order to give context to the the discussion of PIM's goals and its design and implementation process (I present PIM in full detail in Chapter 4). Then, I present the guidelines I extracted from the literature and the consequent goals for the design of a CAD scripting environment, followed by a description of PIM's iterative ideation, implementation, and evaluation process. Also included in this chapter is a short summary of an early project and study I conducted on different interaction and visualization methods of *localized lists* in parametric CAD systems.

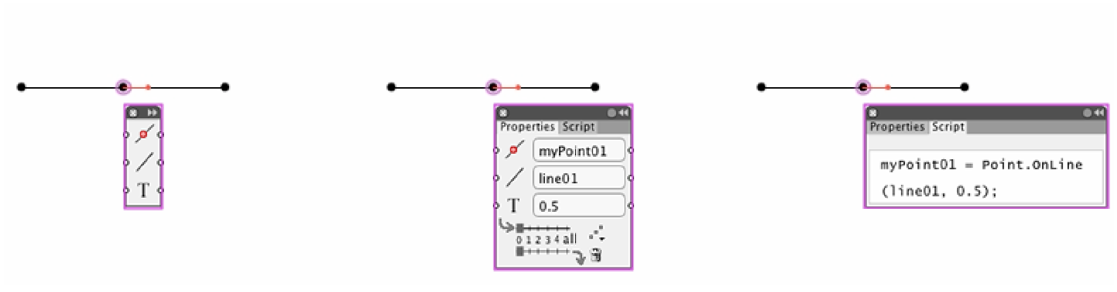
3.1 An overview of Programming In the Model

This is a brief overview of the final outcome of the project that consists of a set of techniques and features for creating a gentle slope scripting environment in CAD systems. I explain Programming In the Model features in detail in Chapter 4. These features can be grouped into three categories: *localization*, *liveness*, and *lookahead*, as described below.

Localization

In order to reduce the cognitive load caused by constant switching between different representations in a CAD interface (the model, the graph, and the script), I propose presenting the designers ***localized access to all the information about the object in the model***

view. For each object, different types of information can be accessed from within the model window, including inputs, replication, script, and dependencies (Figure 3.1). The user has the option to perform as much of the task as possible in any one these representations.



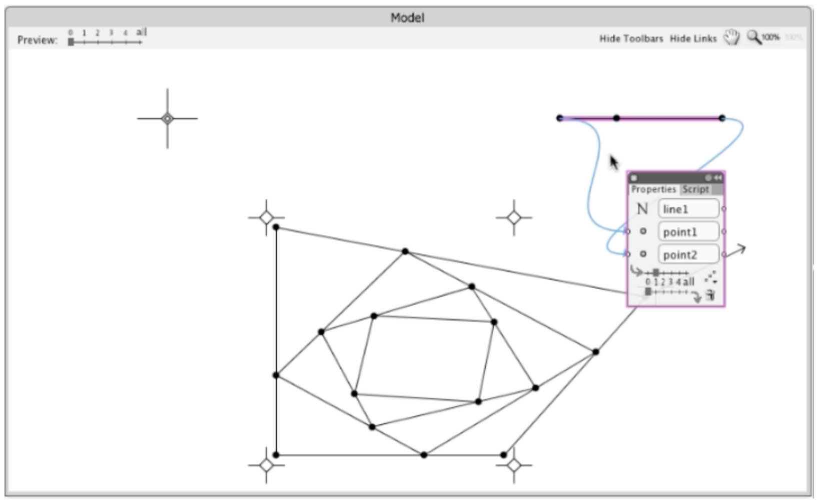
myPoint01

line01

0.5

```
myPoint01 = Point.OnLine
(line01, 0.5);
```

(a) Edit toolbar (b) Expanded edit toolbar (c) Script tab in the model



line1

point1

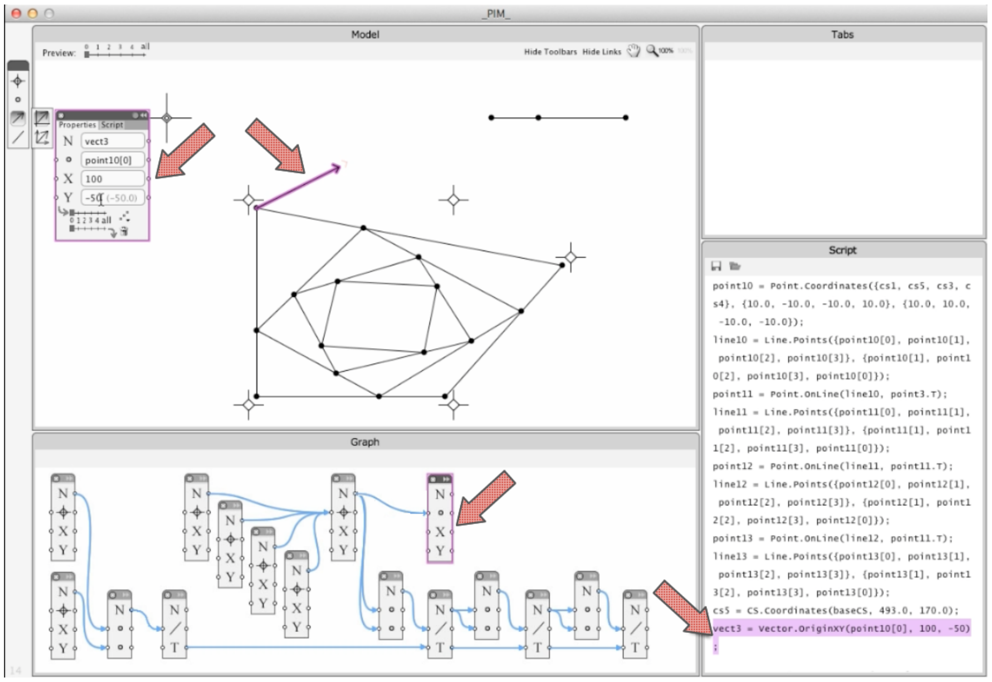
point2

(d) Localized representation of dependencies in the model view

Figure 3.1: Localized information in the model, including name and type of the object, inputs, corresponding scripts, and dependencies.

Liveness

Unlike most CAD systems that block access to the model during scripting, I propose keeping active all representations of the design side-by-side, including the model view, the dependency graph, and the script window (Figure 3.2). Further, aspects of one view should be shown and edited directly in others (localization). In this approach, all views are concurrent and interactive, so the designer can equally access the model and the graph during scripting. 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. The realtime 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.



```
N vect3
point10[0]
X 100
Y -50 (-50.0)
```

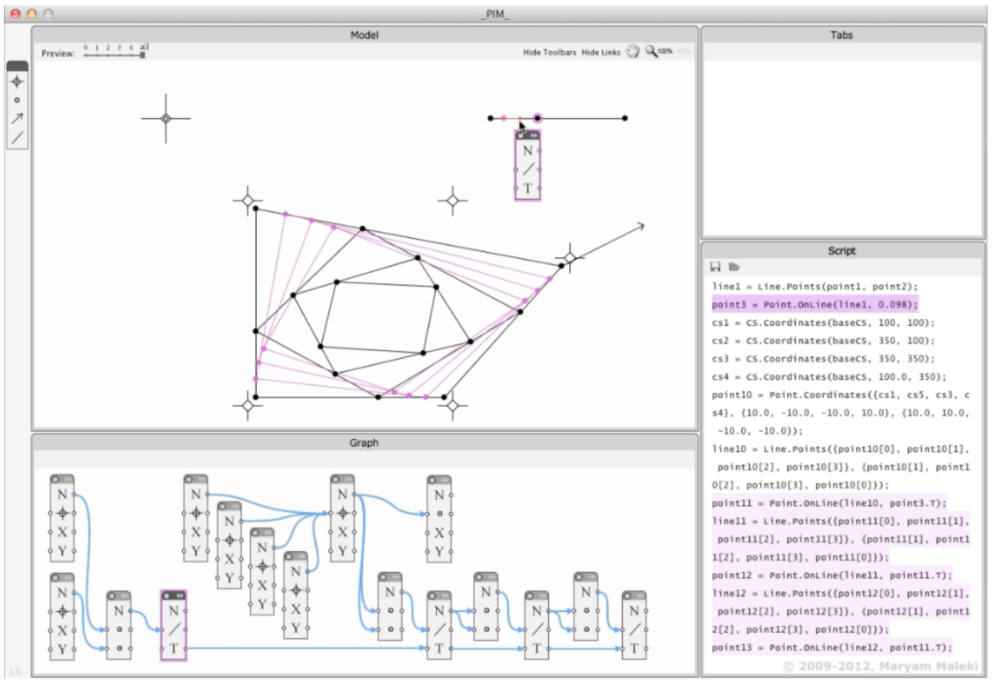
```
point10 = Point.Coordinates({cs1, cs5, cs3, c
s4}, {10.0, -10.0, -10.0, 10.0}, {10.0, 10.0,
 -10.0, -10.0});
line10 = Line.Points({point10[0], point10[1],
 point10[2], point10[3]}, {point10[1], point1
0[2], point10[3], point10[0]});
point11 = Point.OnLine(line10, point3.T);
line11 = Line.Points({point11[0], point11[1],
 point11[2], point11[3]}, {point11[1], point1
1[2], point11[3], point11[0]});
point12 = Point.OnLine(line11, point11.T);
line12 = Line.Points({point12[0], point12[1],
 point12[2], point12[3]}, {point12[1], point1
2[2], point12[3], point12[0]});
point13 = Point.OnLine(line12, point11.T);
line13 = Line.Points({point13[0], point13[1],
 point13[2], point13[3]}, {point13[1], point1
3[2], point13[3], point13[0]});
cs5 = CS.Coordinates(baseCS, 493.0, 170.0);
vect3 = Vector.OriginXY(point10[0], 100, -50)
;
```

Figure 3.2: Any modeling action is reflected in the script window to support users in their transition from modeling to scripting. On the other hand, the result of a scripting action is also reflected in the model and the graph windows, giving the user immediate feedback during coding. The user can switch between these representations, even in the middle of a task.

On the other hand, actions performed by direct manipulation of the model, using the graphical user interface, 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 immediately (something that separates this idea from macros), and (2) they can create code by modeling and the system will translate their actions into code. At any time during this process, the user can choose to work in the script, the model or the graph and the other two representations are updated in realtime. Because of the liveness of the interface and to help the users navigate these representations, the system offers ***brushing and highlighting*** of the data in the script, graph and model.

Lookahead

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. To overcome this barrier, we propose giving the designers a ***preview*** of what happens to the model as a whole and to each downstream object, if a change is applied to an object upstream (Figure 4.13). 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.



```
line1 = Line.Points(point1, point2);
point3 = Point.OnLine(line1, 0.098);
cs1 = CS.Coordinates(baseCS, 100, 100);
cs2 = CS.Coordinates(baseCS, 350, 100);
cs3 = CS.Coordinates(baseCS, 350, 350);
cs4 = CS.Coordinates(baseCS, 100.0, 350);
point10 = Point.Coordinates({cs1, cs5, cs3, c
s4}, {10.0, -10.0, -10.0, 10.0}, {10.0, 10.0,
 -10.0, -10.0});
line10 = Line.Points({point10[0], point10[1],
 point10[2], point10[3]}, {point10[1], point1
0[2], point10[3], point10[0]});
point11 = Point.OnLine(line10, point3.T);
line11 = Line.Points({point11[0], point11[1],
 point11[2], point11[3]}, {point11[1], point1
1[2], point11[3], point11[0]});
point12 = Point.OnLine(line11, point11.T);
line12 = Line.Points({point12[0], point12[1],
 point12[2], point12[3]}, {point12[1], point1
2[2], point12[3], point12[0]});
point13 = Point.OnLine(line12, point11.T);
```

Figure 3.3: Preview in PIM displays the edit alongside the original model and highlights all the objects that are affected by the change.

3.2 Design rationale

3.2.1 Programming in CAD

As described in section 2.2, Blackwell (2002, p. 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) and the sense of directness between their thoughts and the actions of the system (Hutchins et al., 1985).

My strategy: 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. My ideal environment gives the user 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; and dimensions and materials. The scripting language however, has a different notation. Depending on the system, functions and loops and conditionals; classes and instances; arguments, variables and types; and commas, semicolons, and brackets are used to write a program.

My strategy: I propose an environment that allows the user 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.

3.2.2 Barriers to programming in CAD

The concept of Gentle Slope Systems, described in Chapter 2.3.3 (Dertouzos, 1992; Myers et al., 1992), has huge implications for the parametric CAD domain. Systems such as Solidworks have a shallow curve because of their direct manipulation interface, but up to the point when the user opens the programming window (Visual Basic for Applications in Solidworks). Their pure textual programming notations are completely different from the modeling notation, with the scripting window blocking access to the model and its GUI tools, the result of which is a high step in their learning and use curve. Other parametric systems provide scripting languages specific to each system that are usually easier than the general purpose languages. There are some systems, such as GenerativeComponents, that have smaller steps in the curve by introducing programming earlier in the design process. The visual programming language in Grasshopper and Dynamo give the users most of the capabilities of scripting, but with a node-link interface. However, they still do not allow direct manipulation of the model during the visual programming and require the use of a general purpose programming language such as C# for customizations to the level of scripting languages.

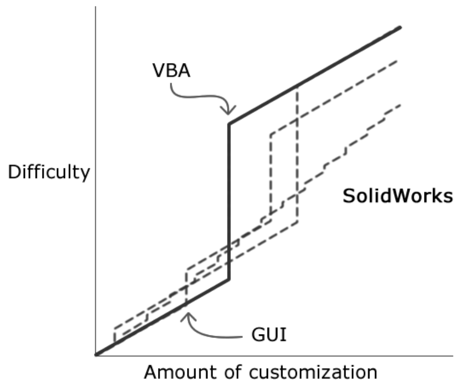
My strategy: I intend to create a more gentle slope system by breaking the process of learning and using scripting in CAD into small, reorderable steps. The ideal goal for a difficulty curve is to be as close to a horizontal line as possible. But given a system with its scripting and general purpose languages, we can only aim for a curve close to a slanting line, connecting each milestone to the next. Walls in the difficulty curve are much harder to overcome than several small steps ending in the same difficulty level. They also make the system look more difficult to learn and use than it is, resulting in a false perceived high cost/benefit ratio in the attention investment model.

Figure 3.4 charts difficulty versus customization in these parametric systems, assuming that all general purpose programming languages are equal in their difficulty level and achieve the same level of customization, and all scripting languages are equal too. Solidworks has a shallow curve at the beginning, but requires VBA programming sooner than other systems, due to the limited customization tools in its modeling GUI. Grasshopper has a small step at the beginning because of its visual programming notation, but has a smooth curve allowing a large amount of customization before using C# is required. Generative Components has a step when the user is introduced to the transaction code and another step for its scripting language, which is easier than general purpose languages. The liveness of the interface and the localization features in PIM creates a curve with many small steps from GUI modeling to scripting by presenting the script equivalent of users' actions to them, as well as concrete and immediate result of their scripting actions. To some extent a user may move these steps around, choosing when to encounter different aspects of scripting and data flow. If the user chooses to ignore liveness and use PIM as a traditional parametric system, then the curve will be similar to GC.

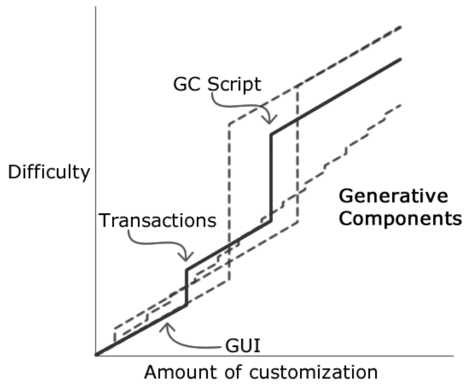
From the six end-user programming barriers that Ko et al. (2004) identify (see section 2.3.1), some are more often encountered by designers in CAD scripting than others. Selection, use and coordination barriers occur when the user does not know what tools to use, how to use them, and how to make them work together. For example, a designer who knows how to create and edit objects using the GUI, may not know how to do the same thing in the scripting window, considering that most CAD scripting windows block access to the model window and the GUI. So they may have a hard time finding the right command or syntax for tasks as simple as creating and editing objects.

Understanding and information barriers occur when there is not enough support for error prevention and error recovery. The distance between users' actions in the script window and the effect taking place and represented in the model window is a contributing factor for understanding barrier, because the user does not see exactly what step in the programming task caused the unexpected result in the model.

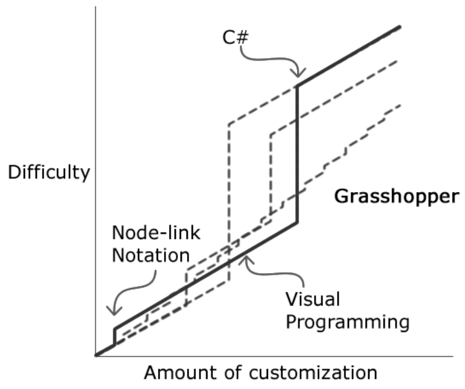
My strategy: I propose to lower selection, use, and coordination barriers by allowing the user to continue using the same GUI tools as before during scripting and keeping the new tools consistent with the modeling tools. By providing immediate feedback through live scripting, we make it possible for the user to identify potential errors and recover from them, before moving on to the next step, therefore lowering understanding and information barriers.



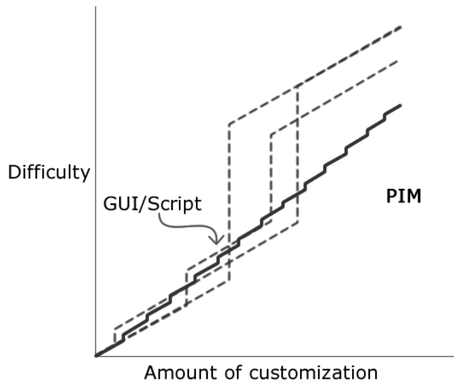
(a) Solidworks has a shallow curve in the GUI, but requires VBA programming for simple customizations.



(b) Generative Component has a step when transaction code is introduced and one when GCScript is needed.



(c) Grasshopper has a smooth curve after the initial step of learning the visual programming notation. There is very little GUI modeling in Grasshopper and C# is required for complex customizations.



(d) PIM intertwines UI modeling and scripting with its liveness and localization features, greatly reducing the steps of learning the notation. These steps are reorderable.

Figure 3.4: Every parametric system presents many barriers, some larger than others. The above diagrams capture the chief barriers for several current systems, as well as PIM. In the diagrams, smaller barriers disappear into the overall learning and use curve.

3.2.3 Fear of code in CAD

Scripting in CAD is just a tool for designers, not the end goal. Therefore they weigh the perceived cost and risk of learning and using programming against the benefits that it brings to their work. In the attention investment model (Blackwell, 2002) (see section 2.3.2), 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 (Ko et al., 2004) and have what has been described as 'fear of code' (Senske, 2005). It is commonly known that visual programming environments such as Grasshopper and Dynamo are popular among designers mostly because they look easy to learn and to use in the first glance.

Note 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 in hand. Attention investment and barriers apply to both learning and use.

My strategy: The ideal CAD system should have a low perceived cost/benefit ratio of scripting, 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. The key word here is *'perceived'*: the environment should both *"be"* and *"be perceived"* easy to use.

3.2.4 HCI guidelines

As discussed in Chapter 2, the literature provides a rich collection of guidelines and frameworks for designing and evaluating programming and end-user programming languages and environments. Because we do not intend to create a new scripting language, we have to settle for the language of the CAD system and focus our work on the environment, regardless of the language. Additionaly, we must take into consideration the fact that we are creating this environment for designers. The goal here is to enable design and empower the designer. With these points in mind, here are some of the guidelines that apply to our work.

- Creating program by direct manipulation, choosing from menus, graphical objects.
- Explain things in context.

- Start programming from something concrete, then generalize it.
- Increase visibility of the environment, show the data, show the flow of the program.
- Avoid hidden dependencies. Make them visible.
- Allow recognition rather than recall to reduce working memory. Show the user what is available for them to use.
- Maintain internal consistency of the system, so that similar things mean the same throughout the interface.
- Maintain external consistency with the user's external knowledge and the language of the domain.
- Prevent syntax errors by making them detectable, so that they do not cause semantic errors.
- Maintain low viscosity and premature commitment, so that end-users can start from something and change it later without too much effort.
- Give immediate feedback to the designers, so that they can react to it.
- Encourage incremental running and testing the program, so that they always have a code that works and they can trust.

3.3 Methodology

The goal of my research was to reduce the perceived cost/benefit ratio and barriers to programming in CAD for designers. My main strategy, which I call Programming In the Model (PIM), was 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, I did not know, nor did the literature provide, features that might meet these goals. Thus I undertook an iterative design and prototyping process in which I had regular reviews with experts in parametric CAD. Figure 3.5 shows the timeline of the project.

At each ideation phase, I designed features that tackled one or more EUP issues. I then demonstrated the feature in wireframe mockups, mostly produced in Adobe Fireworks. These prototypes enabled me to develop scenarios of use of the features and demonstrate

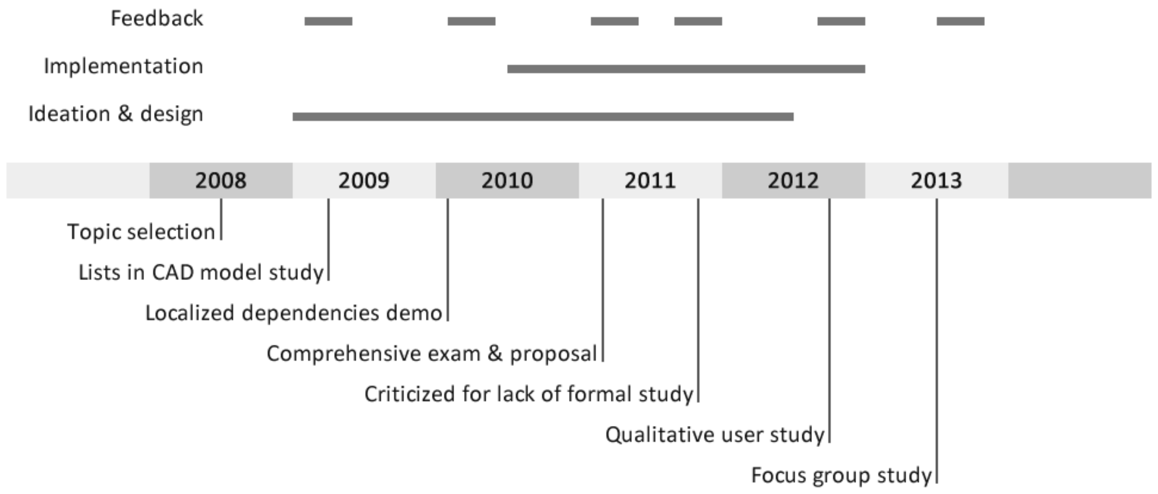


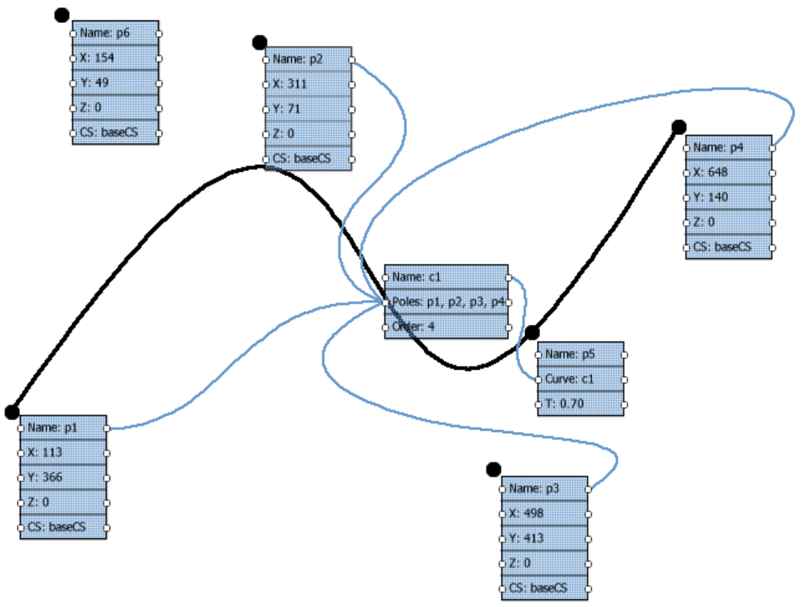
Figure 3.5: Project timeline

them first to my supervisors, and then to whomever I was seeking feedback from. Figure 3.6 show some of these mockups.

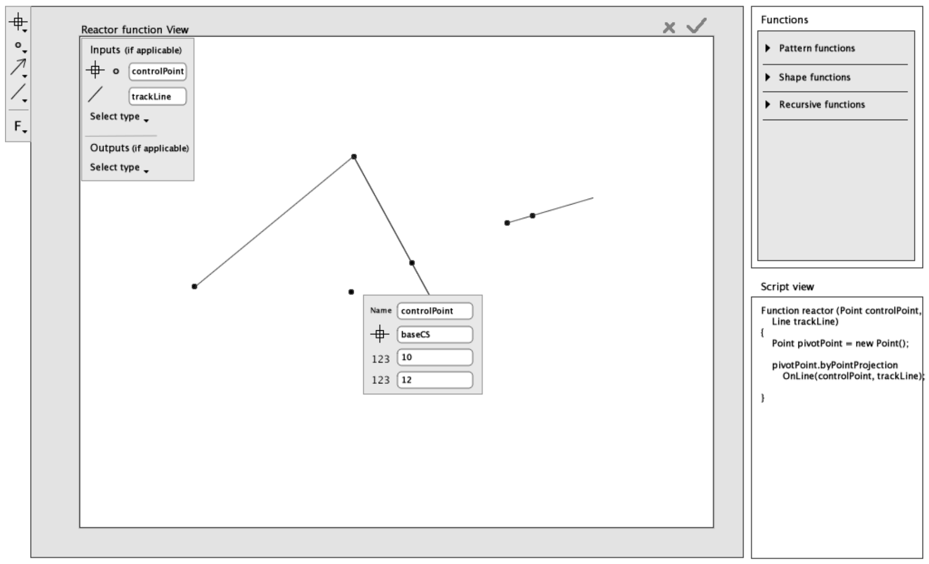
At the early stages of the project, I tried to implement some of these features within existing parametric CAD systems, particularly Generative Component. My attempts at hacking the interface were only partially successful, due to lack of access to the source code and limitations of the CAD interface. I ran a study with this prototype to evaluate the *lists in the model* feature. I present a summary of the study later in this chapter.

After this failed attempt at using commercial CAD systems, I decided to create a standalone prototype, especially because a working prototype is the only way to demonstrate and evaluate a live scripting interface. I chose the Processing programming language, mostly due to its low startup cost. I had to create a full parametric data structure with a small number of geometric objects, in addition to a CAD interface with multiple representations of the model, before I was able to implement Programming In the Model features. I was able to implement the underlying CAD system and some of the PIM features, before I reached the limits of my programming skills. At this point, I supervised two undergraduate students to help me implement the more advanced features[1].

[1]Wherever I refer to the “team” in this document, I mean the programming team, consisting of myself and the two undergraduate students. The team received system design advice from my senior supervisor.



(a) A mockup of localized information and dependencies in the model



```
Function reactor (Point controlPoint,
    Line trackLine)
{
    Point pivotPoint = new Point();

    pivotPoint.byPointProjection
        OnLine(controlPoint, trackLine);

}
```

(b) A mockup of functions in PIM

Figure 3.6: Early wireframe mockups

The process consisted of designing and demonstrating features using wireframe mockups, internal discussions and evaluations, implementing the features in the Processing prototype, and demonstrating the features to and asking for feedback from the CAD community[2]. The iterative process of design, implementation, and feedback happened in different ways over the course of the project, depending on how I was able to access the community. Mainly, I reached out to the community at the Smartgeometry workshops and conferences, through reviews on my paper submissions, at presentations within the school, and in three separate studies that I conducted. The decision to conduct the two formal studies was made when I presented the project to two different conferences in 2011 and received criticism for lack of formal studies. The complete description of these formal studies are presented in Chapters 5 and 6.

3.3.1 Study of localized lists in CAD models

This was a joint course project I did for IAT812[3] and IAT814[4] in SIAT, in which I explored different visualization and interaction techniques for lists in parametric modeling systems.. On the interaction side, I made a prototype in Generative Components that, however limited, allowed the user to create lists of objects directly in the model and leave the syntax of the list to the system. This was the first Programming In the Model feature, where users were able to interact directly with the geometric model to produce code. Figure 3.7 shows a demonstration of a list object moving around in the model and collecting points, with the syntax displayed directly in the model.

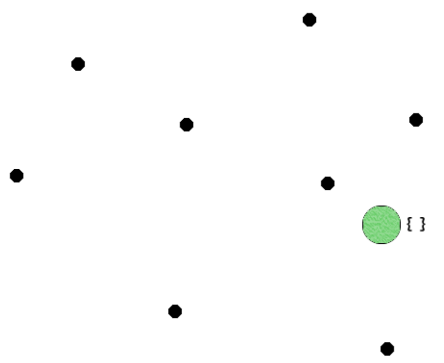
On the visualization side, I investigated the problem of representing these lists to the user, considering issues of scale, nested lists, overlapping lists, order and hierarchy. I developed a prototype in Processing that allowed me to explore different representation options and compare them with each other. Figure 3.8 shows some of those visualizations.

I ran a study to evaluate the effect of this feature on a modeling task. I recruited CAD users in Smartgeometry and students from SIAT and compared creating lists in the model, creating lists in the script, and a combination of both. The result of the study was inconclusive, meaning that there was no statistically significant difference between the three

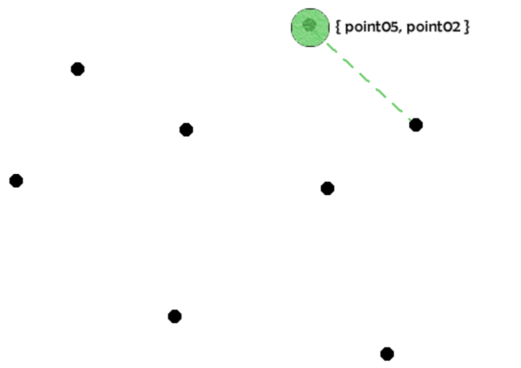
[2]Beside the guidance I received from my senior supervisor and my advisors throughout the project, the only external help I had was in the implementation of the prototype, as described above.

[3]IAT 812 - Cognition, learning, and collaboration

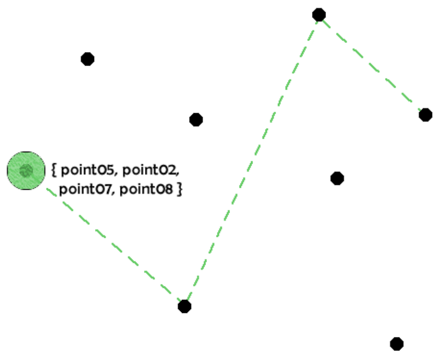
[4]IAT 814 - Knowledge visualization and communication



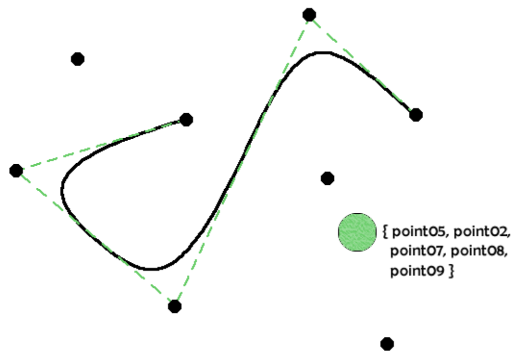
(a) The List object is an empty container.



(b) The user moves the List object over the points to add them to the list.



(c) The List object automatically adds necessary syntax to the list, such as commas and brackets.



(d) The user creates a curve by using the List object as input.

Figure 3.7: The list maker prototype in GenerativeComponents

conditions. The reason was that half of the participants had significant programming experience and no design background. I learned from this experience to only recruit participants from the user group that I designed the system for. Another reason was the limitations that the commercial CAD system put on my prototype, considering that I did not have access to the source code. This failure lead me to develop a stand-alone prototype that was independent from any existing CAD system, so that I could manipulate the interface freely and implement the PIM features. Please refer to Appendices F and G to read the full description of the project.

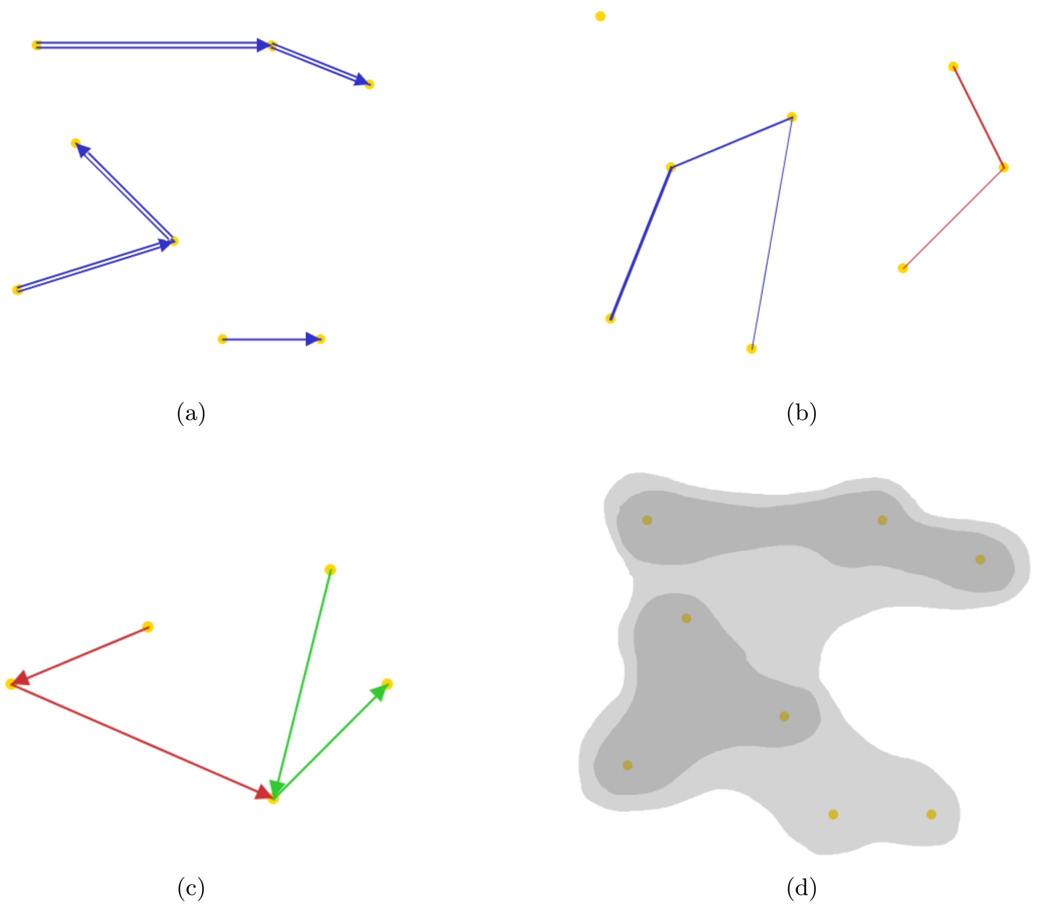


Figure 3.8: Visualization of lists in the CAD model

3.4 Summary of chapter

In this chapter I briefly described PIM features, in order to familiarize the reader with the end result of the project and some of the terms and phrases I use often in this document. Then I presented my arguments, connecting the background research with the goals of Programming In the Model. The third part of the chapter was focused on the process of design, implementation and evaluation of PIM, including an early study of localized lists in the CAD model that was the starting point to the rest of the features of PIM. In the next chapter, I present PIM features and the PIM prototype in detail.

Chapter 4

Programming In the Model

The characteristics of a gentle slope parametric CAD system, as I envision them, fall into three groups, as briefly discussed before. *Localization* refers to features related to local representation of data in the model view where the objects reside. *Liveness* describes characteristics of a live system where all representations update concurrently. Finally, *Lookahead* refers to features that provide the end-user a look into the future state of the model before committing to a change. In this chapter, I present these features in detail, using the prototype whenever possible to demonstrate how they can be applied in a CAD interface. I also describe how a task, such as creating a function, can be done in a PIM-like system[1].

[1]To access the prototype and watch the video demos, please visit http://www.sfu.ca/ mmaleki/Maryam-MalekiWebsite/endUser.html or scan the QR code below.

4.1 Introduction to the PIM prototype

The PIM prototype has a limited number of geometric objects and an interface that comprises a model window, a dependency graph, and a script window (Figure 4.1). The goal of creating the prototype was not to make a new, independent CAD system, but to have a platform to implement and demonstrate PIM features in a realistic setting. The current implementation of PIM prototype, showing only a subset of our ideas, together with a video demo, presents a complete picture of PIM features.

I decided to limit the prototype to a two-dimensional space, because the added complexity of 3D rendering and interaction was beyond the scope of my thesis. The PIM prototype is a miniature parametric CAD system, with four geometric types: coordinate system, point, vector, and line. Being parametric means that the user explicitly defines the relationship between objects, and builds up and edits the design using these relationships[2]. The system propagates any change down the dependency tree and updates the whole model.

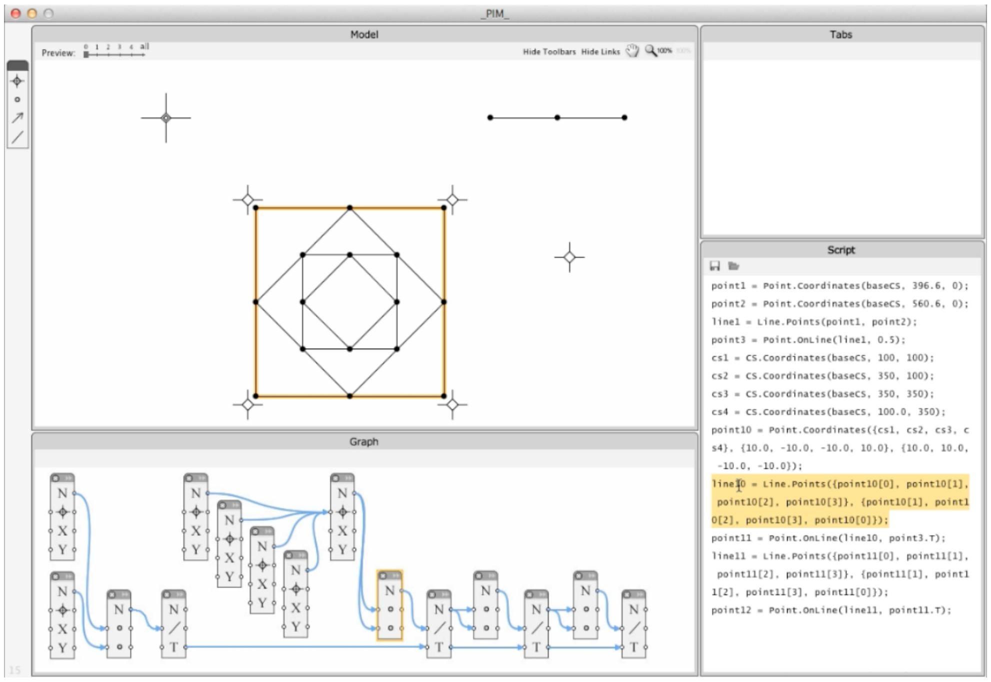
Each type of objects can be created with different methods or *update algorithms*, (equivalent to class constructors in object-oriented programming). Depending on the active update algorithm, the object needs a different set of inputs or arguments. For example, a line may be created by connecting two points, in which case it needs two points as input, or by starting from a point in a certain direction to a certain length, in which case it needs as input a point, a direction vector, and a float value for length. Table 4.1 shows a list of object types and their update algorithms in the prototype[3]. Underneath each update algorithm in the table, there is a list of inputs required for creating the object. For example, a vector By Origin X Y needs a name, a point for origin, and two float numbers for X and Y. I designed an icon for each object type and object update algorithm, as shown in the table, and used them in the prototype, as described later in this chapter.

Implementation

For the parametric data structure, I used an observer/observable pattern. Each object observes its upstream objects and adds itself to their observer list. When an upstream object changes, it fires an update message to its observers, prompting them to update

[2]See Chapter 2.1 to read more about parametric modeling.

[3]Commercial CAD systems are in 3D, and have many more types of objects each, with several update algorithms.



```
point1 = Point.Coordinates(baseCS, 396.6, 0);
point2 = Point.Coordinates(baseCS, 560.6, 0);
line1 = Line.Points(point1, point2);
point3 = Point.OnLine(line1, 0.5);
cs1 = CS.Coordinates(baseCS, 100, 100);
cs2 = CS.Coordinates(baseCS, 350, 100);
cs3 = CS.Coordinates(baseCS, 350, 350);
cs4 = CS.Coordinates(baseCS, 100.0, 350);
point10 = Point.Coordinates({cs1, cs2, cs3, c
s4}, {10.0, -10.0, -10.0, 10.0}, {10.0, 10.0,
 -10.0, -10.0});
line10 = Line.Points({point10[0], point10[1],
 point10[2], point10[3]}, {point10[1], point1
0[2], point10[3], point10[0]});
point11 = Point.OnLine(line10, point3.T);
line11 = Line.Points({point11[0], point11[1],
 point11[2], point11[3]}, {point11[1], point1
1[2], point11[3], point11[0]});
point12 = Point.OnLine(line11, point11.T);
```

Figure 4.1: The interface of the PIM prototype, showing a 2D modeling window, a node-link graph window, a script window, and a fourth window for non-geometric entities.

themselves according to its new state. This programming method made it possible to have a parametric modeler that propagates any change immediately throughout the geometric model.

I also used the Model-View-Controller pattern[4] to separate the data structure from what is being displayed on the screen. For each object, there is a model[5] component that holds the data associated with that object. Then there is the view component that uses that data and renders the object on the screen. The controller component connects the model[6] with the view and also handles interactions with the objects. So depending on the object type and update algorithm, the controller registers clicks on the object, interprets them and takes appropriate actions for that particular interaction.

[4]I use the word *model* to refer to the two or three dimensional model of the design in CAD. The only place I use it differently is when I talk about the MVC pattern. In those cases, a footnote is provided to distinguish the different use of the word.

[5]Model in MVC

[6]Model in MVC

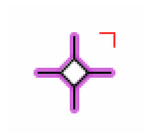
Coordinate System (CS)		
	By_Cartesian_Coordinates	
		Name: String CS: CS X: float Y: float
Point		
	By_Cartesian_Coordinates	
		Name: String CS: CS X: float Y: float
	By_Parameter_on_Line	
		Name: String Line: Line T: float
Vector		
	By_Origin_X_Y	
		Name: String Origin: Point X: float Y: float
	By_Sum_of_Vectors	
		Name: String Vector: Vector Vector: Vector
Line		
	By_Points	
		Name: String StartPoint: Point EndPoint: Point
	By_Origin_Direction_Length	
		Name: String Origin: Point Vector: Vector Length: float

Table 4.1: Object types and their update algorithms in the PIM prototype

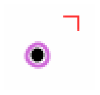
4.2 Localization

4.2.1 Localized information display

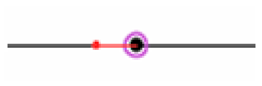
The information that is locally available in the model window comes in several different forms. The first and simplest one is what we call input handles. These handles allow the user to manipulate numeric inputs of each object by clicking and dragging the object itself. This interaction is very direct and gives immediate feedback to the user. Of course handles are not new. Almost every CAD system provides such handles on at least their primitive objects. Handles in PIM prototype are universal for numeric values, including X and Y coordinates of points and coordinate systems, length of lines created by start point, direction and length, and the parameter or T value of points on a line (Figure 4.2).



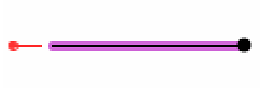
(a) Coordinate system handle changes X and Y coordinates



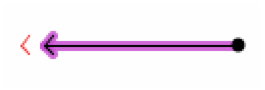
(b) Point by cartesian coordinates handle changes X and Y coordinates



(c) Point on a line handle changes the T parameter of the point on the line



(d) Line by origin, direction, length handle changes the length of the line



(e) Vector by origin, X, and Y handle changes the X and Y transformation of the vector

Figure 4.2: Handles allow direct interaction with numeric inputs in the model window.

To manipulate non-numeric inputs, for example a point or a coordinate system, the user opens a small toolbar that contains buttons for each input. The input buttons when pressed, accept new objects as inputs. For example, to edit a point on a line and change the line it is located on, the user opens the point's toolbar and presses the line input button (as shown in figure 4.3). With the button in the pressed mode, the user clicks on another line in the model. PIM uses this line as input for the point and updates the point immediately and puts it on the line selected by the user. The toolbar always follows the object it represents, unless locked down by the user to a specific location. It does not go outside the visible area of the model window when the object does, but positions itself at the edge of the window closest to the object's location.

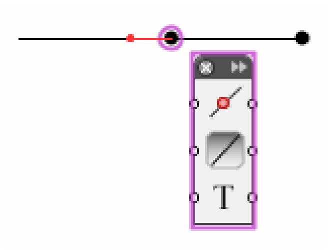


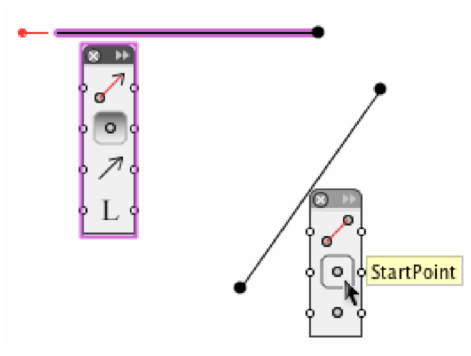
Figure 4.3: The first icon in the toolbar shows the type and update algorithm of a point on a line.The next two icons show the two inputs: a line and a T parameter. The line input button is in the pressed state, ready to accept new input.

Input buttons can also be used to access properties of objects similar to dot-notation. For example (Figure 4.4), to make line02 start from the same point as line01, the user opens line02's toolbar, then presses the startpoint input button. Now line02 is ready for a new startpoint. Then (s)he opens line01's toolbar and presses its startpoint input button. This action makes a relationship between these two lines similar to this expression.

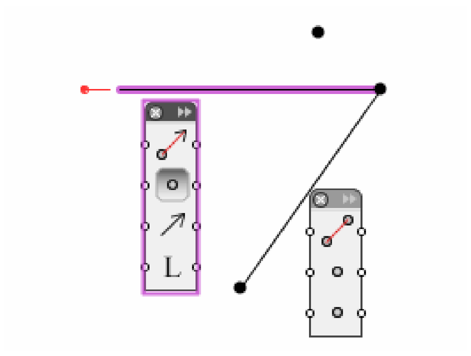
line02 . StartPoint = line01 . StartPoint

The toolbar expands to present more information about the object. For each input, there is a text field or expression field in addition to the input toolbar. The expression field accepts expressions such as object names, dot-notation syntax, and mathematical calculations as shown in Figure 4.5. These names and expressions can be typed in by the user. Alternatively they can be be inserted into the expression fields by direct manipulation. In any text field, pointing to another object in the model (with shift+click) inserts its name in the text. This feature is not new to my prototype, as other systems have similar interactions in place. What is new is that the same interaction works in the script window and allows the user to pick objects from the model during scripting (more details in the following sections).

The icons I designed for object types, object update algorithms, and input types let the user know what type of input is required for each input field. For example, an input icon that looks like a line means that the input type must be a line. Input buttons are strongly typed in PIM. To prevent errors, the input buttons do not respond to mismatched type inputs. In other words, they do not accept the input and wait for a correct selection. If the



(a) To make line01.startPoint to be equal to line02.startPoint, the user opens line01 toolbar, presses the startPoint input button, then opens line02 toolbar and clicks on its startPoint input button.



(b) The result is similar to this expression: line01 . startPoint = line02 . startPoint

Figure 4.4: Accessing dot-notation properties of objects, using direct manipulation in the model.

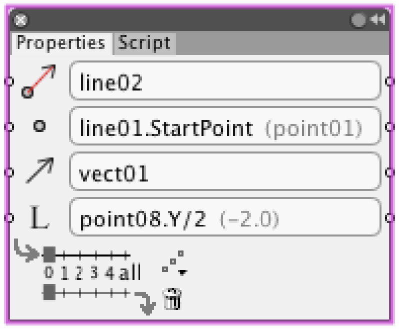
user types in an input with an incorrect type, or a name that does not exist in the model, the text turns red, indicating that there is an error that needs to be fixed.

4.2.2 Localized script representation

Another new feature in the prototype is the script tab in the expanded toolbar that carries a copy of the script that creates the object (Figure 4.6). Depending on the scripting language, one or more lines of code are responsible for the current state of the object. These texts all appear in the script tab and can be edited in place. So if the user wants to make changes to the object that are more than can be handled in the expression fields, (s)he opens the script tab and makes the change in the code, which is immediately reflected in the model as well as the script and graph windows.

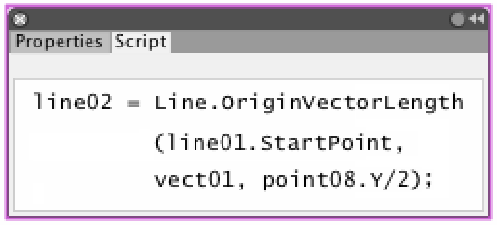
4.2.3 Localized dependency representation

An important piece of information about an object in a parametric modeling system is its relationship to other objects in the model. This information determines what objects and/or relationships need to be edited for this object to change and what will happen to other



```
line02
line01.StartPoint (point01)
vect01
point08.Y/2 (-2.0)
```

Figure 4.5: Input fields accept different types of expressions such as object names, property names (dot-notation), and mathematical operations.

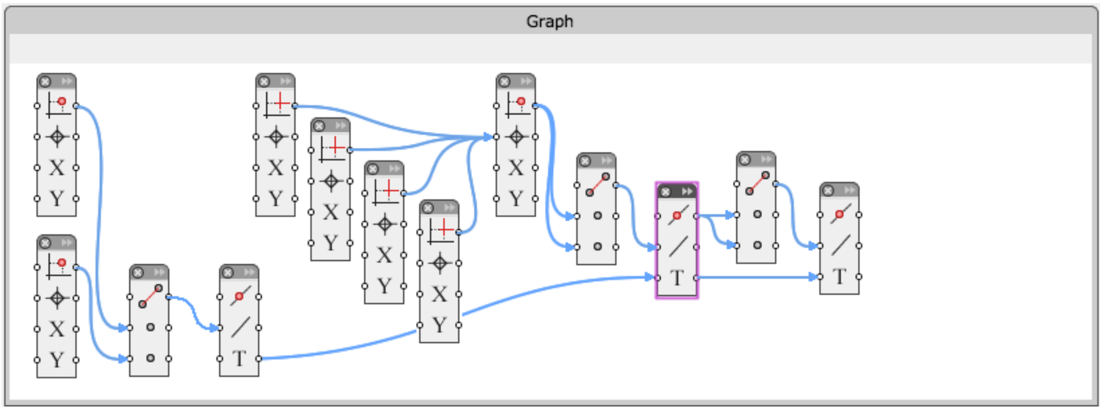


```
line02 = Line.OriginVectorLength
         (line01.StartPoint,
         vect01, point08.Y/2);
```

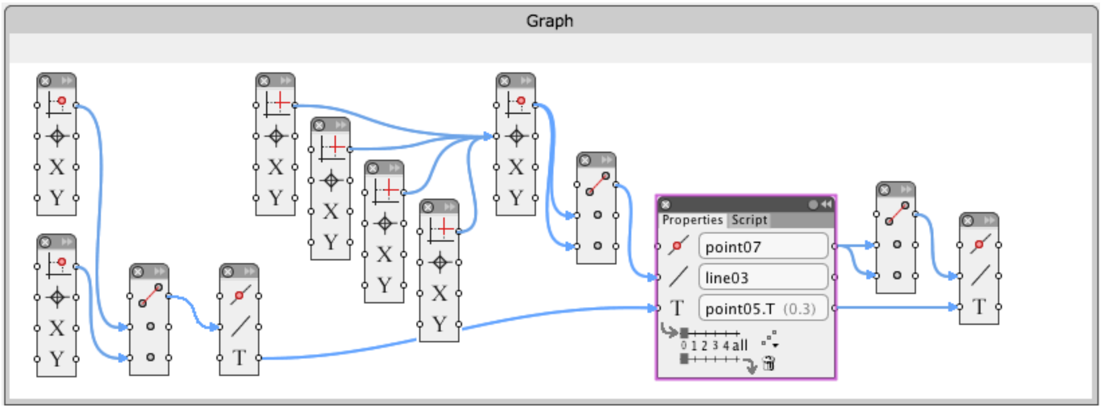
Figure 4.6: The script tab in each toolbar presents a copy of the script and can be edited on the spot in the model window.

objects in the model if it does. In most parametric systems, the dependency information is displayed in a graph diagram in a separate window. The PIM prototype is designed to have a graph window, in which nodes represent objects and directed links represent the flow of data from one object to another. This window is not fully implemented in the prototype as of this time and only contains a mockup dependency graph. In order to maintain consistency, the nodes in the graph are the same as edit toolbars in the model, and when expanded (in a working version), present all the same tools, including input buttons and input fields, as well as the script tab (Figure 4.7).

As described earlier, following this data structure to find how objects are connected requires going back and forth between the model window and the graph window. Sometimes that is inevitable, when the dependency structure is complex or when information about the whole graph is needed. But other times the shift of focus can be avoided by *locally* and *selectively* presenting this data to the user in the model where his/her focus currently lies.



(a)



(b)

Figure 4.7: The graph window is intended to show the dependency structure of the model in the form of a node-link diagram. The nodes are consistent with the edit toolbars in the model with the same functionalities. One of the nodes is expanded in diagram (b) to access more information, including the script tab.

"Locally" means that this data must be represented where the objects or their toolbars are located in the model (explained later). "Selectively" means that the user specifically asks to see this information about one or more objects. It also means that (s)he specifies how deep in the dependency structure (s)he wants to go. For example (see Figure 4.8), for a point that is on a line that connects two points that are in a coordinate system, the user may want to see only one level of upstream dependency (the line), or two levels (the line and two points), or three levels (the line and two points and a coordinate system). Two sliders in the edit toolbar enable the user to select how many dependency levels upstream and/or downstream is to be displayed for this object.

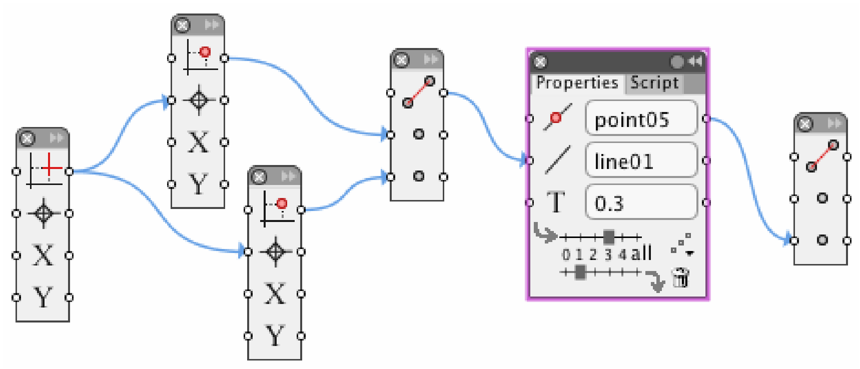


Figure 4.8: For point05, the user has moved the sliders to display three levels of upstream and one level of downstream dependencies.

To maintain consistency throughout the interface, I chose the same representation as the graph window for displaying dependencies in the model. Directed links connect objects, or their toolbars if they are open. For each input, there is an incoming port on the left, connecting to incoming links that bring data to the object, and an outgoing port on the right, connecting to outgoing links that carry data to other objects. Figure 4.9 shows different types of ports and what they all mean.

These links are interactive. For each object, the incoming links can be detached from a current upstream object and attached to another one. The result is similar to editing an input of the object. Links only snap to ports with similar data type to avoid type mismatch errors.

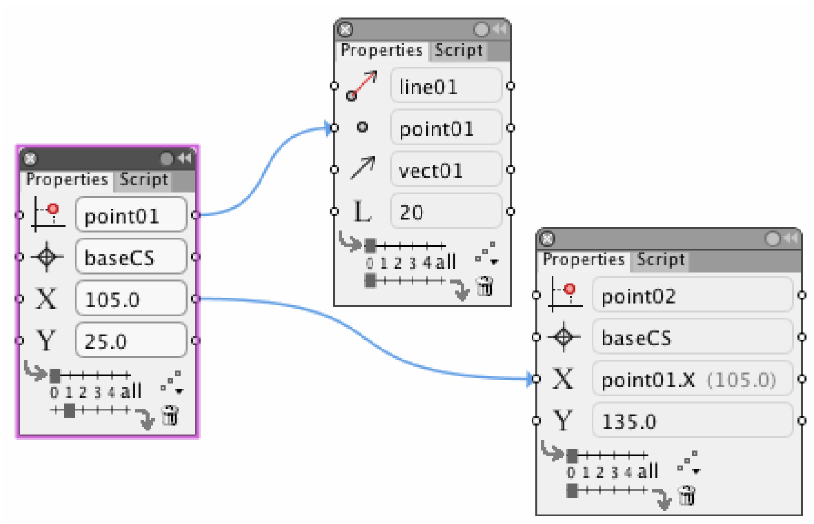


Figure 4.9: When the toolbar is open, the links show details of the relationship. In this example, line01.Origin = point01, so the link exits the name field of point01. But, point02.X = point01.X, so the link exits the X field of point01, showing that point02 gets data from the X input.

Implementation

Dependencies are stored in nested hashtables (series of key/value pairs). Each object not only stores what objects are upstream and downstream, but whether or not the relationship is with the object itself or one of its properties (as shown in Figure 4.9). In some cases, there are multiple dependencies from one object. The nesting in the hashtable is used to store this type of information. So for point01 the hashtable will look like this:

upstream_HASH [CS01: properties_HASH [Obj:__, X:__]]

To implement a realtime rendering of dependency links, we defined a dependency manager for each object. The manager passes the data to the dependency display component that keeps track of where the start and end points of the link are and draws a bezier curve between them. These points or their locations change dynamically as toolbars are opened, closed, or moved and objects are manipulated by the user. The dependency manager is also responsible to determine if a link is to be visible or not, based on the information from the dependency sliders. If a link has a visible status (meaning that somewhere in the model a slider has turned it on), it is rendered by the dependency display.

4.3 Liveness

By far, the most important feature of Programming In the Model is the multi-directional liveness of the interface (Please see Chapter 2.4.4 for more information about liveness). In computer-aided design, the Graphical User Interface is usually quite live. It means that users' actions result in almost immediate effect in the 2D-3D model. But this liveness does not extend to the scripting window. In most of these systems (almost all, except one or two recent ones), opening the scripting window is similar to opening a completely separate application. The user loses access to the rest of the CAD interface and is limited to the tools and notations of the scripting environment. In addition, the script window is not live and does not give any feedback to the user until it is run or compiled, and only if there are no errors, the model updates and reveals the new design.

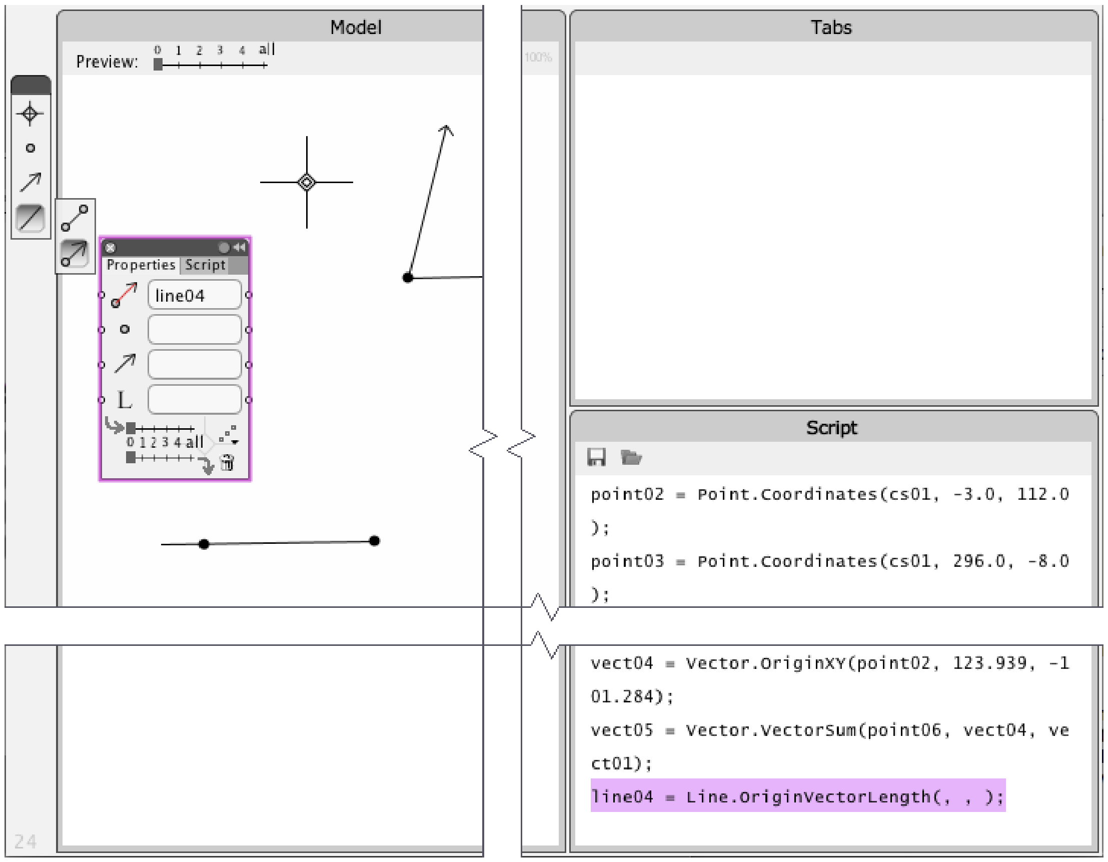
Liveness, when employed in an interface, is usually uni-directional, giving immediate feedback to coding actions only. The liveness in Programming In the Model is multi-directional, meaning that all representations in the interface, including the model, the graph, and the script, are live, work concurrently, and update immediately when there is a change.

This is live coding, in that you code and you see the result immediately. But it is also more: you model and you get the code immediately too. This is what we mean by Programming In the Model.

4.3.1 Model to script liveness

Let me explain this feature with an example. Starting in the model window, I initiate a line by selecting the line button from the create toolbar. Then I select By-Origin-Vector-Length as its update algorithm. PIM opens the edit window as shown in Figure 4.10. The last line of the script shows the unfinished syntax of a line by origin vector length. The code only shows the steps I have already taken: selecting object type and initializing it and selecting its update algorithm. The argument part of the code is empty awaiting inputs. As I continue with selecting inputs in the model, the code gradually gets completed.

A few things happen here that I need to point out. First and foremost, non-programmers get to see how a line is created in the script. Without even touching the script window, they get a glimpse into the world of programming. They notice that there is a line of code that is creating this object, and each action they take during modeling seems to produce corresponding text in this line of code.



```
point02 = Point.Coordinates(cs01, -3.0, 112.0
);
point03 = Point.Coordinates(cs01, 296.0, -8.0
);
vect04 = Vector.OriginXY(point02, 123.939, -1
01.284);
vect05 = Vector.VectorSum(point06, vect04, ve
ct01);
line04 = Line.OriginVectorLength(, , );
```

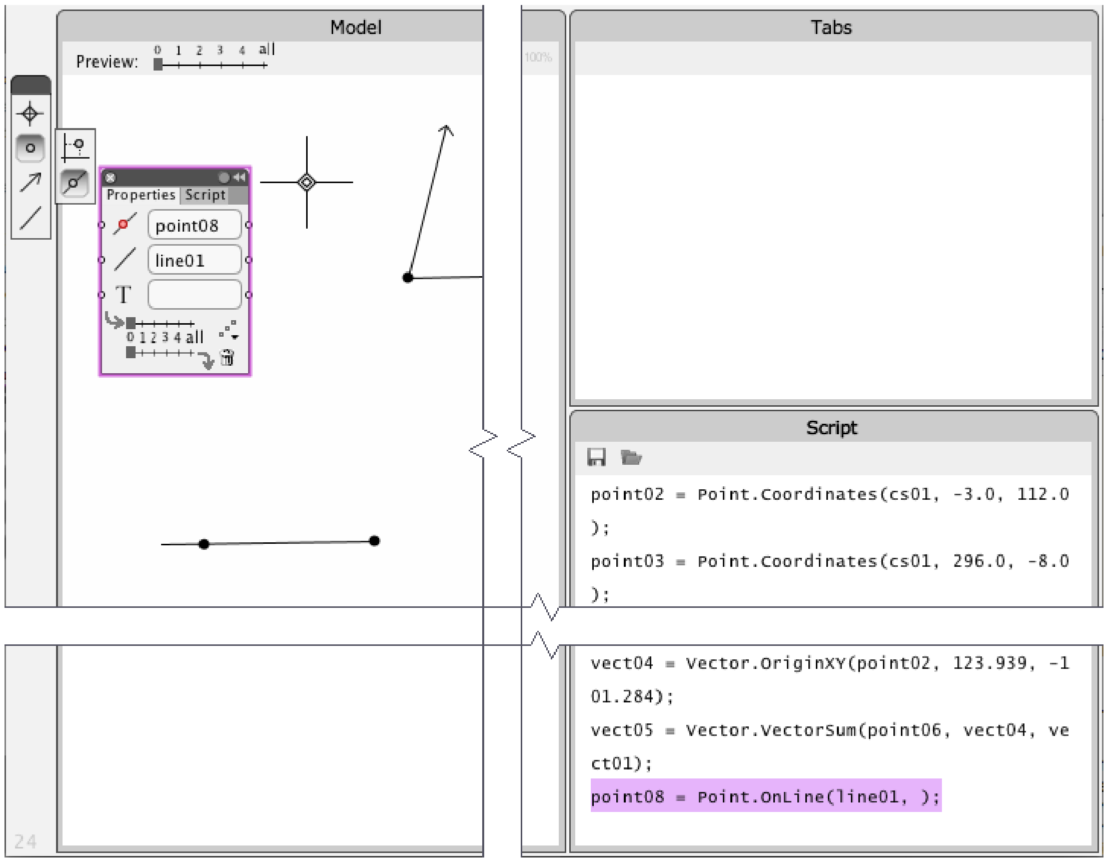
Figure 4.10: Creating a line in the model with the script being generated by the system step by step

Second, the intermediate end-user programmer, who may not know how something is done in the code, can work in the GUI and learn the syntax that is being generated. The liveness allows the user to switch from window to window even in the middle of a task. So the user, when facing a barrier in scripting, can switch to the model window, use GUI tools, then switch back to scripting. This feature extends to all windows in the interface, so the user can choose to work in the model, the graph, and the script at any time during the task.

In addition, liveness gives the users, from novices to experts, the choice to work in the model for simple tasks and let the system generate the code, then move over to scripting for more complex programming actions.

4.3.2 Script to model liveness

Continuing with the example, this time I start in the script window and initiate a point-by-parameter-on-line. Looking at the GUI in the model window, I see that the create toolbar has followed my actions and has opened a point-on-line window (Figure 4.11). If I choose to continue working in the script, I notice that each argument that I provide in the script window, also appears in the input expression field in the model window, giving me a better idea of what each component of the syntax means in terms of modeling actions.



```
point02 = Point.Coordinates(cs01, -3.0, 112.0
);
point03 = Point.Coordinates(cs01, 296.0, -8.0
);
vect04 = Vector.OriginXY(point02, 123.939, -1
01.284);
vect05 = Vector.VectorSum(point06, vect04, ve
ct01);
point08 = Point.OnLine(line01, );
```

Figure 4.11: Creating a point in the script with the model window reflecting the actions

Unlike most CAD systems, the script window does not block access to the model. It means that during scripting, I can highlight other objects in the model, find their names or inputs, and even directly select them as arguments in the script. For example, if I want to select a certain line in the model as input for the point-on-line I am creating (a line that I am looking at in the model window but whose name I may or may not know), I can shift+click on the line in the model and its name appears in the script window where the cursor is. So I use recognition rather than recall to reference objects in the script.

As soon as I provide enough arguments for the object I am creating in the script, PIM displays the object in the model. The same thing happens when I edit an object, as soon as I provide the new argument. For a designer, this is valuable feedback, especially because they need to see what they are designing in order to make design decisions and explore the design space.

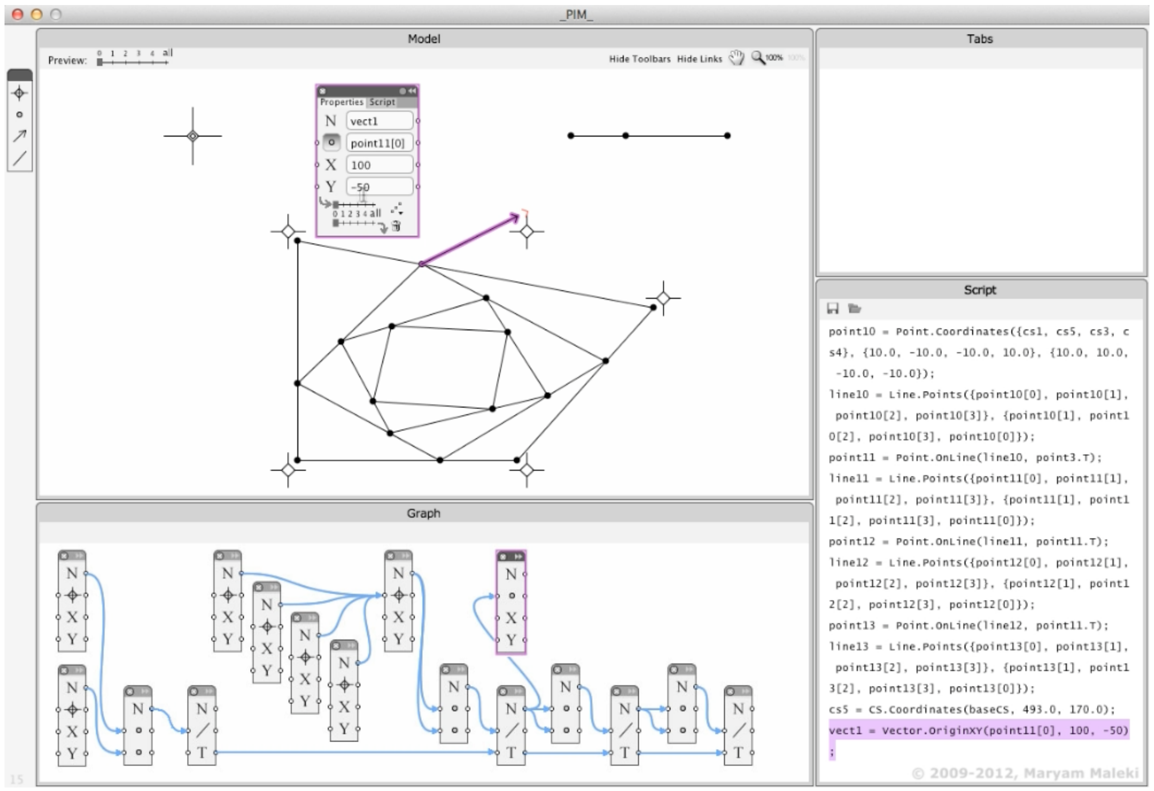
From a programming perspective, the immediate feedback gives the user a chance to check the work at each step and make sure the result is what was intended before moving on. In a more traditional scripting interface, the user works on the code for a while, then compiles it and checks the result in the model. At that point if there are any errors or unintended results, the user needs to go back and find out where in the code the adjustments need to be made. It is important for an end-user programmer to always have a working program, a very recent version that they can go back to if there is a problem. A syntax error in a live interface results in a halt in the immediate feedback, demanding action from the user to fix the error. Semantic errors produce an unwanted result in the model that is immediately noticeable for a user who takes advantage of the live interface and checks the immediate feedback regularly.

4.3.3 Brushing and highlighting

The liveness of the interface is supported by the universal brushing and highlighting feature. The object that is being created or edited at the moment is automatically highlighted throughout the interface, connecting its different representations for the user (Figure 4.12). So the user can quickly find it in any window they look without having to use search tools and getting distracted from the task.

Implementation

Implementing the live interface was an interesting challenge for the team. Any change made anywhere in the interface had to be propagated across all other representations immediately. The MVC system allowed for introduction of the script as another View component for each object. After being parsed, data from the script is handled the same way as data from the visual 2D model by the Controller component associated with the object.



```
point10 = Point.Coordinates({cs1, cs5, cs3, c
s4}, {10.0, -10.0, -10.0, 10.0}, {10.0, 10.0,
 -10.0, -10.0});
line10 = Line.Points({point10[0], point10[1],
 point10[2], point10[3]}, {point10[1], point1
0[2], point10[3], point10[0]});
point11 = Point.OnLine(line10, point3.T);
line11 = Line.Points({point11[0], point11[1],
 point11[2], point11[3]}, {point11[1], point1
1[2], point11[3], point11[0]});
point12 = Point.OnLine(line11, point11.T);
line12 = Line.Points({point12[0], point12[1],
 point12[2], point12[3]}, {point12[1], point1
2[2], point12[3], point12[0]});
point13 = Point.OnLine(line12, point11.T);
line13 = Line.Points({point13[0], point13[1],
 point13[2], point13[3]}, {point13[1], point1
3[2], point13[3], point13[0]});
cs5 = CS.Coordinates(baseCS, 493.0, 170.0);
vect1 = Vector.OriginXY(point11[0], 100, -50)
;
```

Figure 4.12: Highlighting the object of interest in all windows makes it easy for the user to keep track of its changes and also to switch between the windows during the task if necessary.

Developing the script language and its parser was a challenge as well, since it had to be efficient enough not to slow down system operations during editing, but dynamic enough to detect and implement changes as they were being made. An eventual solution to the problem involved using field detection, with commas and brackets serving as markers. This allows updating only those values that have been changed, as they are being changed, saving processing time compared to re-compiling and parsing the entire method or even the entire script.

This method of implementing the live script requires a robust error- and warning-handling system to determine if the line of script becomes invalid at any time due to deletion or invalid placement of the markers in order to prevent compiling objects incorrectly or crashing the system altogether. The script language itself is quite similar to that of other CAD systems; however the parser allows more fluid editing and feedback because of the efficient method of individual argument parsing described above, as well as having a direct connection between each object and its associated line(s) of script.

We use a simple syntax that is comparable to the Processing programming language and it not specific to the PIM prototype. The programming language style we use in PIM is a combination of declarative and procedural.

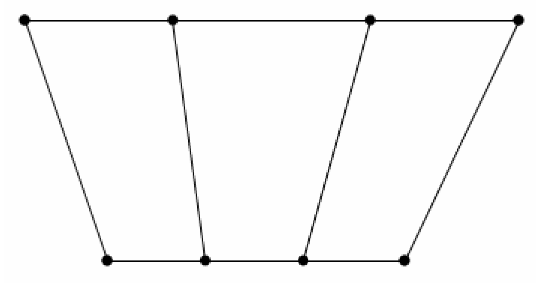
4.4 Lookahead

The Preview feature displays two versions of the model upon user's request, before change and after change (Figure 4.13). This allows the user to compare the new state with the previous state and make an informed decision. This is different from undo, as with undo the user only recovers from an unwanted action performed recently, but with preview the user gets the two states at the same time and can avoid the unwanted action. It is especially useful in parametric systems for two reasons. First, it may not be entirely clear what needs to change and how, for a certain effect in the model, whereas in a non-parametric system, each object is responsible for its own state. Second, when something is changed, there may be side effects in the model that the user has not foreseen. So it is helpful to be able to check the whole model for unwanted effects before confirming an action, with the changed parts highlighted by the system. Without preview, the user would have to confirm an action, then evaluate the result and if necessary, compare it with state of the model before the change. Due to limits of our short term memory, the information from the before model quickly gets lost in our mind and makes this task hard if not impossible. Also, change blindness makes us easily miss a side effect in the model, especially if it is not where our focus of attention is.

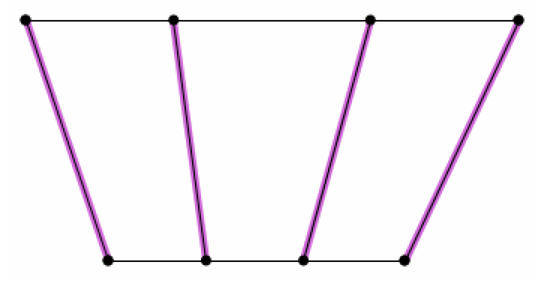
The preview slider gives the user the option to choose how many levels in the dependency structure he/she wants to preview. In the future, I would also like to have individual object preview, so that the user can focus on one or more objects in the model, regardless of their dependency level and without seeing a preview of other objects in the model.

Implementation

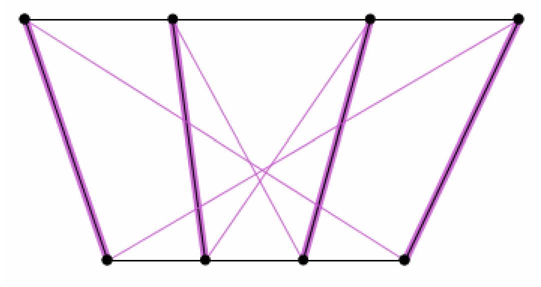
In PIM, any changes performed by the user are temporary until confirmed. The series of changes made before confirmation are stored in a buffer called the temporary model. When confirmed, these changes are pushed to the permanent model. Having two versions of each



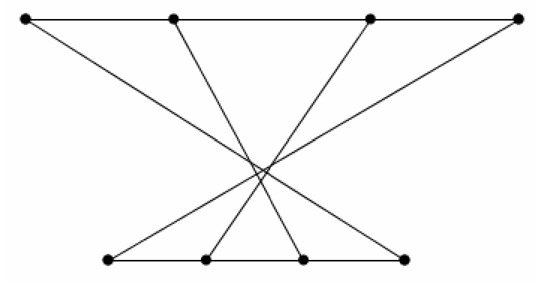
(a) A set of lines connects two point-sets



(b) The user edits the order, in which the line-set connects the points



(c) A preview of the model is displayed in purple.



(d) The user confirms the change, and the preview model becomes permanent.

Figure 4.13: The preview feature

object allows us to do two different things: Firstly it allows for undo functionality, error checking, and even auto-reverting if the system is about to crash as a result of a mistake made by the user. Secondly, it allows for a visual side-by-side rendering of objects for preview simply by adding another view component to them.

4.5 PIM features in use: Functions

In this section, I explain how a PIM-like system handles creating and reusing a code block. Some of the features I talk about in this section are not implemented in the PIM prototype. But they are the natural progression of a live interface such as PIM. I use functions as an example, because functions are a general programming construct that capture the notions of a block and references within and outside a block. However, any piece of code that is to be generalized, stored for future use, and may accept inputs and return outputs can be created in PIM a similar way. Examples of such modules are classes and compound nodes.

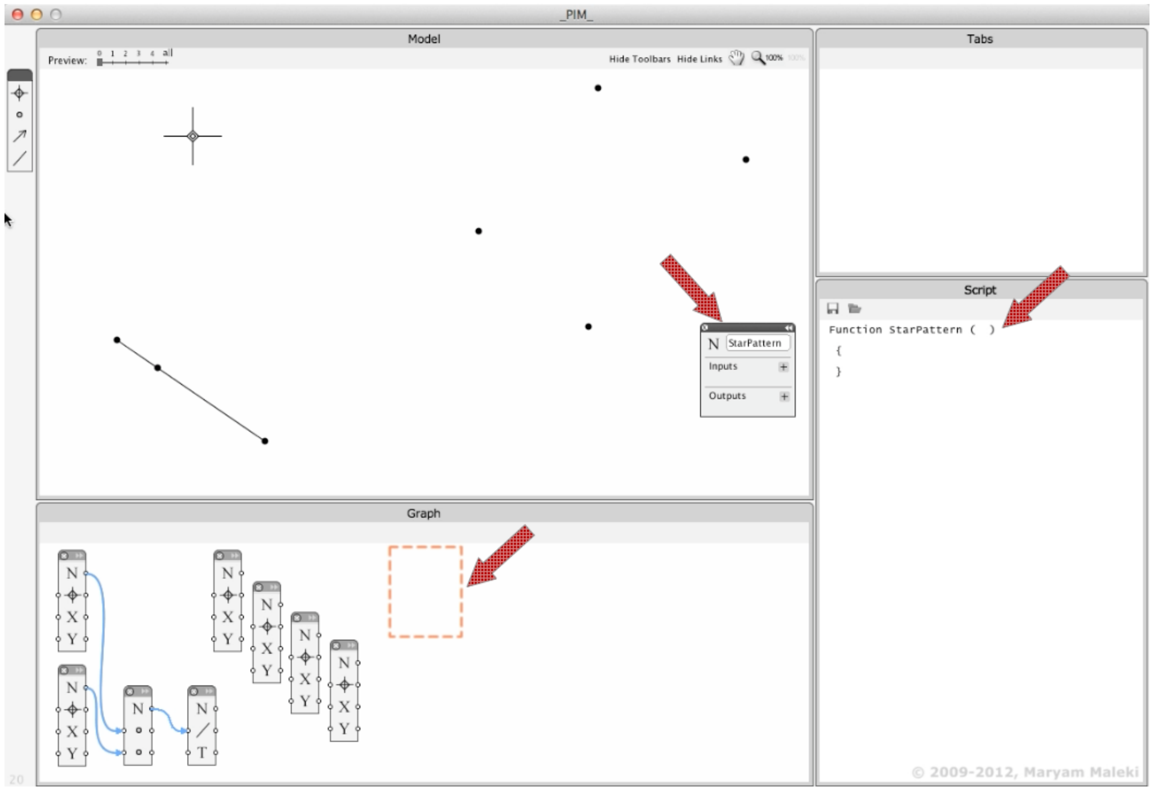
4.5.1 Creating functions

There are three options for creating a function (or a block of code) in parametric CAD systems: from scratch, from an existing part of the model/script, and by copying and modifying another function. A function is like a boundary, drawn around a piece of the model, the graph, and script. A PIM system would treat functions just like boundaries, not only around objects in the file, but around user's actions as a time boundary. What this means is that when the user is working on a function, either creating or modifying it, his/her actions are recorded within the boundary of the function. This is similar to how macros work, but different in that during this time, the system presents complete visibility and accessibility in the entire interface.

Figure 4.14 shows a function from scratch. The graph shows an empty boundary and the script shows an empty function block. The representation of a function in the modeling GUI is very similar to the edit toolbars to maintain consistency, so the user can take advantage of the familiar notation. In this state, the system is in recoding mode: any actions in the model is recorded in the function. Any node that is put inside the function boundary in the graph and any text inside the body of the function in the script are part of the function as well (as shown in Figure 4.15). However, if the user needs to add or modify an object outside the boundary of the function before it is finished, (s)he can temporarily disable the function view, work in the model, then re-enable the function again and continue. This allows the user to specify whether an action is to be recorded inside or outside the function, without having to save or cancel the unfinished function.

The liveness of the system allows the user to switch from one representation to another during the task of creating the function and provides immediate feedback in the other representations. One of the benefits of such live systems is that user works on a concrete version of the function (the 2D/3D model) instead of the abstract script. It also encourages incremental running and testing of the program by presenting the result in the model with each scripting step.

In drawing a boundary around a piece of the parametric structure, the user decides what is inside and what is outside the function. But (s)he also has to determine how the inside components connect to the outside component. This is a typical step in programming by example interfaces. One approach is to have the computer decide how to generalize the

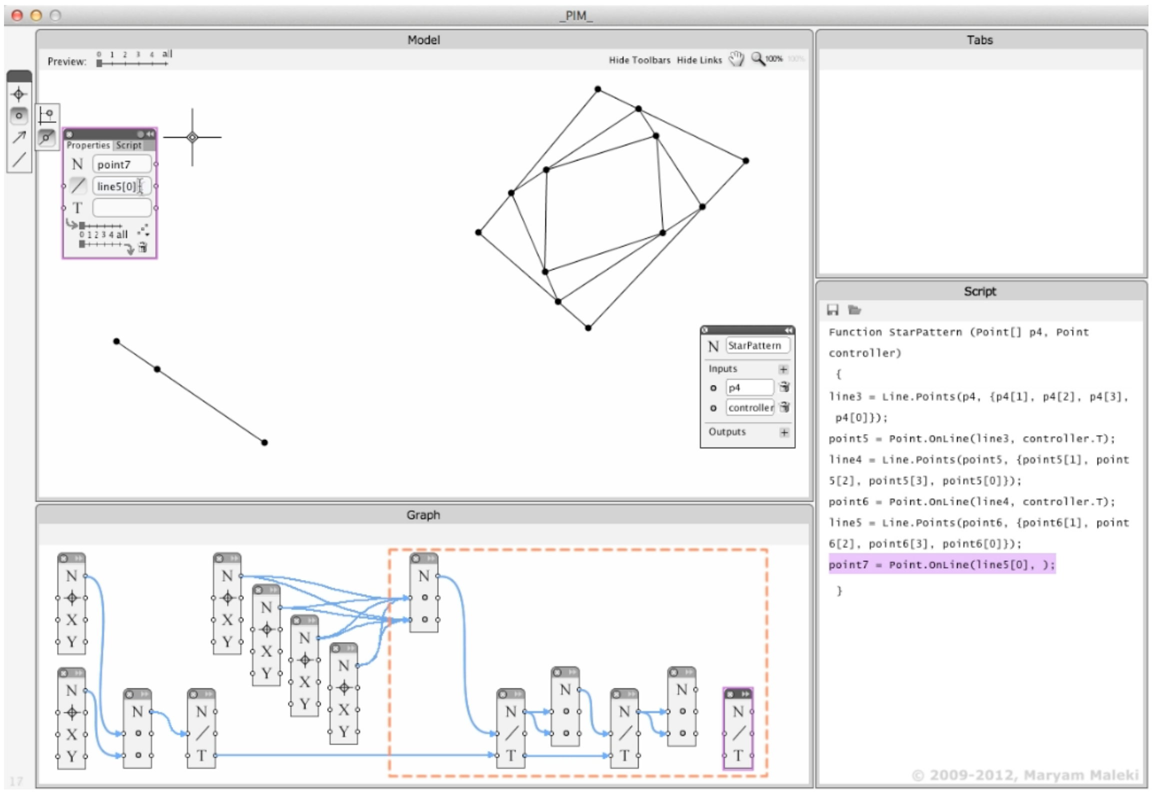


```
Function StarPattern ( )
 {
 }
```

Figure 4.14: Initializning a function from scratch opens an empty function boundary in the script and graph, as well as a blank function toolbar in the model for optional inputs and outputs.

function, which may result in incorrect guesses. A manual generalization or a combination is more desirable.

Figure 4.16(a) shows an example of a dependency graph that represents a model. The function boundary in a parametric model may cross a few dependency links, both incoming and outgoing from its inside nodes. In this example, a−g, b−d, and e−f are incoming links and f−i and k−l are outgoing links. Input links can be defined in three ways. A link defined as an *argument* will be independent from the current model and act as a generalized input link for the function. If a link is defined as a *global variable*, it stays connected to the original object in the model. Whenever this function is called, that input will come in from that object. Of course, the use of global variables restricts the reusability of functions because of the reference to an object in the current model – but end-user programmers regularly use global variables. The third way of solving an incoming link is to define it as a *local variable*

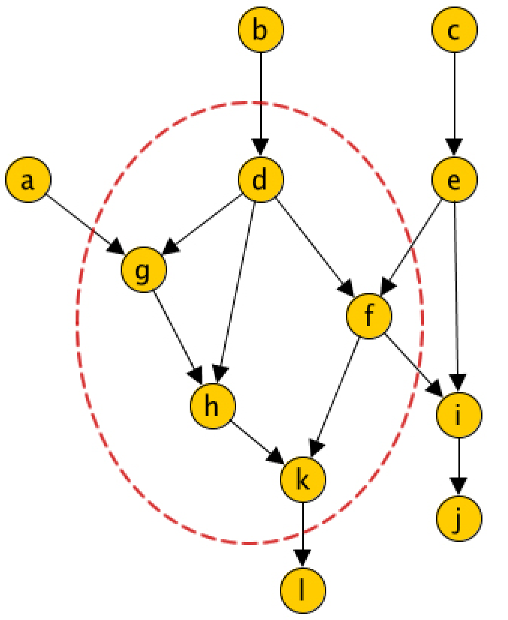


```
Function StarPattern (Point[] p4, Point
controller)
 {
line3 = Line.Points(p4, {p4[1], p4[2], p4[3],
 p4[0]});
point5 = Point.OnLine(line3, controller.T);
line4 = Line.Points(point5, {point5[1], point
5[2], point5[3], point5[0]});
point6 = Point.OnLine(line4, controller.T);
line5 = Line.Points(point6, {point6[1], point
6[2], point6[3], point6[0]});
point7 = Point.OnLine(line5[0], );
 }
```

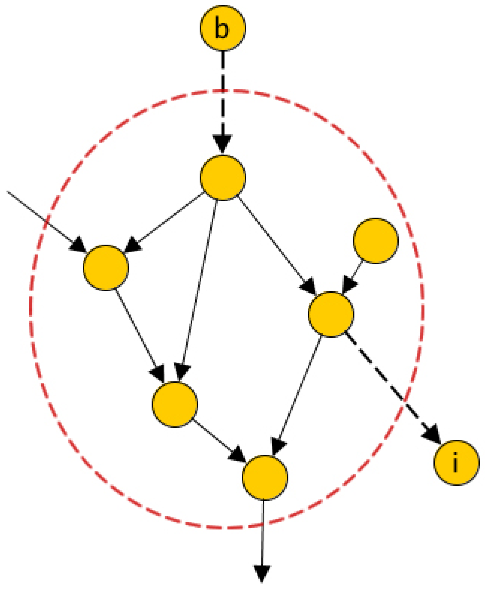
Figure 4.15: The function interface keeps the nodes in the boundary of the function, while the user works in the model, the graph, or the script window.

by taking its current value and defining a local variable in the function that carries that value from now on. This third method cuts the link and removes the need for an input. In this example, let us define link a−g as an argument, link b−d as a global variable, and link e−f as a local variable. The result is shown in Figure 4.16(b). The generalized function now has an argument (coming in from the top-left in the diagram), a global variable (coming in from the top), connecting it to object b in this model, and a local variable, meaning that the current value of the node on the top-right is used in all instances of the function, hence no incoming link.

As for the outgoing links that cross the boundary of a function, the user can decide to cut them, which means no output here, or keep them, which means the function has a side effect on the current model. Just like global variables, side effects also limit the generalizabilty of functions as they create permanent connections between the function and parts of the model. A function may or may not have a return value. The function in this



(a) The user puts an abstract boundary around a piece of the model to be generalized as a function



(b) Then (s)he defines the type of inputs (arguments, global variables, local variables) and outputs (return value, side effect)

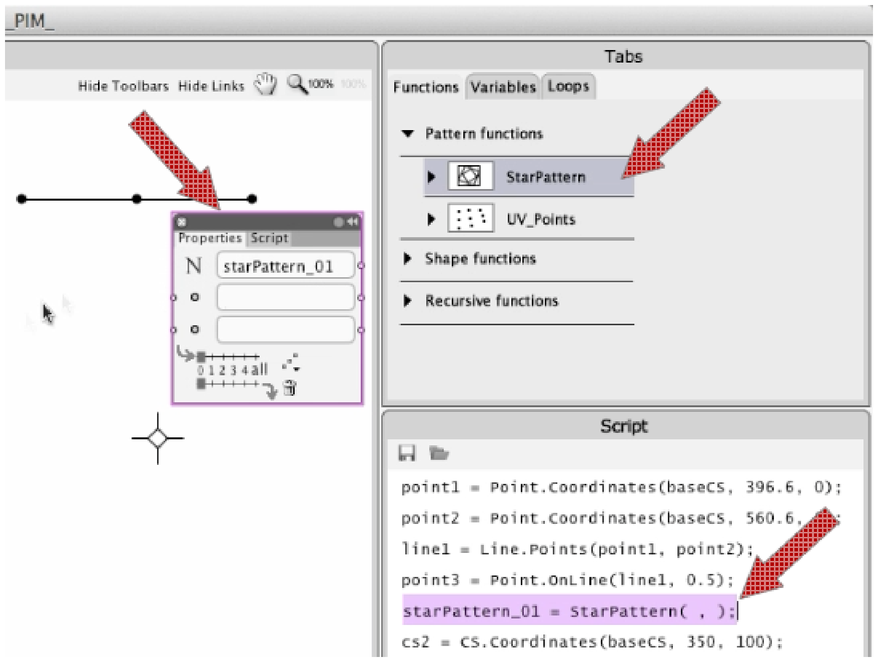
Figure 4.16: Defining a function a parametric system

example returns the object in node k and has a side effect on node i in this model.

4.5.2 Reusing functions

The window on the top-right side of the PIM prototype, which is currently empty, is where I envision non-geometric objects such as functions and loops will be located. Functions can be saved independent from the current file for future use in other models. Consequently, they can be imported into the current file and appear in the function tab. Each function in the function tab has a thumbnail that shows a preview of what the function does as shown in Figure 4.17. For most functions, this preview includes the output geometry of that function.

Function calls in a PIM-like system can be similar to creating new geometric objects. The user selects a function from the function tab and the function appears in the model



```
point1 = Point.Coordinates(baseCS, 396.6, 0);
point2 = Point.Coordinates(baseCS, 560.6,
line1 = Line.Points(point1, point2);
point3 = Point.OnLine(line1, 0.5);
starPattern_01 = StarPattern( , );
cs2 = CS.Coordinates(baseCS, 350, 100);
```

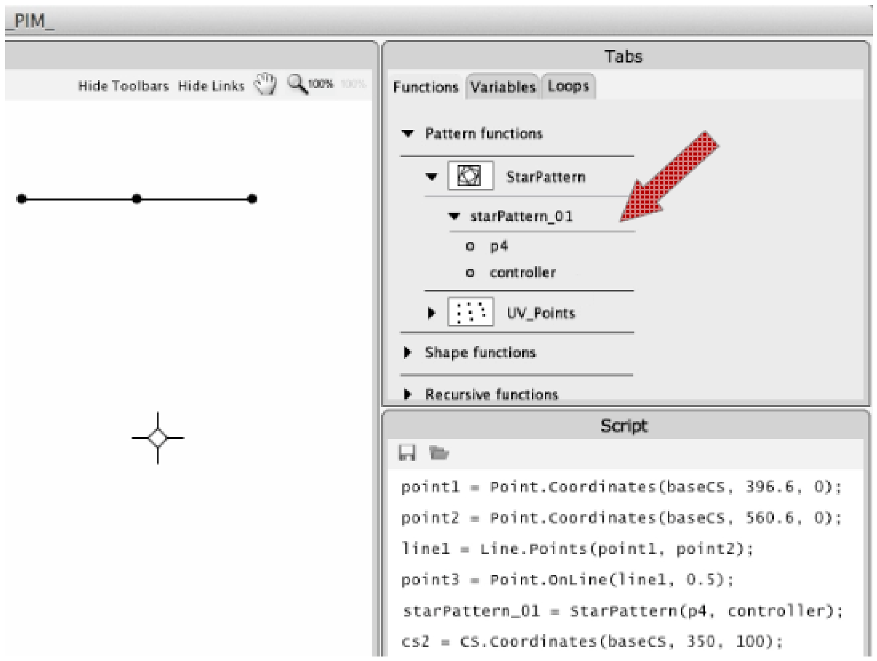
Figure 4.17: Function tab on the top-right shows a list of functions in the model with a preview of their outputs. Selecting a function from the function tab creates an instance of the function, with a toolbar consistent with regular objects' edit toolbars.

in the form of a create/edit toolbar with empty input fields (Figure 4.17). When the user provides enough inputs, the system shows a preview of of the result of the function.

Instances of functions that have been used in the model appear in the function tab with the list of their input and outputs (Figure 4.18).

4.6 Summary of chapter

In this chapter, I presented Programming In the Model features, categorized as localization, liveness, and lookahead features. I use the PIM prototype when possible to demonstrate the features, as some features are not implemented in the prototype yet. I included a brief note about implementation of some of these features in the prototype. I used creating and using functions to demonstrate how a PIM-like system would handle scripting tasks. In the next two chapters, I present the process and results of two formal evaluations of PIM.



```
point1 = Point.Coordinates(baseCS, 396.6, 0);
point2 = Point.Coordinates(baseCS, 560.6, 0);
line1 = Line.Points(point1, point2);
point3 = Point.OnLine(line1, 0.5);
starPattern_01 = StarPattern(p4, controller);
cs2 = CS.Coordinates(baseCS, 350, 100);
```

Figure 4.18: When a function has been used in the model, it shows the instance in the function tab with a list of objects it uses as inputs and objects it returns as output.

Chapter 5

User Study

In order to evaluate Programming In the Model features, I conducted a formal qualitative user study in the 2012 and 2013. 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. I used the PIM prototype to demonstrate the features to the participants and showed them a video demo of the features that were not yet implemented in the prototype. The prototype also allowed them to work with the system and have a hands-on experience with PIM features. In this chapter, I describe the methods I used in the study, the evolution of the study design, and the final format of the study. Then I explain the data analysis and the results of the study.

5.1 Methods

Participant observation

I used participant observation method in the user study, with the participant observer playing the role of a tutor. This was necessary in order to minimize the effect of learning an unfamiliar system on user's experience.

We collected different types of data from the participant observation (Spradley, 1980), including:

- Observation notes
- Interviews (post-task discussion)

- Questionnaires (background information, experience, etc.)
- Recordings of user's interaction with the researcher, especially when they asked questions or needed assistance with a feature
- Screen captures during the tasks as context for the recording and observation notes
- Checklists (Looking for certain items in the observation such as cognitive dimensions and barriers)

Protocol studies

In protocol studies, verbal reports are taken from subjects during a task. It makes explicit what implicitly goes through people's minds and is the best way of extracting what goes through working memory (Ericsson & Simon, 1993; Boren & Ramey, 2000). There are two types of protocol studies:

- Think-aloud protocol: Subjects verbalize anything that goes through their minds including actions and thoughts
- Talk-aloud protocol: Subjects verbalize their actions but do not provide any explanation

I used both types of protocol studies in the user study, when participants worked with the PIM prototype. I asked them to say out loud what they were doing and thinking, especially when they were confused, did not know what to do, or had a thought about a feature. I was particularly looking for instances of barriers happening and whether they were caused by PIM features or any other part of the interface. Ko et al.'s (2004) description of what goes through the user's mind when each type of barrier happens were my guide in identifying them, for example, statements similar to *"I think I know what I want the computer to do, but I don't know what to use..."* or *"I think I know what things to use, but I don't know how to make them work together..."* were indications of selection and coordination barriers.

I collected the following types of data from a think/talk-aloud sessions:

- Observation notes
- Voice recording of user thinking/talking-aloud
- Screen capture to give context to the recording

5.2 Pilot studies

The main goal of developing the prototype was to demonstrate PIM ideas to the users and experts in a meaningful way that can be understood and evaluated. At this stage, I was interested in users' reactions to these new methods and their feedback. I wanted to know if any of PIM features seemed beneficial to the users and whether they could see them being used in their design process.

The user study had four components:

- Pre-task questionnaire to gather some information about user's background and expertise
- Demo of the PIM features and how they work
- Tasks in which participants worked with the prototype and experienced PIM features first-hand
- Post-task feedback from the user on PIM features

I conducted three pilot studies before the main study. These sessions revealed minor and major problems with the prototype that had to be resolved before the formal sessions. These included implementation bugs and errors and confusing or hard-to-use interface elements. Also, the design of the user study evolved and changed after each round of pilot studies. Here is a brief summary of how I designed and finalized the four components of the user study.

Pre-task questionnaire

Questions were designed to familiarize us with user's area of work and education, experience with parametric CAD systems and their scripting environments, and knowledge of general purpose computer programming.

Demo and tutorial

In the first pilot study, the researcher gave a comprehensive demo and tutorial at the beginning of the session. Covering all aspects of the interface as well as all the important features of PIM took about 20-30 minutes, which was too long and somewhat boring for the participants. Also, the how-to portion of the demo got mixed up with the descriptions of PIM features. At the end, most of the focus and questions of the participants were about the interface and how to use it instead of PIM features. Lastly, the incomplete parts of the interface (e.g. the dependency graph) got a lot of unwanted attention from the participants and seemed to affect their perspective of PIM. It was hard for them to imagine a complete system when all they could see was what was implemented in the prototype. After discussions with my supervisors, I decided on two changes. First, I produced a video of PIM features with a mockup of the complete interface, with each feature clearly labeled and explained. With the video, I was able to separate the demo from the tutorial and give enough emphasis on PIM features, while giving the participants a view of the ideal system in its completed state. Second, I divided the tutorial portion into three small segments and spread them throughout the session, so that the user would get to work with the prototype in the first five minutes of the study session.

Tasks

In the first round, I chose two modeling tasks from a corpus of known and representative issues, in particular Woodbury's (2010) design patterns, and modified them for PIM's two-dimensional space. These were quite large tasks that needed about 15-20 minutes each to be completed. The pilot studies showed that the complexity and size of the tasks required the participants to focus almost all of their attention on the details of the tasks and small scale interactions with the interface in order to get the tasks done. This meant that the users were unable to pay attention to the 'big picture' that was the ideas behind PIM. The feedback that they gave us at the end of the sessions were mostly about the usability of the system and not PIM features. At this point I decided to use smaller tasks with less complicated details and keep the focus on PIM features and not on unimportant details of the interface. I designed three sets of tasks with four tasks each. The first set was for practice and used basic interactions with the interface. The second set was focused on scripting, both in the script window and the model window. The last set covered replication, dependencies, and preview. Each set of tasks followed a short demo of the features needed for those tasks.

Post-task feedback

My first choice for collecting users' feedback was to give them the Cognitive Dimensions questionnaire (Blackwell & Green, 2000) (See Appendix E). This is a well-established questionnaire for evaluating interactive systems, with each set of questions focusing on one of 13 dimensions. The initial results showed that the questionnaire was too long and some of the questions were irrelevant to the system and the type of tasks in the study. I shortened some of the questions and eliminated others altogether for the next pilot study. However, at the end, the questionnaire proved to be unsuitable for this study. The questions were designed for a user who had worked with a system for a while and had a deeper understanding of it than was possible to achieve in thirty minutes in a study session. Also, the time it took to fill out the questionnaire did not leave us enough time for a verbal discussion with the participants. So I decided to remove the questionnaire from the study and spend more time on a semi-structured interview or discussion with the participants, with the focus on PIM features.

5.3 Study description

In order to evaluate and improve PIM's design, I conducted a qualitative user study, with both HCI and domain experts as participants. In this section I present the final study plan. I have included the study documents, including the consent form, the pre-task questionnaire, and the tasks in Appendices A and B.

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 the 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, the participant selection provided a means for me to receive valuable critiques of the design choices and directions for future development. I recruited participants until I reached data saturation: that is when I was 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 (a $20 gift card) for their time.

Procedure

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

Part 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.

Parts 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 relationships between objects, in the model window, in the script window, or both. The following two tasks are representative of the broader tasks set (see Appendix B for a complete list of tasks):

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. I 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.

Part 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 events that happened during the tasks. Throughout the study, my 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.

Data

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 observed the sessions and took notes. The data that I collected in the study included a written pre-task background questionnaire, audio (voice) and video (screen capture) files of the demo and the tasks, and an audio file of the post-task open ended interview (discussion). I describe the data analysis process later in this chapter.

5.4 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 (Cooper et al., 2007). Because PIM is a new system, there are no users who are experts with it. As a result, all of the participants started out as PIM novices. In order to scaffold participants into users that could be considered intermediate or expert users of the system, more in line with typical users of CAD systems, I gave them short demonstrations of the functionality in the PIM system. I 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.

The study was purposely qualitative and observational in nature because I wanted to explore how users interacted with features within PIM and what they thought of the features. I was interested in (dis)confirming the features and their relative perceived importance in order to refine and develop PIM. Thus, I was not interested in task performance, completion times, and counts of user errors that one might quantitatively assess. Instead, I wanted detailed observations and thick descriptions that come from an exploratory qualitative study, where I was able to observe users' reactions to PIM features, as well as ask HCI experts to evaluate the usability of these ideas. I 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.

5.5 Reliability and validity

In quantitative research, reliability determines if the research can be replicated and if the measurements remain the same. Validity determines whether the research actually measures what it is intended to measure and how truthful the results are (Kirk & Miller, 1988). However, qualitative researchers seem to find these measures inadequate for or not applicable to qualitative research.

> "While the terms Reliability and Validity are essential criterion for quality in quantitative paradigms, in qualitative paradigms the terms Credibility, Neutrality or Confirmability, Consistency or Dependability and Applicability or Transferability are to be the essential criteria for quality."(Lincoln, 1985)

There is no universal agreement on what reliability and validity mean in qualitative research and how they can be tested. Golafshani (2003) tries to answer these questions by surveying the qualitative research literature.

In terms of reliability, some believe that a demonstration of validity is sufficient to establish reliability, since there is no validity without reliability (Lincoln, 1985). Stenbacka (2001) argues that since reliability deals with the issue of measurement, it has no relevance in qualitative research.

The concept of validity is sometimes explained in qualitative research by other terms such as rigor, quality, and trustworthiness (Lincoln, 1985; Seale, 1999; Stenbacka, 2001; Davies &

Dodd, 2002). The quality of the research and the trustworthiness of the researcher provides confidence in the findings and claims the study as part of proper research.

The question now is how to test or maximize the validity and as a result the reliability of a qualitative study? The ability to generalize findings to wider groups and circumstances is one of the most common tests of validity for quantitative research, but not all qualitative researchers find it applicable to their studies (Maxwell, 1992). Instead, triangulation methods are used in qualitative research for evaluation of findings and controlling bias (Mathison, 1988). Triangulation strengthens the study by combining different kinds of methods or data and giving multiple perceptions about a single reality (Patton, 1990).

> "Triangulation is defined to be a validity procedure where researchers search for convergence among multiple and different sources of information to form themes or categories in a study."(Creswell & Miller, 2000)

In this qualitative study, I employed several triangulation techniques (Denzin, 1970).

- **Data triangulation:** I recruited twelve participants in the study, with different scripting skills, diverse views regarding scripting in CAD, and different occupations, including students, professors, and practitioners. I reached data saturation after twelve sessions, as no new data was emerging.

- **Investigator triangulation:** I was the only investigator present in the sessions, but during data analysis, I had two co-researchers reviewing the coding and analysis process with me. While I was the only person coding the entire data set, one of my co-researchers partially coded the data, in order to increase our confidence in our codes and themes.

- **Theory triangulation:** I coded the data twice, once with a grounded theory approach without any predefined lenses, a second time looking for instances of barriers and cognitive dimensions. The second round did not aid the analysis, as explained in the chapter. So in spite of having planned for more than one theory to use, I did not have a successful theory triangulation in this study.

- **Methodological triangulations:** I had multiple methods of gathering data, including participants' talk-aloud/think-aloud data during the tasks, my observations of the tasks during the session and afterward when watching the videos, and interviews at the end of the sessions.

All of these methods increased my confidence in the results of the study. However, I acknowledge the following threats to the validity of this research.

Confirmation bias

Threat: Did I design the study in a way that it would confirm my point? Did I choose the tasks that make PIM look easy to use? Did I avoid using the tasks that would reveal PIM's shortcomings?

Response: The tasks were chosen to expose the participants to different features of the PIM prototype. Due to the reasons explained earlier, I chose simple, single action tasks that each targeted a feature and were feasible within the limitations of the system. So I only ever intentionally rejected a task when it caused the system to fail and crash. As I was not quantitatively measuring users' performance (speed, accuracy, errors, and so on), I was not concerned with whether the tasks were difficult or easy. The point was to demonstrate features and have a discussion about them.

Threat: Although I claim to have coded the data in the first round without any expectations, did my proximity to the research and my knowledge of the frameworks influence the coding process? Did I record positive events and comments more than I did negative ones?

Response: The second coder that partially coded the data gave us confidence about the coding process. The analysis of the coded data were done in a group and underwent several rounds of discussion and revision.

Response: In regards to recording positive and negative events, I believe that there might have been a reverse threat in recording the events. When the task is in progress and everything is going well, there is usually less recording of data than when something goes wrong. For example, I did not recorded the following positive events: "The user successfully initiated the action." "The user managed to choose the right input." "The user found the command she were looking for." "The user correctly typed the syntax." These events happened many times during a task and did not indicate anything of importance. But as soon as something did not happen as expected, users got confused or asked for tips, things were hard to find, or wrong syntax was entered, it was deemed negative and recorded. So I am not concerned about this threat as negative events were very hard to miss or ignore.

Participant observer effect

Threat: Did the fact that I had created the prototype made the participants hesitant to criticize it (author's effect)? Did they say positive things about the features to make me happy? Did the fact that I knew some of them before the study increase the possibility of author's effect?

Response: This is a valid threat to be concerned about, but hardly avoidable because of the limited number of candidates with the qualifications that we were looking for. Most of the people in this pool either knew me or my supervisor, in person or by reputation. My strategy was to explicitly ask them to give their honest opinion without any consideration for my feelings. By looking at the data, I was mostly satisfied with the fact that although the positive comments may have been a little exaggerated, the participants did not hold back on their negative feedback.

Threat: Did I influence their perception of PIM's ease of use when I tutored them? Did I lead them in liking some features when I taught them how to use the system?

Response: I admit that it took effort to tutor someone to use a system I had created, without leading them in one direction or another. I am not a seasoned participant observer and this was an invaluable experience in self control and also in acknowledging my shortcomings when I watched the videos. If I were to do this study again, I would train someone to tutor the participants and I would only be a silent observer during the tutorials and tasks.

5.6 Data analysis

I used NVivo10 software to sort, transcribe, and analyze the data. In the first round of data analysis, I used open, axial, and selective coding from grounded theory. (Charmaz, 2006). With no predefined codes, I went over the data and let it speak to me. I recorded any meaningful event or statement that I came upon and I ended up with a list of short sentences, ordered alphabetically. Examples of those initial extractions are available in the first column of Table 5.1.

At this point, I reviewed the data that I extracted and tried to find common themes and possibilities for grouping the statements and events (Miles & Huberman, 1994; Creswell & Creswell, 2007). 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. Here I explain what each of these categories were and why I chose them. The sample data coding in Table 5.1 helps explain these codes.

I turned the background data I collected in the pre-task questionnaire into classifications for each participant. With this method, all the video files and later their transcripts had user classification data assigned to them, such as gender, age, programming and CAD background, and so on.

Whether a piece of data was extracted from my observations or from users' verbal comments were important, because a verbal statements may be influenced by the author's effect, meaning that it was said to please me, while an observation of a task can reveal true problems and barriers. However, my interpretation of an observation my be influenced by my bias and/or my knowledge of the system. Keeping track of the nature of the data helped look for inconsistencies in the analysis.

I 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.

And finally, I was interested to know which PIM feature the data was related to. These all became codes (or nodes in NVivo10). Each piece of transcript text and the corresponding portion of the video/audio were tagged with one or more code(s) from each category.

As I mentioned earlier, I set aside the frameworks I had studied and looked at the data on its own. I recorded anything that came to mind when watching the videos and listening to the discussions. Some things appeared quite often in the data, others were more sparse but worth recording nonetheless. I put similar codes into themes, as follows.

- General reaction *(Reaction to the features, liking or disliking them)*
- Usability issues *(When a usability problem was observed or talked about)*
- Comparison *(When participants compared PIM or its features with another system (CAD or else) or when they talked about other systems trying to explain something)*
- Suggestion *(When participants suggested a new feature, or an extension or improvement to an existing feature)*
- Complexity *(When Participants raised concern about scaling up PIM in real world systems and its application to complex CAD models)*

- Concrete-Abstract *(When Participants mentioned (directly or indirectly) the abstract nature of the script and/or the concrete nature of the model)*
- Mental model *(Anything related to users' understanding of the system and whether it does or does not match with their mental model)*
- Legibility *(Anything related to the legibility and readability of the system)*
- Fear of code *(When participants referred to the fear or hesitation to use scripting (Also related to attention investment model))*
- Learning *(Any statement or event related to learning activity, intuitive or progressive)*
- Search-Navigation *(Any mention of or action that entailed searching and navigating in the model and/or script)*
- Shift of focus *(When the issue of shift of focus was raised by the participant or observed during the tasks)*
- Errors *(Any reference to error prevention or detection)*
- Design exploration *(When participants talked about design exploration or trial and error in design)*
- Gentle slope *(When an instance of gentle slope was observed or referred to)*
- Screen real estate *(When participants raised concern about the verbosity of PIM)*
- Other *(Any data that I could not label, but did not want to discard.)*

In spite of my best efforts, I cannot be sure if these codes purely stood out from the data or came partly out of the earlier lenses I had. Nonetheless, they were strongly evident in the data. A second coder (one of my supervisors) also partially coded the data (about 35% of the data) to ensure that we were not missing anything. His codes were either similar to or high level groupings of my codes.

Data	Participant	Nature	Type	Feature	Theme
Participant compared preview with undo in other systems	#06 male 40-49 age architecture education 10 yrs CAD 1 yrs scripting	Statement	Neutral	Preview	Comparison
Participant got confused about what happened in the model during scriptin	#04 male 20-29 age architecture student 2 yrs CAD 3 months scripting	Event	Negative	Liveness	Error
Participant said: *"Live scripting/modeling makes scripting more like drawing and drawing more like scripting"*	#09 female 20-29 age architecture 5 yrs CAD 2 yrs scripting	Statement	Neutral	Liveness	Concrete-abstract Mental model
Participant said: *"Script tab in the model makes code more tangible and sorts code in 3D."*	#11 male 30-39 age architecture 10 yrs CAD 10 yrs scripting	Statement	Positive	Script tab	Concrete-abstract search-navigation
Participant switched to modeling tools in the middle of scripting	#04 male 20-29 age architecture student 2 yrs CAD 3 months scripting	Event	Neutral	Liveness	Shift of focus Gentle slope Other
Participant moved the toolbar so that the outgoing links all go to the right and are all visible	#01 female 20-29 age computational design 7 yrs CAD 3 yrs scripting	Event	Neutral	Dependencies; Localized information	Usability Other
Participant made a mistake identifying which object is downstream of this point	#01 female 20-29 age computational design 7 yrs CAD 3 yrs scripting	Event	Negative	Dependencies	Error Mental model Search-navigation

Table 5.1: Coding a sample dataset

When all of the above codes were agreed upon, I went over the data again and coded it with this coding scheme. Table 5.1 shows the codes for a sample dataset. With the data tagged as such, I was able to extract information from it and look closer at different aspects of PIM. Queries in NVivo10 allowed me to view a report of, for example, *positive statements* said about *dependencies in model*, or any *event* regarding *liveness* that constitutes a *barrier.*

I planned to have a second round of coding and use the principles and frameworks that I had used to guide me in designing and creating PIM as codes, including Ko et. al.'s (2004) six learning barriers in end-user programming, Blackwell's (2002) attention investment model, and Green's (2000) cognitive dimensions of notations, as described in Chapters 2 and 3.2. Unfortunately this round of coding did not produce any usable results, due to the following reasons.

As described earlier, cognitive dimensions of notations are not events that happen and can thus be observed. They are descriptors that a system either is or is not, or have or does not have. And unless the participants are aware of the framework and have experience with it, we rarely hear them mention the dimensions in their comments and feedback. These dimensions are more suitable for evaluating the general state of a system by experts. They require more time and deeper understanding of the system if they are to be used by the users. For that reason, instead of coding the user study data with the dimensions, we arranged a focus group with expert participants to evaluate PIM against the cognitive dimensions. The description of the focus group study appears in Chapter 6.

The six learning barriers in end-user programming were more tricky. These barriers did occur, but it was hard to identify whether they happened because of the effects of learning a new system or they were true results of PIM features. The study was designed to expose users to a new approach to scripting in CAD and find out their initial reaction to it and their feedback about it. It was not powered for observing their learning process and identifying learning issues. The original study of learning barriers in Visual Basic (Ko et al., 2004) was conducted in the context of a university course with more time to teach and learn the system and more opportunities to observe and identify the barrier.

5.7 Results

After studying the codes that had emerged in the analysis, I grouped them into higher level themes. I explain them in detail under the headings of *search-navigation*, *learning*, *fear of code*, *mental model*, *errors*, *trial and error*, and *complexity*.

As this was not a quantitative study, the numbers did not test a hypothesis. Rather, they provided a way to look at the data more closely and make sense of it. When a number was high, it meant that there was probably something happening that ought to be investigated. However, the low numbers did not mean *not-important*. I looked at the data as a whole and reported what I deemed important and relevant to Programming In the Model.

Other than the themes mentioned above, there were comments and observations that did not relate to PIM features per se, but had more to do with the design decisions I had made in the design of the PIM prototype interface. I call these *usability issues* and explain them later in this section.

General reaction

I separated the data into what I had observed during the tasks and what participants had said either during the talk aloud or in the discussion after the tasks. Table 5.2 shows how many times observations and feedback got tagged as negative, neutral, or positive. When I watched the tasks after the study, I mostly focused on when things went wrong or unexpected events happened. There were many instances when everything worked, e.g. participants found the objects easily, knew what tools to use, and performed the task successfully. But what really mattered was when things did not work well or as expected, or when participants used the features in surprising ways.

	Negative	Neutral	Positive
Feedback	9	33	71
Observation	11	21	16

Table 5.2: General reaction

Table 5.3 shows how many times observations and feedbacks were tagged as negative, neutral, or positive over the course of the study and breaks it down by features. For example, the fourth row of the table shows that displaying dependencies in the model received a total of 13 negative tags: 6 negative comments stated verbally by users and 7 negative situations that I observed of the users' interactions. The same feature received a total of 18 neutral tags: 8 neutral statements by participants and ten neutral situations that I observed. The feature also received 13 positive tags comprised of 12 positive statements and 1 positive interaction.

	Negative	**Neutral**	**Positive**
	Total (Feedback, Observation)		
Localization			
–properties in model	3 (1, 2)	15 (2, 13)	4 (4, 0)
–script tab in model	1 (1, 0)	1 (1, 0)	6 (5, 1)
–dependency in model	13 (6, 7)	18 (8, 10)	13 (12, 1)
Liveness			
–brushing/highlighting	0 (0, 0)	4 (4, 0)	16 (13, 3)
–realtime updating	0 (0, 0)	7 (4, 3)	39 (32, 7)
–functions	0 (0, 0)	0 (0, 0)	1 (1, 0)
Lookahead (preview)	2 (2, 0)	4 (4, 0)	7 (7, 0)
General	3 (0, 3)	14 (11, 3)	9 (8, 1)

Table 5.3: Type of feedback for PIM features. # Total (# Feedback, # Observation)

Liveness of the interface received positive feedback from the participants. They were mostly pleasantly surprised to see the live interface and referred to their past experiences with other parametric CAD systems' *un-live* windows (including GenerativeComponents, Grasshopper, and CATIA) and the problems they had with them. They mentioned the code being closer to the model, and finding and making sense of things being much easier. Brushing and highlighting within the live interface also received high praise, as well as localized information display in the model window (including edit toolbars, script tab, and the idea of dependencies in the model). The actual implementation of dependencies in the model view did not receive a good feedback. Preview also received mixed feedback from the participants. I will discuss these points in more detail in the following pages.

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)

	Negative	**Neutral**	**Positive**
Localization			
–properties in model	0	0	1
–script tab in model	0	0	3
–dependency in model	0	2	7
Liveness			
–brushing/highlighting	0	3	13
–realtime updating	0	1	8
–functions	0	0	0
Lookahead (preview)	0	0	0
General	0	0	1

Table 5.4: Number of times data was tagged for search-navigation

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 me after the first few scripting tasks that they performed using the prototype. Some participants did not even wait for me 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)

	Negative	**Neutral**	**Positive**
Localization			
–properties in model	0	0	1
–script tab in model	0	0	0
–dependency in model	0	0	0
Liveness			
–brushing/highlighting	0	0	1
–realtime updating	0	1	13
–functions	0	0	0
Lookahead (preview)	0	0	0
General	0	1	2

Table 5.5: Number of times data was tagged for learning.

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 does the same thing as the model:

"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)

And this quote expresses the participant's feeling about scripting: *"So I guess this is like the middle layer in Dante's Inferno, when programming is real hell. This is better at least because we have that middle layer."* (P10)

	Negative	**Neutral**	**Positive**
Localization			
–properties in model	0	0	0
–script tab in model	0	0	0
–dependency in model	0	0	0
Liveness			
–brushing/highlighting	0	0	0
–realtime updating	0	1	8
–functions	0	0	0
Lookahead (preview)	0	0	0
General	0	3	0

Table 5.6: Number of times data was tagged for fear of code

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. I 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 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 my disappointment, PIM caused some errors during the

	Negative	**Neutral**	**Positive**
Localization			
–properties in model	0	0	0
–script tab in model	0	0	0
–dependency in model	0	0	1
Liveness			
–brushing/highlighting	0	0	2
–realtime updating	0	0	8
–functions	0	0	0
Lookahead (preview)	0	0	2
General	0	0	0

Table 5.7: Number of times data was tagged for errors

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 I 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 I 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.

They also requested more selective preview, not only selecting downstream levels, but

	Negative	**Neutral**	**Positive**
Localization			
–properties in model	0	0	0
–script tab in model	0	0	0
–dependency in model	0	0	0
Liveness			
–brushing/highlighting	0	0	3
–realtime updating	0	0	9
–functions	0	0	0
Lookahead (preview)	0	1	4
General	0	0	0

Table 5.8: Number of times data was tagged for trial and error

also selecting one or more objects in the model and previewing those only.

The live interface also seemed to help in design exploration.

"When I was working with Catia, there were times when I had to go to scripting, but I didn't know what happened in the model. So every time I had to go between Catia and scripting and call it to see how it was changing. It was too much time running between two platforms. But here I can see them side-by-side." (P2)

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 PIM would handle certain aspects of the features in real models, for example, which object will carry a script in the script tab when it defines multiple objects, or how it would highlight an object in the script when it appears in multiple places.

	Negative	**Neutral**	**Positive**
Localization			
–properties in model	0	0	1
–script tab in model	0	1	1
–dependency in model	2	1	4
Liveness			
–brushing/highlighting	0	0	1
–realtime updating	0	0	0
–functions	0	0	0
Lookahead (preview)	2	1	1
General	0	2	2

Table 5.9: Number of times data was tagged for complexity

"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

I took note of the usability issues when they were observed in the tasks or noted by participants. These arose from my interface design choices and development limitations. The most important usability issues for me 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. I 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 I believe to be the choice of visual representation.

	Usability
Localization	
–properties in model	9
–script tab in model	1
–dependency in model	19
Liveness	
–brushing/highlighting	2
–realtime updating	1
–functions	0
Lookahead (preview)	2
General	2

Table 5.10: Number of times data was tagged for usability

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). I tried to incorporate these suggestions and requests that I deemed effective in improving the usability of the prototype. Other issues remain unresolved, pending more investigation and redesign.

5.8 Summary of chapter

In this chapter, I described the formal user study that I conducted to evaluate Programming In the Model features. I explained the methods and rationale of the study, as well as the analysis of data and the results. As mentioned in the analysis section, I was not able to apply the lens of Cognitive Dimensions of Notations framework to the study, due to participants' limited time working with the system and the fact that these dimensions are not events that can be tagged, but descriptors of the system. But I still wanted to analyze PIM with the Cognitive Dimensions. So I arranged a focus group with HCI/CAD experts to do the analysis. I explain the process and the results of the focus group in the next chapter.

Chapter 6

Expert Focus Group

The cognitive dimensions of notation framework (Green, 2000) is a design and evaluation guideline for interactive systems. We used this framework to analyze five parametric CAD systems in a separate project and I wanted to find out how PIM features affect each dimension, positively or negatively. As I mentioned earlier in Chapter 5, the cognitive dimensions questionnaire was not effective for the short exposure that the users had with PIM in the user study and needed more in-depth knowledge and experience with the system. Later, I tried to use the dimensions as codes for analyzing the user study data, looking for instances when dimensions occurred. However, I soon realized that it was hard to observe and code cognitive dimensions during tasks because they are high level concepts that do not necessarily appear in user's actions. For example, a system may have a high viscosity (= a low level of liveness) but this is not something that *occurs* during a short task. Another example is closeness of mapping between the notation and the domain world that does not *happen* at a point in time, but may be high or low when we look at a notation in a system.

So I decided to gather a groups of experts in HCI and CAD and have a focus group analysis of PIM features with the lens of cognitive dimensions. A focus group (Morgan, 1998; Stewart, 2007) allowed me to have an open ended, but moderated interview with a group of people familiar with both the domain and the framework and gather significant amount of qualitative data in a short amount of time. The fourteen dimensions in the framework made it easy to *focus* the discussion and gather the shared understanding of the participants regarding their application to PIM. The structure of a focus group also made it possible for the participants to have a natural conversation (comparing with a one-on-one interview) and to build on each others comments (Stewart, 2007).

6.1 Description

Framework

Green's (2000) cognitive dimensions of notations was the lens chosen for this focus group. The goal was to analyze PIM features against each dimension and find out how they differ from other parametric CAD systems. Here is a summary of the dimensions. For more information, please refer to Chapter 2.2.2.

- Viscosity: *Resistance to change*
- Visibility: *Ability to view components easily.*
- Premature Commitment: *Constraints on the order of doing things.*
- Hidden Dependencies: *Important links between entities are not visible.*
- Role-Expressiveness: *The purpose of an entity is readily inferred.*
- Error-Proneness: *The notation invites mistakes and the system gives little protection.*
- Abstraction: *Types and availability of abstraction mechanisms.*
- Secondary Notation: *Extra information in means other than formal syntax.*
- Closeness of Mapping: *Closeness of representation to domain.*
- Consistency: *Similar semantics are expressed in similar syntactic forms.*
- Diffuseness: *Verbosity of language.*
- Hard Mental Operations: *High demand on cognitive resources.*
- Provisionality: *Degree of commitment to actions or marks.*
- Progressive Evaluation: *Work-to-date can be checked at any time.*

We did not include abstraction and secondary notation dimensions in the focus group analysis, simply because PIM does not offer any tools for either dimensions other than what is provided in the underlying CAD system, therefore there was no need to evaluate it against those dimensions.

Participants and moderators

I recruited five participants for the focus group. They were all familiar with cognitive dimensions and had intermediate or expert knowledge of at least one parametric CAD system. They were recruited from researchers at the Computational Design Lab at the School of Interactive Arts and Technology, Simon Fraser University.

I took the role of the moderator of the session. I demonstrated the system and made sure all dimensions were discussed in a timely manner and all participants had a chance to express their opinions. During the session, I refrained from expressing my own views on the topics discussed, so that I did not introduce any bias in the results. I will discuss other threats to the validity of the study later in this chapter. My co-advisor was present in the focus group session as an independent observer.

Procedure

After signing the consent form (Appendix C), specifically agreeing to take part in the focus group and have their identities revealed to other people in the group, participants filled out a questionnaire at the beginning of the session, providing general information about their age and gender, and background information about their expertise and field of work. After explaining the procedure to the participants, I gave a demo of the prototype and also showed them parts of a video demo of PIM. We then started discussing the dimensions, which were sorted based on their importance and relevance to our research.

The participants were given a written list of dimensions, with a short description of each (Appendix D). The session was divided into four segments, each covering three dimensions, in order to fit all the twelve dimensions in the two-hour session. In each segment, participants spent 5-7 minutes writing their analysis for the three dimensions in that segment, before discussing them in the remaining 15 minutes. This was to ensure that their initial impression of PIM was not deluded by comments from other participants and that everyone's thoughts and feedback were heard regardless of how much time they had to speak.

The moderator was responsible to keep track of time and prompt the group to move on to the next segment in time. In addition to the written comments, an audio recording of the session formed the main sources of data. Screen recording during the demos and a video, capturing what was drawn or written on the whiteboard during the discussions, accompanied the audio to provide context for the comments.

6.2 Reliability and validity

In this qualitative study, I employed several triangulation techniques (Denzin, 1970). (I explained in Chapter 5.5, why I used triangulation as a method of ensuring reliability and validity.)

- **Data triangulation:** I recruited five experts to participate in the focus group. Having more than one expert analyzing PIM gave me the opportunity to gather different opinions and to report when they agreed or disagreed on something.
- **Investigator triangulation:** There were two investigators present at the focus group session. I gave the demo and moderated the session, while the other researcher observed and took notes. Having a second investigator also reinforced my refraining from inserting my own comments and thoughts into the conversation. I was the only investigator who coded the data, simply because codes were predefined (cognitive dimensions) and easy to spot in the data.
- **Theory triangulation:** I did not use theory triangulation. This was a cognitive dimensions of notations analysis, therefore only one framework was used in analyzing the data.
- **Methodological triangulations:** I had multiple methods of gathering data, including participants' verbal discussions and their written feedback on each dimension. Having both the questionnaire and the discussion made it possible to hear everyone's individual opinions before they were impacted by others' opinions, and then observe those opinions challenged or confirmed by other participants.

I acknowledge the following threats to the validity of this research.

Confirmation bias:

Threat: I believe that there was minimal potential for confirmation bias in collecting or analyzing the data. I simply recorded everything and reported them under each dimension. But as I will explain shortly in this chapter, I did my own cognitive dimensions analysis of PIM features to accompany the expert's analysis. There is the possibility that my analysis is skewed towards confirming PIM's features. For that reason, I clearly distinguish between my own opinion and the result of the focus group analysis.

Author's effect:

Threat: Did the fact that I had created the prototype made the participants hesitant to criticize it? Did they say positive things about the features to make me happy? Did the fact that I knew them before the study and our previous work together increase the possibility of author's effect?

Response: This threat to validity was similar to the user study, but more pronounced in the focus group. At the time, I had no other option for finding participants that knew both cognitive dimensions and parametric modeling. If I have the opportunity to conduct another focus group, I would insist on only recruiting participants that did not know me and my work. Given the limitations in this study, I would at least exclude myself from the session and allow others to demo the prototype and moderate the session to limit my effect on the data.

6.3 Data analysis

The format of the raw data that I collected in the focus group was:

- A written pre-task background questionnaire for each participant
- A written document for each participant, containing their comments and thoughts about PIM separated by dimensions
- An audio recording of the discussions in the focus group session
- A video recording of sketches and notes written on the whiteboard in the room, that was synched with the above audio

I used NVivo10 to sort and analyze the focus group data. I applied similar categorizing codes as the user study to tag the data for *participant background*, *type* (verbal, written), and *PIM feature*. In addition, I coded the data with *cognitive dimensions* codes. These codes (or tags) made it easy to track the data based on the dimension that was discussed, the PIM feature that was mentioned, and the type of feedback that was received (Morgan et al., 1998).

6.4 Results

In this section I report the result of the expert focus group evaluation of PIM through the lens of the cognitive dimension framework. Cognitive dimensions are not rules for designing notations systems, but guidelines to keep in mind and evaluate the design with. Some dimensions are more important for some notations or some tasks than others. Improving the system in one dimension may result in reducing its score in another, something that the authors of cognitive dimensions framework call *"trade-offs"* (Blackwell et al., 2001; Green et al., 2006).

Due to the limited time we had with the expert group, we had to go over too many dimensions in a short time. Therefore, some cognitive dimensions did not get the depth they deserved in the discussions. Some PIM features also received little focus from the group. I, as an HCI and CAD expert, have my own opinions that may differ from those of the focus group participants. I did not offer my opinions and analysis in the focus group. But I present them here, separate from the opinion of the focus group participants. I admit that I may be biased in analyzing my own system. But I base my analysis first on the discussions of the experts in the focus group, then on my observations in the user study, and finally on my deep knowledge of the system.

Viscosity

Resistance to change. How much effort is required to perform a single change?

Participants in the focus group believed that viscosity in PIM is no different than viscosity in the underlying parametric CAD system. In other words, PIM features do not increase or decrease the inherent viscosity of parametric systems. Features such as functions and copy/paste reduce repetition viscosity but they are not specific to PIM.

"There are some things that are inherent from parametric modeling, e.g replication reduces repetition viscosity, deferred decision reduces premature commitment. PIM only has incremental effect on these. These may not be that relevant to PIM." (P1)

> **My analysis:** How easy or difficult it is to make a change in a CAD system depends on those characteristics of the system that are unaffected by PIM feature, such as the programming language and the type of parametric structure it uses. Liveness, for example, makes it easier to see the

> result of a change, but does not change how difficult it is to make that change.

Visibility

Ability to view components easily. An important component is juxtaposability, the ability to see any two portions of the program on screen side-by-side at the same time.

Participants noticed that highlighting and immediate updating of all views can play positive roles in increasing the visibility of the system.

"A key feature is that each window reflects the state of the others as soon as it can, e.g., having the right dialog box pop up as soon as an update method is chosen, the ability to juxtapose script, dataflow, and object dialogues on the model should help the user better understand how these interface types interact. Highlighting (brushing) works well, but the dataow highlighting needs to be more pronounced." (P1)

> **My analysis:** PIM's liveness presents different representations of the model to the user at the same time and highlighting across those representations makes the object of interest easily visible. Users can ask to view different types of data in the model at any time, accessing information that may be hidden in the other windows next to the model. All of this improves PIM's visibility.
>
> The price of presenting multiple representations on the screen at all time is a system that is diffuse, meaning it uses many symbols to present the data. We need more screen real estate to keep these windows open all the time. The trade-off for presenting localized information in the model view is that at times, especially with scale, the model itself may become less visible due to all the additional elements displayed on it.

They also commented about PIM's visibility issues when scaled up. Common questions we heard in the session were:

- How do you deal with a 20-page code? How do you highlight things that are not in one page? Does it jump between pages constantly?

  "I was just thinking, if you have a big model, and you change some small part of it and you have a long script that goes on for 20 pages, is it going to suddenly jump to

that part of the script? In other words, is the script sort of flashing back and forth? So how do you handle it? " (P3)

- How do you handle visibility of large, 3D models?
- What happens to the visibility of the model when there are several toolbars and links and scripts open on the model?

  "I wonder about a scenario when you have multiple dependencies, toolbars, sliders, script tabs, mini toolbars, are open simultaneously, would that compromise the visibility of the model itself?" (P5)
- Is highlighting enough to deal with change blindness problems in large models?

> **My analysis:** This is a a design problem that we would get to if/when we decide to take PIM forward. I will discuss this in more detail in the discussion of scalability.

Another topic of interest was the visibility of functions in PIM. The issue that was pointed out was whether we should display the function call or the body of the function, when an object created by a call to the function is highlighted. As functions and function calls were not fully implemented at the time, certain design decisions still had not been made and tested. My initial response was that the call is highlighted by default, and if requested by the user, the function itself will be opened. Participants seemed to think that this approach was not *"PIM-like"* and reduced the visibility of PIM.

> **My analysis:** I still think that a design could be possible where both the call and the function can be visible at the same time, when an instance is highlighted. This makes the function more visible, while maintaining abstraction when only the function call is needed.

Premature Commitment

Constraints on the order of doing things. Do programmers have to make decisions before they have the information they need?

We received mixed reviews about premature commitment. Some participants thought that PIM does a good job of reducing premature commitment, especially with preview.

"[Premature commitment is] low. There are no constraints or very little on the order." (P6)

Others said that PIM does not add anything to general parametric modeling when it comes to premature commitments. So strategies such as deferred decisions, and features such as functions reduce premature commitments, but they are parts of any parametric system.

"There are many issues that are in the nature of parametric modeling, for example making change in the beginning of modeling vs. at the end of modeling. If it's a long model, premature commitment becomes an issue. Because at the beginning you may think it's the most logical idea, but at the end, you may want to change it. I could not see any evidence here as to how PIM can enhance side effects of premature commitment." (P2)

> **My analysis:** One of the goals of PIM is to reduce premature commitment, by allowing the user to evaluate their scripting work in the model line by line, when they write a function. When they finish the function and want to commit it, they already know that it works and makes the model that they wanted. Live scripting in my opinion reduces premature commitment by revealing the information required to evaluate the work.

Hidden Dependencies

Important links between entities are not visible. If one entity cites another entity, which in turn cites a third, changing the value of the third entity may have unexpected repercussions (side effects). How easy is it to spot these side effects?

Once again, the question was whether PIM can handle hidden dependencies in large models as well as small ones, in 3D as well as in 2D, and in complex or long scripts as well as short, simple ones. Scalability seemed to be a dominant issue that reflected their evaluation of PIM in several dimensions.

"Scalability is an issue for any system, not just particular to this. How are you going to react when you start to get larger, more realistic model?" (P2)

> **My analysis:** I agree with the evaluation of the experts that PIM has a scalability issue and that dependencies will not be as visible and straightforward to follow when the number of objects grows. This is not only

> PIM's problem, but many other parametric CAD systems and visual programming environments have the same issue of scalability. At this time, I do not have a solution for this problem. More design and testing need to be done.

The problem with the visibility of functions and function calls, explained above, was linked to increased hidden dependencies.

"The function box is highly symbolic. For example, if a function makes 3 points and is used twice to make, say, p1, p2, & p3 and q1, q2, & q3, then pointing at, say, q2 should show both the second function call and the code in the function that makes q2. Dont be afraid of putting non-geometric objects in the dataow and in the model." (P1)

> **My analysis:** Whenever a function is used more than once in a parametric modeler, it creates hidden dependency, due to the fact that a change to the function may cause an unintended effect on one of the function calls. Systems that show the function as a node in the graph and its calls as downstream relationships, as we do in PIM, help the user know how many calls have been made to the function. In addition, PIM presents more detailed view of functions and their instances in the function tab, with information such as inputs that are used in the function call and outputs that it produces. Of course as it is our policy in PIM, highlighting any of these calls and input/output objects in the function tab, highlights them in the other views, making it easier to spot dependencies of function.

> **My analysis:** Reducing hidden dependencies and making side-effects more visible was one of our goals when we designed the preview feature. Parametric systems that offer a graph-like representation of dependencies make it easier to follow object relationships upward and downward in the dataflow graph. We take this one step further with preview by highlighting and bringing forward a subset of the graph that must be attended to. When preview is turned on in PIM, it highlights any object downstream that will be affected by the change to the current object. So before, during, and after the edit, but before finalizing it, the user can see all entities that may suffer from side-effects, not only in the graph view, but also in the

> script and most importantly in the model view. This feature I think was not fully explained by me and understood by the participants.

Role-Expressiveness

The purpose of an entity is readily inferred. How easy is it to answer the question "What is this bit for?" How easy is it to "read" the programming language?

One participant wrote that role-expressiveness in PIM is

"... really, really easy. Through showing an object in multiple representations, the user can access it in its most familiar mode. This enables discovery of 'what this bit does?' " (P1)

Another participant wrote:

"Role-expressiveness is consistent in each notation. But across notations, because naming is not revealed in the graph and the geometry, it is really hard to see what it is and what it is not. Naming could be confusing." (P2)

So the feedback was not consistent among the experts in the group regarding role-expressiveness in PIM.

> **My analysis:** Role expressiveness of entities in relation to the rest of the model can be problematic in parametric modeling. In addition to providing a node-link representation of the data structure in the graph window, similar to several other CAD systems, PIM offers the same data selectively in the model view. When the user is unsure about the role of an object and how it affects or is affected by other objects, the option is provided to ask for dependencies to be displayed right there, showing the objects themselves in the model, rather than the nodes that represent them in the graph. Besides eliminating the need to shift the focus from model to graph and back, turning on the dependencies gives an instantaneous glance into the role of the object and how it connects to other parts of the model.
>
> A side effect of displaying dependencies in the model is reducing the visibility of the model. Although dependencies can be turned on and off on demand and it only takes a few seconds to follow the links and find the

information about relationships, it still makes the model window more cluttered and its elements less visible.

Error-Proneness

The notation invites mistakes and the system gives little protection.

PIM's extreme liveness and the way in which it links different views were said to provide strong error protection. Participants commented that being able to work in the representation that is the least error-prone and avoiding the spaghetti code of dataflow reduces errors in PIM.

"PIMs extreme liveness provides strong error protection. Many PCAD mistakes are in mixing up the spaghetti code of the dataow. PIMs brushing helps here. By linking the different PIM views, the user should be able to use the representation that is least error prone." (P1)

However, one participant pointed out that managing all the visual elements of PIM, such as where they are, when they are on or off, and what they represent, causes cognitive fatigue and make the user more prone to making mistakes.

"I think the model added with these UI elements can easily be crowded and confusing. And working with complex model, my guess is that create, cause cognitive fatigue. If there are so many things that you need to manage at the same time. " (P2)

My analysis: These elements, especially the ones that are located in the model window, all appear by request. So a user can simply not use any of them and treat PIM as a simple parametric CAD system with the addition of liveness. These additional UI elements can be used gradually and by discretion when they offer benefits to the user.

Closeness of Mapping

Closeness of Representation to Domain. How closely related is the notation (the programming world) to the domain it is describing (the problem world)?

"What is the domain notation here?" The question came up while discussing closeness of mapping. One of the participants answered:

"For CAD, the domain is objects in the world. We say we take the domain as the model and the other notations as the notations. The more accurate view would be to take each

notation pair as a notation/domain. And there are 3 or 6 such notation pairs. How close is scripting to the model, scripting to dataflow, model to dataflow, functions to each of those." (P1)

> **My analysis:** If we take the model as *the problem world* and other notations as *the machine world*, then PIM brings each of the notations closer to the domain notation by presenting them side-by-side and connecting them through liveness and highlighting. When users see a coordinate system highlighted in the model when they are working on the syntax of a coordinate system in the script, it helps them make associations between the abstract code and the concrete model.
>
> Another aspect of closeness of mapping is the *distance* users feel between their mental model of what they want to do and how they do it using the system. For example, when they want to bring two points closer in the model, they may need to change the T parameter on one of them from 0.2 to 0.85. This operation is far from the intention of bringing the points closer. This distance cannot be eliminated due to the nature of operations in parametric modeling, but PIM allows the user to directly manipulate the objects in the model instead of being limited to the script, therefore reducing the distance to some extent.

An interesting point was made by a participant, stating that with scale, PIM may have to use additional techniques (windows, toolbars, etc.) to handle *PIM features.* For example, when turning on the dependency links in a large model, PIM may have to have a separate window just for the links. So scale may force PIM to drift away from the close mapping that it has with the domain.

"I was just thinking that, ... taking for example the dependency links, if you have a large model, you may want special interface to control the link. Something a lot more complicated that the spider. And I don't know what that might be. But the overall point is that with scale, the PIM interface will probably start having representations that are specific to handling PIM or the relationship between windows, which have nothing to do with the domain itself, (the domain meaning modeling). This is about closeness of mapping. So scale may force PIM to drift away from the close mapping to the domain itself." (P3)

Consistency

Similar semantics are expressed in similar syntactic forms.

An issue with consistency in PIM was said to be that all graphical elements map to code elements, but not all code elements map to graphical objects, such as functions that do not appear in the model or graph. This is an inconsistency in representation.

"All the graphical elements map to code elements. No exceptions. But code elements, functions, do not map into graph nodes or model elements." (P6)

A participant noted that parametric CAD is highly inconsistent. PIM aims to make it more consistent by situating elements of the code in the model, elements of the dataflow in the model, and ultimately elements of the model in the dataflow, through its extreme liveness.

Diffuseness

Verbosity of language. How many symbols or graphic entities are required to express a meaning?

Participants generally agreed that PIM does not alter the diffuseness of parametric CAD systems, especially because the graphical elements appear only on demand and disappear soon after.

"PIM would be appalling if all of its components were always visible. Controlling when elements appear is a crucial aspect of the interface. It would be very verbose if we saw PIM always on at max. So the way in which that is controlled in the interface is probably the single most difficult thing to get right, knowing when I want a dialog box and where and keeping it from being too Microsofty and annoying." (P1)

> **My analysis:** Diffuseness of PIM is a side-effect of improving other dimensions, such as visibility, role-expressiveness, closeness of mapping, and hidden dependencies. Whether this design decision is the right one depends on PIM's success in achieving its goals.

Hard Mental Operations

High demand on cognitive resources. A notation can make things complex or difficult to work out in your head, by making inordinate demands on working memory, or requiring deeply nested goal structures.

PIM was said to reduce hard mental operation by visually connecting the code with the model and not requiring the users to do that in their head. However, managing all the elements of PIM and connecting them together was deemed to be difficult for the users.

A participant asked what PIM offers to the expert user of the system. Another participant made the observation that it reduces hard mental operation caused by names in parametric CAD.

"Parametric modeling inherently relies on names. Watching an expert try to find names is torture in most systems. They can't do it in the spaghetti code. They can sort of do it by search. They can't get it in the model. They have to remember it in the script. But what they really want to do is point to it and get it anywhere they can get it. And so there is one place where linking between representation is good." (P1)

> **My analysis:** As mentioned in error-proneness, PIM's cluttered interface was deemed to be a source of confusion, hard-mental operation, and errors by one of the experts in the focus group. I think that this is not much of an issue because users can choose what to see in the model window at any time and those elements disappear immediately after confirming the task. In addition, I believe that it takes some time for users to get used to the interface and be able to read it easily. It remains to be seen whether a prolonged experience with PIM features makes a difference in its error-proneness and hard mental operation.

> **My analysis:** Preview in PIM reduces the need to follow long chains of dependencies, looking for potential side-effects, by highlighting anything that will be affected by a particular object, therefore reducing hard mental operations common in parametric systems.

Provisionality

Degree of commitment to actions or marks. Is it possible to make provisional actions: recording potential design options, sketching, or playing what-if games.

The general reaction to PIM's preview feature was that it increases provisionality by allowing the user to see a preview of the model before confirming the change, but it is not completely provisional, as it does not save the preview state and only allows the user to work on one instance of the model at a time. So to get to other iterations, undo-redo operations are needed.

"Preview only works with one instance of the model. If you want to see more than one instances, you have to go through sequences of preview-undo commands, in order to create 10 different versions of your model. But at any point in time, you can only see one model. In this case I think provisionality is perhaps low." (P6)

> **My analysis:** Another goal of preview in PIM was to allow the user to explore what-if scenarios quickly and safely without leaving the current state of the model behind. I expected this to increase the provisionality of the system. But a true provisional system should also allow the user to save the alternate state and turn it into an alternative design. This observation takes us to the design exploration domain and introduces several new research problems, such as how to visualize and interact with the alternative state and how to save the alternate state of the dependency structure and visualize it in the graph. I leave this trail to my colleagues in the computational design group, who work on alternative exploration research.

Progressive Evaluation

Work-to-date can be checked at any time. Can a partially-complete program be executed to obtain feedback on the question "How am I doing"?

PIM's immediate feedback to scripting actions received high praise for providing opportunity for progressive evaluation.

"You see in the script your progress, even before your operation is complete. That can actually be a kind of a peripheral feedback, just to know that things are working. I myself, in pretty much everything I do, seem to need a lot of feedback. I look at the rendered version of what I am doing every 5 seconds." (P3)

The fact that PIM presents a preview of the object as soon as enough input is provided and before confirming the operation makes it possible for the user to evaluate the action immediately.

> **My analysis:** Progressive evaluation is a goal that we aimed for by introducing immediate feedback in the model for scripting actions, rendering

objects in the model as soon as enough inputs are provided. Functions or blocks of code get the most benefit from this feature by allowing the user to evaluate their work step-by-step within the function and recovering from errors before moving on to the next step.

6.5 Discussion

As mentioned before, the cognitive dimensions are to be used at the discretion of the designer and with the considerations for the type of tasks and users the system is being designed for. As a parametric CAD system, PIM inherits shortcomings of such systems in several of the dimensions. My goal was to improve the following dimensions in PIM.

- Visibility: *Ability to view components easily.*
- Premature Commitment: *Constraints on the order of doing things.*
- Hidden Dependencies: *Important links between entities are not visible.*
- Role-Expressiveness: *The purpose of an entity is readily inferred.*
- Consistency: *Similar semantics are expressed in similar syntactic forms.*
- Provisionality: *Degree of commitment to actions or marks.*
- Progressive Evaluation: *Work-to-date can be checked at any time.*

In concept, PIM appears to improve the visibility (juxtaposibility) and role-expressiveness of parametric CAD systems. The side-by-side, live windows with brushing and highlighting allows for more information to be visible to the user at any time. It also helps reveal the role of each entity ("what this bit it for"), by connecting it (for example, a piece of code) to its other representations (the node(s) in the graph and geometric objects in the model it represents). However, there was valid concern among the participants about PIM's visibility and role-expressiveness in terms of scalability, for example, in large, 3D models with several hundred lines of code. This is something that needs to be evaluated with a more powerful prototype, capable of handling such large models and 3D geometry.

The hidden dependency dimension is important in parametric CAD, because of the fact that in a parametric model, all components are related to each other and their state is dependant on others' state. So a change in one object may impact several other objects in the model. The graph view reveals these dependencies with a node-link representation of the model and its data structure. Adding scripting to the mix, it becomes a tedious task to find how objects are related to each other. The combination of brushing and highlighting and localized dependency representation in the model view, reduces hidden dependencies to some extent. In addition, the preview feature in PIM reveals side-effects, a problem that results from hidden dependencies in parametric CAD. Again, scalability of these features were questioned in the focus group.

The live scripting interface in PIM allows the user to see and evaluate the model after each small action in the script, hence improving the progressive evaluation dimension. This feature, along with the preview feature, reduce the need for premature commitment.

Situating elements of the code in the model and maintaining direct manipulation of the model during scripting improves the consistency of the system. However, the inconsistency between the programming language and the model remains unaffected by PIM features.

I was not successful in improving the provisionality of parametric CAD modeling with the preview feature. Preview needs additional features such as saving the alternate state to contribute to the provisionality of PIM.

6.6 Summary of chapter

In this chapter, I explained the procedure and results of the focus group analysis of PIM features, with the lens of cognitive dimensions of notations. During the session, I refrained from expressing my opinion and analysis in order to reduce confirmation bias. But in the results section of this chapter, I included my two cents, separate from the results of the focus group. In the next chapter, I will discuss the results of both studies and present my conclusions.

Chapter 7

Discussion and conclusions

In this section, I discuss my findings about PIM features and their implications for the computer-aided design field. I also present the lessons I learned from this project and my conclusions.

7.1 Discussion

7.1.1 Research Questions

I started this research with a few questions that in my opinion, needed to be answered. I found partial answers to these questions in the literature, and the rest I investigated by designing, creating, and evaluating a prototype system. In this section I restate my research questions and discuss how I tried to answer them.

- *RQ 1:* What makes scripting/programming a difficult task? How can we reduce this difficulty in CAD systems?
  The difficulty of scripting in CAD arises mostly from the difficulties of programming in general. Programming requires algorithmic thinking, as well as knowing how to translate instructions to the machine language. It takes away the ability to directly manipulate the model in CAD and requires the user to use the programming notation to express his/her intent (See section 2.2).

  I implemented a few features in the prototype that I hypothesize would reduce this difficulty. The multi-directional liveness of PIM allows the users to directly manipulate the model and supports them in learning the new programming notation by translating their actions to the scripting language.

- *RQ 2:* What characteristics in a CAD scripting environment help reduce designers reluctance to use scripting?
  The attention investment model presents the perceived cost of a programming task as an important factor in users' decision to program (See section 2.3.2). I hypothesize that keeping the script in a "black box", separate from the modeling window, contributes to the perceived difficulty of scripting, due to the fear of unknown that it causes in designers. The live interface of PIM brings scripting to the surface and makes it a part of the interface, equal to the model and graph windows. Having the script update automatically also gives the designers a chance to gradually familiarize themselves with the syntax during modeling tasks. I received feedback form the participants in the user study to the same effect, which is an indication that this is a good starting point to answer this research question. Comprehensive, long term, ethnographical research is needed to prove this hypothesis.

- *RQ 3:* What are the barriers designers face when they program in CAD? How can we reduce these barriers?
  The programming and end-user programming literature provide us with rich descriptions of barriers that both groups of users face and guidelines for reducing those barriers (See section 2.3.1). We hypothesize that PIM has the potential to reduce selection, use, and coordination barriers by maintaining access to the GUI tools during scripting, as well as understanding and information barriers by providing immediate feedback in the model and encouraging incremental running and testing of the program (See section 3.2.2). Evidence of reduced barriers is hard to observe in a short, tutored study session. A long term, carefully designed study is needed for proving this claim.

- *RQ 4:* How can we enable designers to transfer and use their skills and expertise in modeling to scripting?
  Designers can continue to model in PIM, using their existing knowledge, when they need to write a script. This feature does not replace scripting altogether, only extends the transition period from one notation to another and replaces the big jump in the learning curve with several smaller steps.

7.1.2 System features

In this section, I discuss 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 code and geometry. Participants in the user 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, I feel the first step in breaking the barriers to scripting in CAD, is to include *liveness* features such as those implemented in PIM.

Liveness requires multiple representations of the data to be visible simultaneously, resulting in increased visibility and juxtaposibility of the interface. The model to code liveness can be confused with macros (as was by a focus group participant), but there is a major difference. When a novice creates a macro, the goal is not to learn programming. It is to automate a repetitive task. If the user tries to edit the macro code, (s)he faces a program in an unfamiliar language with almost no legible mapping between the syntax and the domain (for example the Visual Basic for Applications code and the Solidworks model). PIM's code to model liveness not only generates the code from user's modeling action (as done in macros), but also presents this translation from one notation to another to the user immediately. This feature supports those users who want to learn the scripting language, by gradually offering them the equivalent of their modeling actions in the programming language. I also note that, as one user study 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.

A live interface, especially one with extreme multi-dimensional liveness, must be experienced first-hand in order for its effects to be fully understood. I believe that to be the reason why user study participants expressed more appreciation for PIM's liveness than focus group participants. They got hands-on experience with the PIM prototype and worked with the live scripting interface, while participants in the focus group only watched a fifteen second demo of the interface and did not get to experience it.

The next logical decision in a live interface is to implement brushing and highlighting across all of the representations. Currently in parametric CAD systems with a node-link graph window, brushing is often used to connect the model with the graph and find objects from one in the other. Adding the script window to the mix is a positive step towards increasing the visibility of the system and role-expressiveness of its entities. Any element anywhere in the interface can be easily mapped to a known representation of it in another window, revealing its role. It helps novice end-user programmers with questions such as *"What does this line of code mean? What does it do?"*, when they attempt to comprehend a script. The other side of the equation is that when an element is known in one notation, it can be found in the other windows, using brushing and highlighting.

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. My 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 user study. Failing to follow dependency links in the model and to 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 I 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.

7.1.3 Scalability

Another challenge was related to scale. The problem has three sides. The first is performance: can our machines handle live coding with complex 3D models? The answer is irrelevant. They may or may not be able to handle it now, but soon they will be, as they are getting better everyday. But 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 3D 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” without losing visibility. I was aware of this challenge when I started this project and I am constantly looking for better and more scalable ways to represent data in PIM. What I 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.

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. But these are only early and partial solutions: complexity will be a challenge for PIM as it is for all visual programming systems.

On the positive side, brushing and highlighting as a search and navigation tool helps with scalability. Finding the code(s) that creates or edits a model object in a 1000-line script is much easier done by hovering the mouse over it in the model and having the system find it in the script. The same goes for the graph. Preview as a watch for effects (and side-effects) also helps the user notice changes in a large model with complex relationships and avoid missing side-effects.

7.2 Lessons learned

Other than the findings of the research, there are lessons to be learned from this research process. These may seem obvious to an experienced researcher, but are valuable to me and my future research projects.

7.2.1 Design

I recently designed a visual interface for a simulation software, as part of a separate project and in collaboration with Autodesk Research. I applied the same methodology, identifying issues and looking for guidelines, but the design process was different from PIM's design. In fact, I believe that it was much improved. In this project, I designed the ideal interface, based on the goals and guidelines we had set for ourselves, without worrying about implementation. This does not mean that I was careless and unrealistic about what can or cannot be done. As a matter of fact, there were two experienced programmers in the team that constantly challenged me and my unrealistic ideas. However, my designs were solely based on the requirements and guidelines and not limited by my own abilities as a programmer. The project is still in progress and I am sure my design will change when it comes time for implementation. But overall, I was confident that I had designed the best solutions possible for the problems we had identified.

In designing PIM features, I feel that I limited my imagination to what we were able to implement within the timeline of the project and the collective expertise of the development team. If I were to do this project again, I would postpone implementation, and design to the best of my "design" knowledge and abilities. Ironically, even the features that I designed conservatively, sound unrealistic and inapplicable to some people.

7.2.2 Implementation

Although I am proud of the system I created with the help of my co-developers, and in spite of the opportunity it gave me to show how a fully live interface would be like, I acknowledge its shortcomings for anything more than demonstrational purposes. The prototype took shape gradually, with me starting it from scratch, and then other people picking it up and adding features to it. There was no system architecture design at the beginning, leaving us shorthanded at the end. I was not concerned, nor equipped to deal with optimization. And the initial choice of the programming language (Processing) was certainly not the right choice for what the system turned out to be at the end. However, these decisions were the best I was able to make at the time and I do not think I could have done it any better, given my knowledge and expertise in computer science. For my next projects, I will approach system implementation more carefully and plan ahead.

7.2.3 Evaluation

I presented PIM features to a few people inside my research group and to independent designers and researchers and received feedback and critics. However, the two formal studies at the end of the project were the main methods of evaluation of PIM features. I was adamant to hold off on the studies until I had a prototype that was ready to be used by the participants. This decision delayed the studies until the very end, after the implementation was done. Looking back at the project, I now think that I could have designed smaller studies earlier to demonstrate PIM features and adjust my design and implementation based on the feedback from those studies. Now that the implementation has stopped and I am at the end of my Ph.D. work, there is regretfully, very limited opportunity for improvement to the PIM features and the prototype.

7.3 Conclusions

7.3.1 Summary

This research was motivated by my interest in end-user programming, especially in computer-aided design systems. Skeptics may argue that there is no need for programming in CAD systems, and in fact, design cannot be done by scripting. I was on that side of the battlefield myself, until I was introduced to parametric CAD. These systems require modeling to be done through defining constraints and relationships, which will sooner or later, require some degree of programming. Not only is it possible to design in parametric CAD systems, but also it gives the designer more freedom to explore and design complex forms and geometries. And the wave of new architects using parametric CAD systems indicates that end-user programming in CAD is here to stay.

The problem of programming in CAD is the unfamiliarity of CAD users with programming. These end-users program not for its own sake, but only as a means for completing a task. The programming notation, textual or visual, is different from the modeling notation and programming concepts are mostly unknown to them. There is a general hesitation, and even fear, among designers to start scripting, which prevents then from taking advantage of the capabilities that scripting brings to their work. This problem was the focus of my work.

I started this research by looking for ways to make a close connection between scripting and modeling in CAD. I tested my initial ideas by presenting them to designers and realized that I was on the right track. More ideation, implementation and evaluation followed, until I felt ready to conduct formal studies with the PIM prototype. The qualitative results showed us how each feature was received and experienced by the participants. Many questions were raised that need further research, and many issues were discovered that require redesign.

7.3.2 Closing remarks

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. My 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, I am 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.

If this project is to move forward, the next step should be redesigning some of the features such as dependency links and implementing 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.

Bibliography

Aish, R. (2003). Extensible computational design tools for exploratory architecture. In *Architecture in the digital age: Design and manufacturing*. New York, NY: Spon Press. 1, 157

Aish, R. (2012). DesignScript: Origins, explanation, illustration. In C. Gengnagel, A. Kilian, N. Palz, & F. Scheurer (Eds.), *Computational design modelling* (pp. 1–8). Springer Berlin Heidelberg. 24

Aish, R. & Woodbury, R. F. (2005). Multi-level interaction in parametric design. In Butz, A., Fisher, B., Krüger, A., & Oliver, P. (Eds.), *SmartGraphics, 5th Intl. Symp., SG2005*, LNCS 3638, (pp. 151–162)., Germany. Springer. 5

Autodesk (2013). Dynamo visual programming for designers. Accessed on 11/04/2013 at http://autodeskvasari.com/dynamo. 20

Baroth, E. & Hartsough, C. (1995). Visual programming in the real world. In *Visual object-oriented programming*, (pp. 21–42). 20

Bartram, L. (2009). IAT 814: Knowledge, visualization, and communication. Graduate course at School of Interactive Arts and Technology, Simon Fraser University. 176

Blackwell, A. (2002). First steps in programming: A rationale for attention investment models. In *Proceedings IEEE 2002 symposia on human centric computing languages and environments*, (pp. 2–10)., Arlington, VA, USA. 9, 10, 16, 36, 40, 85

Blackwell, A. & Burnett, M. M. (2002). Applying attention investment to end-user programming. In *Proceedings IEEE 2002 symposia on human centric computing languages and environments*, (pp. 28–30). 17, 27, 30

Blackwell, A. & Collins, N. (2005). The programming language as a musical instrument. *Proceedings of PPIG05 (psychology of programming interest group)*. 24

Blackwell, A. F., Britton, C., Cox, A., Green, T. R. G., Gurr, C., Kadoda, G., Kutar, M. S., Loomes, M., & Nehaniv, C. L. (2001). Cognitive dimensions of notations: Design tools for cognitive technology. In *Cognitive technology: Instruments of mind*, volume 2117. Berlin, Heidelberg: Springer Berlin Heidelberg. 13, 103

Blackwell, A. F. & Green, T. R. G. (2000). A cognitive dimensions questionnaire optimised for users. In *12th workshop of the psychology of programming interest group*. 75

Boren, T. & Ramey, J. (2000). Thinking aloud: Reconciling theory and practice. *IEEE transactions on professional communication*, *43*(3), 261–278. 72

Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A., & Miller, P. (1997). Mini-languages: A way to learn programming principles. *Education and information technologies*, *2*(1), 65–83. 29

Burg, B., Kuhn, A., & Parnin, C. (2013). Live programming workshop. Accessed on 11/04/2013 at http://liveprogramming.github.io/2013/. 24

Burnett, M. M. (1999). Visual programming. *Wiley Encyclopedia of Electrical and Electronics Engineering*. 19

Burnett, M. M. (2009). What is end-user software engineering and why does it matter. Siegen, Germany. Heidelberg : Springer. 30

Burnett, M. M., Atwood, J. W., Djang, R. W., Reichwein, J., Gottfried, H. J., & Yang, S. (2001). Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of functional programming*, *11*(2), 155–206. 22, 27

Burnett, M. M., Atwood Jr, J. W., & Welch, Z. T. (1998). Implementing level 4 liveness in declarative visual programming languages. In *1998 IEEE symposium on visual languages*, (pp. 126–133). IEEE. 24, 25

Burnett, M. M. & Baker, M. J. (1994). A classification system for visual programming languages. *Journal of Visual Languages & Computing*, *5*(3), 287–300. 19

Burnett, M. M., Baker, M. J., Bohus, C., Carlson, P., Yang, S., & Zee, P. V. (1995). Scaling up visual programming languages. *Computer*, *28*(3), 45–54. 19

Burnett, M. M., Cook, C., & Rothermel, G. (2004). End-user software engineering. *Commun. ACM*, *47*(9), 53–58. 30

Burnett, M. M., Yang, S., & Summet, J. (2002). A scalable method for deductive generalization in the spreadsheet paradigm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, *9*(4), 253–284. 27

Burry, M. (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*, Taiwan. 5, 156

Charmaz, K. (2006). *Constructing grounded theory: A practical guide through qualitative analysis*. Pine Forge Press. 81

Cooper, A., Reimann, R., Cronin, D., & Cooper, A. (2007). *About Face3: The essentials of interaction design.* Indianapolis, IN: John Wiley & Sons. 77

Costagolia, G., Delucia, A., Orefice, S., & Polese, G. (2002). A classification framework to support the design of visual languages. *Journal of visual languages & computing*, *13*(6), 573–600. 19

Creswell, J. W. & Creswell, J. W. (2007). *Qualitative inquiry & research design: Choosing among five approaches* (2nd ed.). Thousand Oaks: SAGE Publications. 81

Creswell, J. W. & Miller, D. L. (2000). Determining validity in qualitative inquiry. *Theory into practice*, *39*(3), 124–130. 79

Cypher, E. & Halbert, D. C. (1993). *Watch what I do: Programming by demonstration.* MIT Press. 22

Davies, D. & Dodd, J. (2002). Qualitative research and the question of rigor. *Qualitative health research*, *12*(2), 279–289. 78

Davis, A. & Keller, R. (1982). Data flow program graphs. *Computer*, *15*(2), 26–41. 20

Davis, D. (2011). Yeti. Accessed on 31/01/2014 at http://www.yeti3d.com/. 24

Denzin, N. K. (1970). *Sociological methods: A sourcebook.* Butterworths London. 79, 101

Dertouzos, M. (1992). ISAT summer study: Gentle slope systems; making computers easier to use. Presented at Woods Hole, MA. 18, 37

Ericsson, K. A. & Simon, H. A. (1993). *Protocol analysis verbal reports as data* (Rev. ed.). MIT cognet. Cambridge, Mass: MIT Press. 72

Experimental Media Research Group, St-Lucas Art College, A. B. (2013). Nodebox. Accessed on 11/04/2013 at http://www.nodebox.net. 24

Florit, G. (2013). Live coding. Accessed on 2/12/2013 at http://livecoding.io/. 25

Gantt, M. & Nardi, B. A. Gardeners and gurus: Patterns of cooperation among CAD users. In *Proceedings of the SIGCHI conference on Human factors in computing systems.* 5, 14

Golafshani, N. (2003). Understanding reliability and validity in qualitative research. *The qualitative report*, *8*(4), 597–607. 78

Green, T. R. G. (1989). Cognitive dimensions of notations. In *People and computers V*, (pp. 443,460). University Press. 11, 23

Green, T. R. G. (2000). Instructions and descriptions: Some cognitive aspects of programming and similar activities. In *Proceedings of the working conference on advanced visual interfaces*, (pp. 21–28)., Palermo, Italy. ACM. 11, 13, 85, 98, 99

Green, T. R. G., Blandford, A. E., Church, L., Roast, C. R., & Clarke, S. (2006). Cognitive dimensions: Achievements, new directions, and open questions. *Journal of visual languages & computing*, *17*(4), 328–365. 103

Green, T. R. G. & Petre, M. (1996). Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of visual languages & computing*, *7*(2), 131–174. 158

Hix, D. (1993). *Developing user interfaces : Ensuring usability through product and process*. New York: John Wiley & Sons. 23

Hoc, J. M. & Nguyen Xuan, A. (1990). Language semantics, mental models and analogy. *Psychology of programming*, *10*, 139–156. 23

Hoffmann, C. M. & Joan-Arinyo, R. (2005). A brief on constraint solving. *Computer-aided design & applications*, *2*(5), 655–663. 5

Hundhausen, C., Farley, S., & Brown, J. (2006). Can direct manipulation lower the barriers to programming and promote positive transfer to textual programming? an experimental study. In *Visual languages and human-centric computing, 2006. vl/hcc 2006. IEEE symposium on*, (pp. 157–164). 29

Hutchins, E. L., Hollan, J. D., & Norman, D. A. (1985). Direct manipulation interfaces. *human-computer interaction*, *1*(4), 311. 9, 10, 36

Jabi, W. (2012). *Parametric design in architecture*. Laurence King Publishing. 2

Johnston, W. M., Hanna, J. R. P., & Millar, R. J. (2004). Advances in dataflow programming languages. *ACM computing surveys (CSUR)*, *36*, 1–4. ACM ID: 1013209. 20

Jones, C. (1995). End user programming. *Computer*, *28*(9), 68–70. 2, 14

Jones, S. P., Blackwell, A., & Burnett, M. M. (2003). A user-centred approach to functions in excel. In *Proceedings of the eighth ACM SIGPLAN international conference on functional programming*, (pp. 165–176)., Uppsala, Sweden. ACM. 26, 27

Kelleher, C. & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM computing survey*, *37*(2), 83–137. 28, 158

KhanAcademy (2012). Khan academy computer science. Accessed on 11/04/2013 at http://www.khanacademy.org/cs/new. 24

Kirk, J. & Miller, M. L. (1988). Reliability and validity in qualitative research. *International journal of qualitative studies in education*, *1*(1). 78

Ko, A. J. (2007). Barriers to successful end-user programming. In *End-User Software Engineering*, Dagstuhl, Germany. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. 15, 16

Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M. M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B. A., Rosson, M. B., Rothermel, G., Shaw, M., & Wiedenbeck, S. (2011). The state of the art in end-user software engineering. *ACM computing survey*, *43*(3), 21:121:44. 14, 30

Ko, A. J. & Myers, B. A. (2005a). A framework and methodology for studying the causes of software errors in programming systems. *Journal of visual languages & computing*, *16*(12), 41–84. 16

Ko, A. J. & Myers, B. A. (2005b). Human factors affecting dependability in end-user programming. In *Proceedings of the first workshop on end-user software engineering*, (pp. 1–4)., St. Louis, Missouri. ACM. 16

Ko, A. J., Myers, B. A., & Aung, H. H. (2004). Six learning barriers in end-user programming systems. In *Proceedings of IEEE symposium on visual languages and human centric computing*, (pp. 199–206). 15, 38, 40, 72, 85

Kosinski, P. R. (1973). A data flow language for operating systems programming. In *ACM SIGPLAN notices*, volume 8, (pp. 89–94). 20

Lauria, S., Bugmann, G., Kyriacou, T., & Klein, E. (2002). Mobile robot programming using natural language. *Robotics and autonomous systems*, *38*(3), 171–181. 23

Lieberman, H. & Fry, C. (1995). Bridging the gulf between code and behavior in programming. In *Proceedings of the SIGCHI conference on human factors in computing systems*, CHI '95, (pp. 480–486)., New York, NY, USA. ACM Press/Addison-Wesley Publishing Co. 2, 8, 10

Lieberman, H., Paterno, F., Klann, M., & Wulf, V. (2006). End-user development: An emerging paradigm. In *End user development*, volume 9 (pp. 1–8). Dordrecht: Springer Netherlands. 17

Lincoln, Y. S. (1985). *Naturalistic inquiry*, volume 75. SAGE Publications. 78

Loui, R. (2008). In praise of scripting: Real programming pragmatism. *Computer*, *41*(7), 22–26. 1

MacLean, A., Carter, K., Lvstrand, L., & Moran, T. (1990). User-tailorable systems: Pressing the issues with buttons. In *Proceedings of the SIGCHI conference on human factors in computing systems: Empowering people*, CHI '90, (pp. 175182)., New York, NY, USA. ACM. xiv, 17

Mathison, S. (1988). Why triangulate? *Educational researcher*, *17*(2), 13–17. 79

Maxwell, J. A. (1992). Understanding and validity in qualitative research. *Harvard educational review*, *62*(3), 279–301. 79

McIver, L. & Conway, D. (1996). Seven deadly sins of introductory programming language design. In *Proceedings of the 1996 international conference on software engineering: Education and practice (se:ep '96)*, (pp. 309–316). IEEE Computer Society. 28

McNeel, R. (2010). Grasshopper: Generative modeling with rhino. 20

Miles, M. B. & Huberman, A. M. (1994). *Qualitative data analysis: An expanded sourcebook*. Thousand Oaks: SAGE Publications. 81

Miller, L. A. (1981). Natural language programming: Styles, strategies, and contrasts. *IBM systems journal*, *20*(2), 184–215. 23

Morgan, D. L. (1998). *The focus group guidebook*. Thousand Oaks, California, USA: SAGE Publications. 98

Morgan, D. L., Krueger, R. A., & King, J. A. (1998). *Analyzing and reporting focus group results*. Thousand Oaks, Calif.: SAGE Publications. 102

Morin, R. & Brown, V. (1999). Scripting languages. *The journal of Apple technology (MacTech)*, *15*(9). 1

Myers, B. A. (1992). Demonstrational interfaces: A step beyond direct manipulation. *Computer*, *25*(8), 61–73. 22

Myers, B. A., Burnett, M. M., Wiedenbeck, S., & Ko, A. J. (2007). End user software engineering: CHI 2007 special interest group meeting. In *CHI'07 extended abstracts on human factors in computing systems*, (pp. 2125–2128). ACM. 2, 14

Myers, B. A., Burnett, M. M., Wiedenbeck, S., Ko, A. J., & Rosson, M. B. (2009). End user software engineering: CHI 2009 special interest group meeting. In *CHI'09 extended abstracts on human factors in computing systems*, (pp. 2731–2734)., Boston, MA, USA. ACM. 30

Myers, B. A., Ko, A. J., & Burnett, M. M. (2006). Invited research overview: End-user programming. In *CHI '06 extended abstracts on human factors in computing systems*, (pp. 75–80)., Montreal, Quebec, Canada. ACM. 14

Myers, B. A., Pane, J. F., & Ko, A. J. (2004). Natural programming languages and environments. *Communications of the ACM*, *47*(9), 47–52. 23

Myers, B. A., Smith, D. C., & Horn, B. (1992). Report of the end-user programming working group. In *Languages for developing user interfaces* (pp. 343–366). Natick, MA, USA: A. K. Peters, Ltd. xiv, 18, 37

Nardi, B. A. (1993). *A small matter of programming: Perspectives on end user computing.* Cambridge, MA: MIT Press. 2, 14

Nardi, B. A. & Miller, J. R. (1991). Twinkling lights and nested loops: Distributed problem solving and spreadsheet development. *International Journal of Man-Machine Studies*, *34*(2), 161184. 15

Nielsen, J. (1994). Heuristic evaluation. *Usability inspection methods*, *17*, 25–62. 23, 28

Pane, J. F. & Myers, B. A. (1996). Usability issues in the design of novice programming systems. 28, 158

Pane, J. F. & Myers, B. A. (2006). More natural programming languages and environments. In *End user development* (pp. 31–50). Springer. 23

Pane, J. F., Myers, B. A., & Miller, L. B. (2002). Using HCI techniques to design a more usable programming system. In *Human centric computing languages and environments, 2002. proceedings. IEEE 2002 symposia on*, (pp. 198–206). 2, 8

Patton, M. Q. (1990). Qualitative evaluation and research methods (2nd ed.), 532. 79

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., & Paterson, J. (2007). A survey of literature on the teaching of introductory programming. In *ACM SIGCSE bulletin*, volume 39, (pp. 204–223). ACM. 28

Peters, B. (2007). The smithsonian courtyard enclosure: A case-study of digital design processes. In *Proceedings of the 27th annual conference of the association for computer aided design in architecture*, Halifax (Nova Scotia). xvi, 157

Peters, B. & Peters, T. (2013). *Inside SmartGeometry.* John Wiley & Sons. 2

Petre, M. (1995). Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, *38*(6), 33–44. 19

Rogalski, J. & Samurcay, R. (1990). Acquisition of programming knowledge and skill. In *Psychology of programming*, Computers and People, (pp. 157). 8

Rubin, M. J. (2013). The effectiveness of live-coding to teach introductory programming. In *Proceeding of the 44th ACM technical symposium on computer science education*, (pp. 651–656). ACM. 24

Scaffidi, C., Shaw, M., & Myers, B. A. (2005). Estimating the numbers of end users and end user programmers. In *Visual languages and human-centric computing, 2005 IEEE symposium on*, (pp. 207–214). 14

Schmidt, K. & Wagner, I. (2004). Ordering systems: Coordinative practices and artifacts in architectural design and planning. *Computer Supported Cooperative Work (CSCW)*, *13*(5-6), 349408. 5

Seale, C. (1999). Quality in qualitative research. *Qualitative inquiry*, *5*(4), 465–478. 78

Senske, N. (2005). Fear of code: An approach to integrating computation with architectural design. Thesis, Massachusetts Institute of Technology. 40

Shneiderman, B. (1983). Direct manipulation: A step beyond programming languages. *Computer*, *16*(8), 57–69. 9, 23, 36

Spradley, J. P. (1980). *Participant observation.* New York: Holt, Rinehart and Winston. 71

Stenbacka, C. (2001). Qualitative research requires quality concepts of its own. *Management decision*, *39*(7), 551–556. 78

Stewart, D. W. (2007). *Focus groups: Theory and practice.* SAGE Publications. 98

Streich, B. (1992). Should we integrate programming knowledge into the architect's CAAD education? In *CAAD instruction: The new teaching of an architect?* 156

Tanimoto, S. L. (1990). VIVA: A visual language for image processing. *Journal of visual languages & computing*, *1*(2), 127–139. 24

Van-Roy, P. & Haridi, S. (2004). *Concepts, techniques, and models of computer programming.* Cambridge, Mass.: MIT Press. 8

Victor, B. (2012). Learnable programming. Accessed on 11/04/2013 at http://worrydream.com/#!/LearnableProgramming. 25

Ware, C. (2005). *Information visualization: Perception for design* (2nd ed.). San Francisco California USA: Morgan Kaufmann. 176

Whitley, K. N. (1997). Visual programming languages and the empirical evidence for and against. *Journal of visual languages & computing*, *8*(1), 109–142. 19

Woodbury, R. F. (2008). End-user programming. Graduate course at School of Interactive Arts and Technology, Simon Fraser University. 158

Woodbury, R. F. (2010). *Elements of parametric design.* Taylor and Francis. 1, 2, 4, 5, 8, 74

Woodbury, R. F. & Burrow, A. L. (2006). Whither design space? *AIE EDAM: Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, *20*(2), 63–82. 5

Appendix A

User study consent form

Simon Fraser University
Programming in the Model in CAD Systems

Informed Consent Form

Study ID: 2012s0591

Participant ID: ——————

This research is being conducted under the permission of the Simon Fraser Research Ethics Board. The chief concern of the Board is for the health, safety and psychological well-being of participants. Should you wish to obtain information about your rights as a participant in research, the responsibilities of researchers, or have questions, concerns, or complaints, please contact the following people with the study ID 2012s0591.

Primary contact: Robert Woodbury, Academic supervisor, School of Interactive Arts and Technology, Simon Fraser University, Surrey, BC, Canada, V3T 0A3, Tel: ———-, email: ———-.

Secondary contact: Hal Weinberg, Director, Office of Research Ethics, Simon Fraser University, Burnaby, BC, Canada, V5A 1S6, Tel: ———-, email: ———-.

———————————————————————————————————-

- **Title of Study:** Programming in the Model in CAD Systems
- **Researcher:** Maryam Mokhtar Maleki
- **Supervisor:** Robert Woodbury

- **Department:** School of Interactive Arts and Technology, Simon Fraser University

---

Goal of the Study.

Designers and engineers who use CAD may at some point need to write scripts to automate an action or solve complex geometry. This study is part of the principal investigator's Ph.D. thesis on end-user programming in Computer-Aided Design (CAD) systems. In order to improve designers' scripting experience, we present a collection of techniques regarding the programming and modelling environments in CAD. The goal of this study is to evaluate these techniques, using PIM, a prototype CAD system developed for this purpose.

What to expect.

In this study, you will be introduced to the PIM system and asked to perform a few tasks. These tasks are chosen from common CAD modelling and scripting tasks. You will fill in a pre-task questionnaire at the beginning of the session and answer a set of questions about the system after the task. You can choose to answer these question on paper, by typing, or verbally. Then you will participate in a brief conversation with the researchers about this experience. During the session, one or more researchers will observe you. We ask that you think aloud during the task, especially when you face a challenge or notice a problem in the system. You are free to ask questions from the researcher(s) throughout the study. The computer screen will be recorded by a computer program. With your permission, the session will be voice recorded. This is an essential tool in recording and analyzing all aspects of the session, therefore, If you do not wish to be voice recorded, we will not be able to continue the study and will have to terminate the session.

Risks and Benefits.

There is no risk involved in participating in this study. If you feel tired or uncomfortable at any time during the session, you can take a break, or withdraw from the study completely. Upon completion of the study, you will receive a $20 gift card.

Confidentiality.

Your name and identity will be kept confidential and will not be reported in any publications. Your voice will be changed on the recording to hide your identity. Data collected for this study, including voice recording, transcripts, and digital questionnaires will be kept in a password protected folder on the researcher's computer, which is locked in a secure cabinet when not in use, and backed up on SFU SIAT research server, also protected by a password.

After the analysis is complete, the data will be transferred to a secure external hard drive and will be kept in a locked cabinet alongside the hard copies of the questionnaires until year 2015 when they will be destroyed.

Participant's Rights.

Your participation is completely voluntary and has no effect on your coursework and/or your job. You may withdraw from the study at any time, including during the procedure, without any consequence. The researchers may need to contact you in the near future to verify the data collected during the study and/or their interpretation of it. You are free to refuse future contact. You may obtain copies of the results of this study upon its completion by contacting Maryam M. Maleki, email: ———-.

————————————————————————————————-

Your signature on this form will signify that you have received this consent form that describes the procedures and possible risks and benefits of this research study; that you have received an adequate opportunity to consider this information; and that you voluntarily agree to participate in the study.

Voice recording: (Please choose one)

☐ I consent to be voice recorded during the session.

☐ I do not wish to be voice recorded and would like to withdraw from this study.

Future contact: (Please choose one)

☐ I consent to be contacted in the future about this study. Email: ——————————-

☐ I do not wish to be contacted in the future about this study.

Participant First Name: —————————— Last Name: ——————————

Participant Signature: ——————————————- Date: ————————

Appendix B

User study questionnaire and tasks

Pre-task questionnaire

1. Gender: Male ☐ Female ☐
2. Profession (e.g. architecture design, drafting, mechanical engineering, computational design, HCI, interaction design, etc.):

   ____

3. Do you use Computer Aided Design (CAD) as part of your work / studies? ________
   On average, how many hours/week do you work with CAD systems? ________
4. If you do use CAD, please choose one of the following:

   ☐ I work primarily in the graphical user interface, directly interacting with the 2D/3D model and interface buttons.

   ☐ I spend most of my time writing programs or scripts.

   ☐ It's almost 50/50.
5. Do you have any experience with computer programming or scripting? ________
   If no, proceed to question 10.
6. How long have you been programming as part of your work or school projects? ____

7. What type of programmer do you consider yourself to be? ☐ Beginner ☐ Intermediate ☐ Expert

8. What languages have you programmed in? __________________________________-

   ____

9. How many formal programming courses have you taken? ______________________

10. For each of the CAD systems below please indicate a) if you have ever worked with the system, b) how many years of experience you have working with it, and c) if you have written any programs or scripts in the system's scripting environment.

	Experience	Years of experience	Programming/ scripting
Solidworks	☐	________	☐
Grasshopper	☐	________	☐
Generative Components	☐	________	☐
Revit	☐	________	☐
Catia	☐	________	☐
Digital Project	☐	________	☐
Maya	☐	________	☐
Cinema 4D	☐	________	☐
3D Studio Max	☐	________	☐
Rhino 3D	☐	________	☐
AutoCAD	☐	________	☐
ArchiCAD	☐	________	☐
Inventor	☐	________	☐
Form Z	☐	________	☐
Microstation	☐	________	☐
ProEngineer	☐	________	☐

☐ Demo 1

The researcher demonstrates the modeling interface.

☐ Task 1

- Step 1: Create a line by points that connects two of the existing points in the model.
- Step 2: Create two points in the model by Cartesian coordinates so that they always have the same X coordinate.
- Step 3: Edit the line you made in step 1 and choose one of the points from step 2 as its start point.
- Step 4: Select one of the vectors in the model and using its handle, edit it so that it is almost horizontal.

☐ Demo 2

The researcher demonstrates the scripting interface.

☐ Task 2

- Step 1: Identify the left most coordinate system in the model, then highlight it and find its syntax in the script window. Change its Y input to 50 in the script.
- Step 2: Find the only line in the model that is created by origin, vector, and length. Edit this line in the script window and change its vector input.
- Step 3: In the script window, create a line that connects point03 and point05.
- Step 4: In the script window, edit the T input of point05 so that the line you created in step 3 is almost horizontal.

☐ Video 1

Please watch the 4-minute video demo.

☐ Demo 3

The researcher demonstrates more features of the prototype.

☐ Task 3

- Step 1: Create a replicated point (or an array of points or a point series) in the model with three values for each of X and Y inputs. Try different replication types.
- Step 2: In the script, create a vector by origin XY, and use the second from the left, third from the top point in the point array in step 1 as its origin.
- Step 3: Using the dependency links, can you tell what objects will be affected if you edit the vector called vect01, without actually editing it?
- Step 4: Turn the preview on. Now edit vect01, so that vect05 rotates about 90 degrees from its current state.

☐ Video 2

Please watch the 4-minute video demo.

Appendix C

Focus group consent form

Simon Fraser University

Programming in the Model in CAD Systems

Informed Consent Form

Study ID: 2012s0591

Participant ID: ————————

This research is being conducted under the permission of the Simon Fraser Research Ethics Board. The chief concern of the Board is for the health, safety and psychological well-being of participants. Should you wish to obtain information about your rights as a participant in research, the responsibilities of researchers, or have questions, concerns, or complaints, please contact the following people with the study ID 2012s0591.

Primary contact: Robert Woodbury, Academic supervisor, School of Interactive Arts and Technology, Simon Fraser University, Surrey, BC, Canada, V3T 0A3, Tel: ———-, email: ———-.

Secondary contact: Hal Weinberg, Director, Office of Research Ethics, Simon Fraser University, Burnaby, BC, Canada, V5A 1S6, Tel: ———-, email: ———-.

———————————————————————————————-

- **Title of Study:** Programming in the Model in CAD Systems
- **Researcher:** Maryam Mokhtar Maleki
- **Supervisor:** Robert Woodbury

- **Department:** School of Interactive Arts and Technology, Simon Fraser University

---

Goal of the Study.

Designers and engineers who use CAD may at some point need to write scripts to automate an action or solve complex geometry. This study is part of the principal investigator's Ph.D. thesis on end-user programming in Computer-Aided Design (CAD) systems. In order to improve designers' scripting experience, we present a collection of techniques regarding the programming and modelling environments in CAD. The goal of this study is to evaluate these techniques, using PIM, a prototype CAD system developed for this purpose.

What to expect.

In this study, you will participate in a focus group. You will fill in a short questionnaire and then watch a demo of the PIM system. Afterwards, the researcher will ask the group questions related to the cognitive dimensions framework and the PIM system. You are asked to take part in the discussion and express your opinion. With your permission, the session will be voice recorded. This is an essential tool in recording and analyzing all aspects of the session, therefore, If you do not wish to be voice recorded, we will not be able to continue the study and will have to terminate the session.

Risks and Benefits.

There is no risk involved in participating in this study. If you feel tired or uncomfortable at any time during the session, you can take a break, or withdraw from the study completely.

Confidentiality.

You will be in the same room with other participants of this focus group. So your identity will be revealed to them. However, you will not be asked questions about your personal life, but only your feedback and opinion about a computer system. At this point, you can choose to withdraw from the study if you are not comfortable with this arrangement. Your name and identity will be kept confidential after the session and will not be reported in any publications. Your voice will be changed on the recording to hide your identity. Data collected for this study, including voice recording, transcripts, and digital questionnaires will be kept in a password protected folder on the researcher's computer, which is locked in a secure cabinet when not in use, and backed up on SFU SIAT research server, also protected by a password. After the analysis is complete, the data will be transferred to a secure external hard drive and will be kept in a locked cabinet alongside the hard copies of the questionnaires until year 2015 when they will be destroyed.

Participant's Rights.

Your participation is completely voluntary and has no effect on your coursework and/or your job. You may withdraw from the study at any time, including during the procedure, without any consequence. The researchers may need to contact you in the near future to verify the data collected during the study and/or their interpretation of it. You are free to refuse future contact. You may obtain copies of the results of this study upon its completion by contacting Maryam M. Maleki, email: ———-.

————————————————————————————-

Your signature on this form will signify that you have received this consent form that describes the procedures and possible risks and benefits of this research study; that you have received an adequate opportunity to consider this information; and that you voluntarily agree to participate in the study.

Voice recording: (Please choose one)

☐ I consent to be voice recorded during the session.

☐ I do not wish to be voice recorded and would like to withdraw from this study.

Future contact: (Please choose one)

☐ I consent to be contacted in the future about this study. Email: ——————-

☐ I do not wish to be contacted in the future about this study.

Participant First Name: —————————— Last Name: ——————————

Participant Signature: ——————————————- Date: ————————

Appendix D

Focus group questionnaire

Section One

Visibility: *Ability to view components easily.*

The visibility dimension denotes simply whether required material is accessible without cognitive work; whether it is or can readily be made visible, whether it can readily be accessed in order to make it visible, or whether it can readily be identified in order to be accessed. An important component is juxtaposability, the ability to see any two portions of the program on screen side-by-side at the same time.

Role-Expressiveness: *The purpose of an entity is readily inferred.*

How easy is it to answer the question "What is this bit for?" How easy is it to "read" the programming language?

Hidden Dependencies: *Important links between entities are not visible.*

Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic? If one entity cites another entity, which in turn cites a third, changing the value of the third entity may have unexpected repercussions (side effects). How easy is it to spot these side effects?

Additional notes and comments:

Section Two

Viscosity: *Resistance to change*

How much effort is required to perform a single change? There are two types of viscosity: repetition viscosity (many actions of the same type) and knock-on viscosity (where further actions are required to restore consistency).

Premature Commitment: *Constraints on the order of doing things.*

Do programmers have to make decisions before they have the information they need? An example is writing the contents list (with page numbers) before you write the book! The difficulty is that until you see your first efforts, you often have little idea of how to make your initial choices: yet the nature of the system may be such that these choices are forced upon you before you are ready.

Provisionality: *Degree of commitment to actions or marks.*

Is it possible to make provisional actions: recording potential design options, sketching, or playing what-if games.

Additional notes and comments:

Section Three

Error-Proneness: *The notation invites mistakes and the system gives little protection.*

It is conventional to distinguish between "mistakes" and "slips", where a slip is doing something you didn't mean to do, where you knew what to do all along but somehow did it wrong; contrast with those parts of program design and coding that are deeply difficult, where mistakes of problem analysis are quite common.

Progressive Evaluation: *Work-to-date can be checked at any time.*

Can a partially-complete program be executed to obtain feedback on the question "How am I doing"? The opposite type of environment, in contrast, prohibits any testing until the program is completely ready. Only then can it be compiled and run.

Consistency: *Similar semantics are expressed in similar syntactic forms.*

Users often infer the structure of information artifacts from patterns in notation. If similar information is obscured by presenting it in different ways, usability is compromised.

Additional notes and comments:

Section Four

Closeness of Mapping: *Closeness of Representation to Domain.*

How closely related is the notation (the programming world) to the domain it is describing (the problem world)? Closeness of mapping applies to both "entities" and "operations" on those entities.

Diffuseness: *Verbosity of language.*

How many symbols or graphic entities are required to express a meaning? Some notations can be annoyingly long-winded, or occupy too much valuable real estate within a display area.

Hard Mental Operations: *High demand on cognitive resources.*

A notation can make things complex or difficult to work out in your head, by making inordinate demands on working memory, or requiring deeply nested goal structures.

Additional notes and comments:

Appendix E

Cognitive dimensions questionnaire

Please read the instructions and answer the following questions about the system you used in this study. If you need more space, please use the back of the sheet and include the four-letter code of the question with your answer. You can choose to fill this form on the computer at http://websurvey.sfu.ca/survey/120117928. If you have any questions or if you need clarification, please don't hesitate to ask the researcher.

Instructions and definitions

You might need to think carefully to answer the questions in the next sections, so we have provided some definitions to get you started:

The *product* is the ultimate reason why you are using the notational system: what things happen as an end result, or what things will be produced as a result of using the notational system. In this case, the CAD model is the product of the notational system.

The *notation* is how you communicate with the system: you provide information in some special format to describe the end result that you want, and the notation provides information that you can read. Notations have a structure that corresponds in some way to the structure of the product they describe. They also have parts (components, aspects etc.) that correspond in some way to parts of the product. Notations can include text, pictures, diagrams, tables, special symbols or various combinations of these. Some systems include multiple notations. In this case, the model, the script, the graph, and the dialogue boxes are different notations of the system.

Questions

[VIJU]

How easy was it to see or find the various parts of the notation while it was being created or changed? Why?

What kind of things were more difficult to see or find?

If you needed to compare or combine different parts, could you see them at the same time? If not, why not?

[PROG]

How easy was it to stop in the middle of creating some notation, and check your work so far? Could you do this any time you liked? If not, why not?

Were you able to find out how much progress you had made, or check what stage in your work you were up to? If not, why not?

[HMOS]

What kind of things required the most mental effort with this notation?

Did some things seem especially complex or difficult to work out in your head (e.g. when combining several things)? What were they?

[HIDD]

Were dependencies between parts visible? What kind of dependencies were hidden?

[PREM]

When you were working with the notation, could you go about the job in any order you liked, or did the system force you to think ahead and make certain decisions first?

If so, what decisions did you need to make in advance? What sort of problems can this cause in your work?

[CONS]

Where there were different parts of the notation that meant similar things, was the similarity clear from the way they appeared? Please give examples.

Were there places where some things ought to be similar, but the notation made them different? What were they?

[ERRP]

Did some kinds of mistake seem particularly common or easy to make? Which ones?

Did you often find yourself making small slips that irritated you or made you feel stupid? What are some examples?

[CLOS]

How closely related was the notation to the result that you were describing? Why?

Which parts seemed to be a particularly strange way of doing or describing something?

[VISC]

When you needed to make changes to previous work, how easy was it to make the change? Why?

Were there particular changes that were more difficult or especially difficult to make? Which ones?

[DIFF]

Did the notation a) let you say what you wanted reasonably briefly, or b) was it long-winded? Why?

What sorts of things took more space to describe?

[ROLE]

Was it easy to tell what each part is for in the overall scheme? Why?

Were there some parts that were particularly difficult to interpret? Which ones?

[PROV]

Was it possible to sketch things out when you were playing around with ideas, or when you weren't sure which way to proceed? What features of the notation helped you to do this?

What sort of things could you do when you didn't want to be too precise about the exact result you were trying to get?

Discussion

Please have a conversation with the researcher about the following questions.

- Considering the video demo you watched earlier, can you think of obvious ways that the design of the system could be improved? What are they? Could it be improved specifically for your own requirements?
- Would a PIM-like system be useful to you? How? or Why not?

Appendix F

List-Based Programming Tasks in the CAD Model

Abstract

Most Computer-Aided Design (CAD) systems provide programming capabilities for designers to explore and create without being limited to the boundaries of the interface. However, computer programming presents a challenge for non-programmer users of CAD, as in any other end-user programming environment. Architects, mechanical engineers and other CAD users' job usually involve creating, modifying, and representing a 2D or 3D model. Therefore they acquire a great deal of experience with the model and have a good visual and spatial understanding of it. We propose *programming in the model* as a way of using designers' expertise with the model to enhance their programming experience, as well as minimizing the mental shift between computer programming and design. In this study, we chose *list* as a programming element and tried to bring the task of making lists to the model, in a prototype created in Generative Components software. We then conducted an experiment to test and compare the prototype against the existing system with two groups of expert and novice users of the software. Although no statistical significance were found in the results to prove the hypothesis, there are interesting findings that suggest future paths for this research.

Introduction

Most Computer-Aided Design (CAD) systems 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. In such cases, users need to use computer programming to have more capabilities and freedom to explore. However, the shift from design space to programming environments is more challenging than it sounds. We suggest using designers' spatial and visual abilities for programming purposes to overcome this challenge, by visualizing some programming concepts in the model, in order to bring programming space and design space closer together.

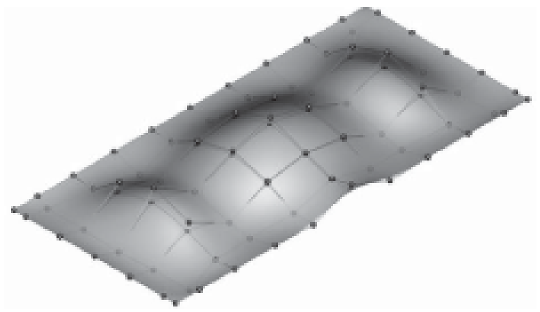
Computer-Aided Design systems

Computer-aided design refers to the use of computer technology in the designing, drafting and manufacturing of a product such as an engine in mechanical engineering, or part or all of a building in architectural design. In the early days, CAD systems were mainly used to represent the final outcome of the design. Later, faster computers and 3D modeling took CAD to the next level. Nowadays, parametric modeling has made it possible to change the inputs and produce different results very rapidly, which helps designers throughout the design process.

Burry (1997) divides designers into two groups. Some designers accept whatever software programmer (CAD developer) has provided for them. These are the people who are less likely to write macros for their word processing software. Others need to free themselves from these constraints by using some degree of programming, from doing simple customizations to the CAD software they use, to scripting iterative operations, and finally having full access to the code.

Why do designers need to know computer programming? And can one design using the medium of computer programming (Streich, 1992)? CAD developers try to improve their software to meet designers' needs. But no matter how perfect the computer system is, at some point the designer will encounter its limits and want to exceed them, not to have creativity be limited by the system and design be determined by it. To do so, he needs some knowledge of computer programming. Streich (1992) argues that "computer aided designing ends up in programming because only in this way new and original concepts can be transferred from the designer's mind into the instrument computer."

The Smithsonian courtyard enclosure is a good example of the use of computer programming in architectural design and manufacturing (Peters, 2007). The design of the glass canopy, shown in figure F.1, was initiated by Norman Foster's hand drawn sketch. From that point, all the geometry was generated by a computer program, developed incrementally throughout the design process and as part of it by the Specialist Modeling Group in Foster and Partners architecture firm. Because of the complex design of the canopy and its environmental and structural constraints, the parametric script was a necessary tool that gave the designers the ability to explore a large range of design ideas and evaluate them and also the flexibility to accommodate the changes and shifts that are inevitable parts of any design process.


(a) Design of the canopy's control polygon in CAD

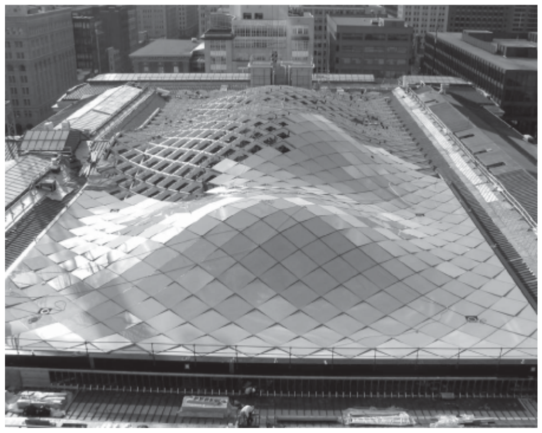

(b) Canopy under construction

Figure F.1: Smithsonian courtyard enclosure (Peters, 2007)

Aish (2003) categorizes architectural design into two categories: conventional and exploratory. The majority of architectural design and construction work is in the first category. The conventional CAD systems that support this category are either low level (line, arc,...) or high level (wall, window,...) and work well for the conventional designers. The exploratory design though, challenges conventions by introducing new forms and geometry, as the above example shows. He argues that one of the key requirements for an 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. In order to do that, the designer has to have necessary programming skills.

It is marked by persistent failure to relate programming and design tasks. With the tools available in the 1980's and 1990's, programming was too often isolated from design situations. Transfer from programming to design work seldom occurred. Today things are different. Every significant CAD system has a built-in programming language (some of these are easier to use than others). Many major design schools have faculty and coursework that engage these tools in design tasks. But there is still a long way to go to achieve perfection, and a lot needs to be done to improve end-user programming in CAD.

End-User Programming

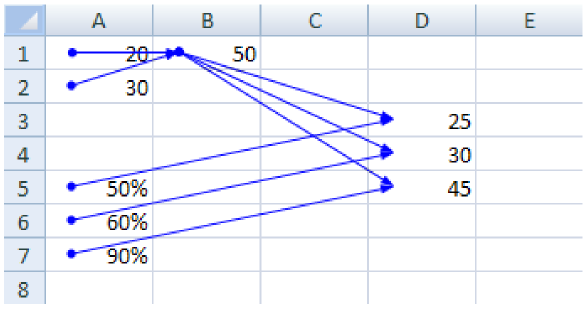
In general, end-user programmers are people who are professional in domains other than programming who need to write programs in the context of their expertise and as part of their job in order to get a task done (Woodbury, 2008). End-user programmers may use any programming language independently or as part of a software depending on the nature of their job, from spreadsheets for accountants, and Processing for graphic designers, to SolidWorks for mechanical engineers, and AutoCAD for architects.

For an end-user programmer, the focus is on the task in hand and not the programming, and the goal is mostly to do the task, rather than to produce efficient, reusable code (Woodbury, 2008). 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 GUI.

There are several barriers to learning and using programming for novices and end-user programmers. Kelleher & 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 & Petre (1996) consider *closeness of mapping* to be one of the important criteria for evaluating visual programming languages. 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 & Myers (1996) emphasize on 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. These are some of end-user programming issues that we try to address in this project.

Spreadsheets are good examples of end-user programming environments that bring programming directly to the problem domain. In spreadsheets, the problem domain is the cells and numbers in them. User can write programming statements directly into the cells and see immediate feedback in the form of numbers. Figure F.2 shows the Trace Precedent feature in an Excel spreadsheet. This is an equivalent of the hierarchy graph in any dataflow programming language that shows the flow of data in the program (for example, in Generative Components' symbolic view, shown in figure F.3(a)). The difference is that this graph is displayed in the problem domain using the cells and nodes, instead of in a separate node-link diagram.



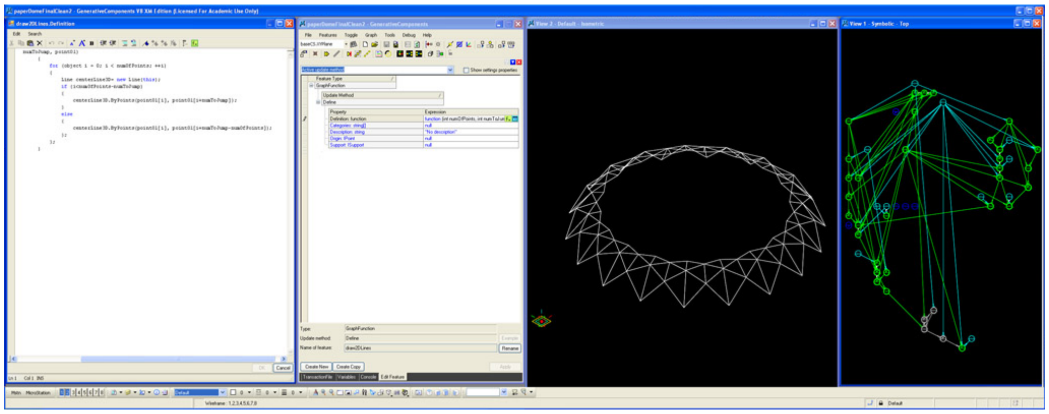
	A	B	C	D	E
1	20	50			
2	30				
3				25	
4				30	
5	50%			45	
6	60%				
7	90%				
8					

Figure F.2: Trace Precedent feature in an Excel spreadsheet

Programming in the model

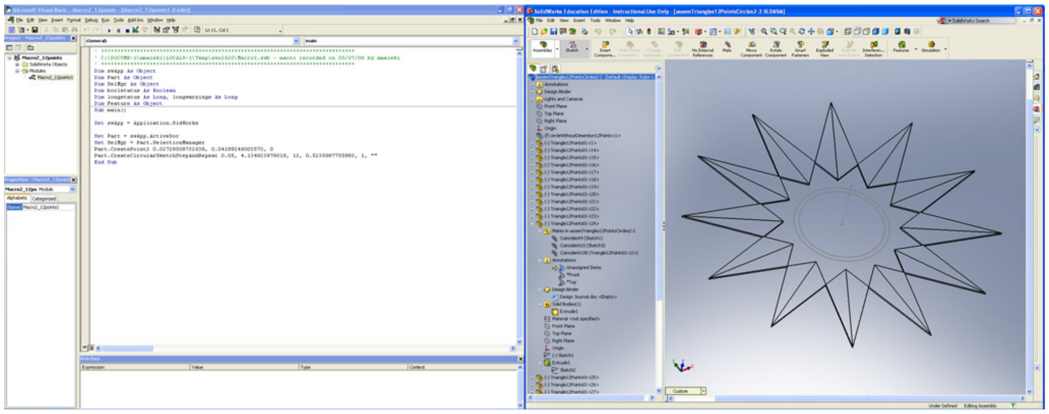
Users of Computer-Aided design systems have at least one special skill in common: they can understand and relate to a 2D or 3D model very easily. They are experienced and comfortable with the model and it makes sense to them. In working with CAD systems, their purpose usually is to create a model. This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

In most CAD systems, the model, which is the result of any design or draft, is represented in a window in the interface called the model view. There are usually other windows showing the programming environment of the software, some for editing purposes, and others that show symbolic representations of the model depending on the type of system and the programming paradigm being used. These windows may be completely separate or may have some overlap with one another. Figure F.3 shows these windows in two different CAD interfaces, Generative Components and SolidWorks. Three forms of representation of a bezier curve in Generative Components are displayed in figure F.4. Note that the hierarchy of objects and the underlying script are completely separate from the model of the bezier curve.



```
numToJump, point01)
    {
        for (object i = 0; i < numOfPoints; ++i)
        {
            Line centerLine3D= new Line(this);
            if (i<numOfPoints-numToJump)
            {
                centerline3D.ByPoints(point01[i], point01[i+numToJump]);
            }
            else
            {
                centerline3D.ByPoints(point01[i], point01[i+numToJump-numOfPoints]);
            };
        };
    }
```

(a) Generative Components interface. Windows from left to right: script editor, property manager(edit view), model view, symbolic view,

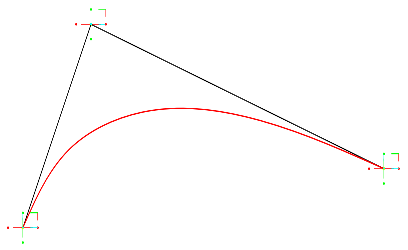


```
' ******************************************************************************
' C:\DOCUME~1\smaleki\LOCALS~1\Temp\swx2620\Macro1.swb - macro recorded on 05/07/08 by smaleki
' ******************************************************************************
Dim swApp As Object
Dim Part As Object
Dim SelMgr As Object
Dim boolstatus As Boolean
Dim longstatus As Long, longwarnings As Long
Dim Feature As Object
Sub main()

Set swApp = Application.SldWorks

Set Part = swApp.ActiveDoc
Set SelMgr = Part.SelectionManager
Part.CreatePoint2 0.02729508732638, 0.04189244003573, 0
Part.CreateCircularSketchStepAndRepeat 0.05, 4.134923979019, 12, 0.5235987755983, 1, ""
End Sub
```

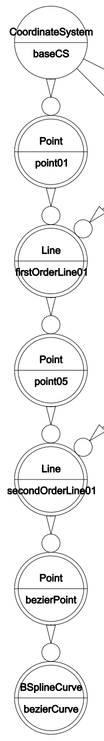
(b) SolidWorks interface on the right, including property manager bar and the model view, along with Microsoft Visual Basic window on the left, for writing and editing macros

Figure F.3: Examples of common CAD interface

As much as designers are good with the model itself, most of them are not as comfortable with programming/scripting environments. Some designers struggle to understand the underlying structure of the programming language and/or the structure of the model. Others have a hard time making sense of the syntax of the program. In most CAD systems, the task of programming and scripting is separate from the task of designing the model, both in the interface and in designer's modes of thinking (see figure F.5). We believe that designers'



(a) A Bezier curve in the model view



(b) Symbolic representation of the Bezier curve in Generative Components

```
int order = firstLines.Count;
double t = 0;
for (int k = 0; k < 11; k++)
{
 Line tempLine = new Line();
 tempLine = firstLines;
 for (int i = order; i > 0; i--)
 {
  if (i>1)
  {
   Point point01 = new Point();
   point01.ByParameterAlongCurve(tempLine, t);
   Line stepLine = new Line();
   for (int j = 0; j < i-1; j++)
   {
    Line stepChild = new Line(stepLine);
    stepChild.ByPoints(point01[j], point01[j+1]);
   }
   tempLine = stepLine;
  } else
  {
   Point point01 = new Point(this);
   point01.ByParameterAlongCurve(tempLine[0], t);
  }
 }
 t = t + 0.1;
}
```

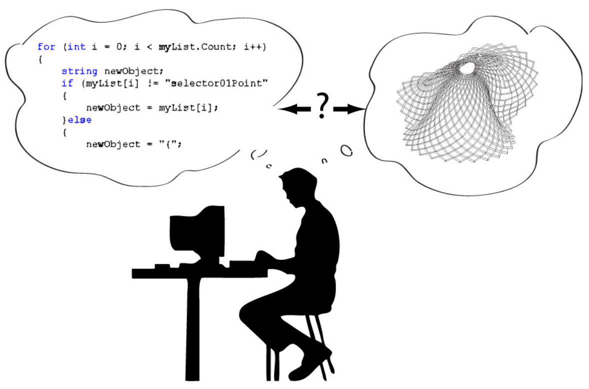
(c) Part of the code that produces the Bezier curve

Figure F.4: Three representations of a Bezier curve in Generative Components

unique ability in understanding the model can help them with programming, if we are able to successfully bring elements of the program to the model view. By doing that, we may help expert users as well as novices by making programming elements more accessible to them and helping them better link the model with its underlying structure.

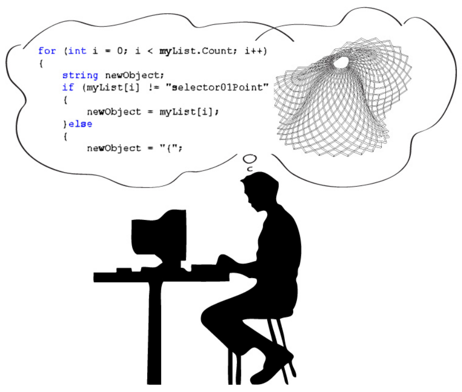
There are several different programming elements to be shown in the model, such as relationship between objects, loops, variables, conditional statements, and lists. Of course some of them are harder to visualize than others, such as functions or variables. And there are different ways of visualizing these elements. It is important to make sure adding this programming information to the model does not cause data clutter and confusion.

There has been some attempts by CAD developers to make programming more accessible



```
for (int i = 0; i < myList.Count; i++)
{
    string newObject;
    if (myList[i] != "selector01Point"
    {
        newObject = myList[i];
    }else
    {
        newObject = "{";
```

(a) Programming is usually separate from the model



```
for (int i = 0; i < myList.Count; i++)
{
    string newObject;
    if (myList[i] != "selector01Point"
    {
        newObject = myList[i];
    }else
    {
        newObject = "{";
```

(b) Bringing programming closer to the model

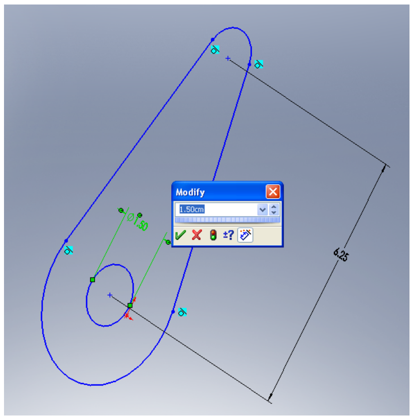
Figure F.5: Computer programming and the design of the model in CAD

and understandable for their users, though programming has rarely made it to the model itself. Some of the existing CAD systems have tried to use the model view for purposes other than displaying the model and selecting objects, such as adding and editing dimensions on the spot. Others have used visual programming to substitute textual representation of the program with visual elements (Figure F.6). Although program is visually represented in visual programming, using boxes and links, etc, it is still different and separate from the model.

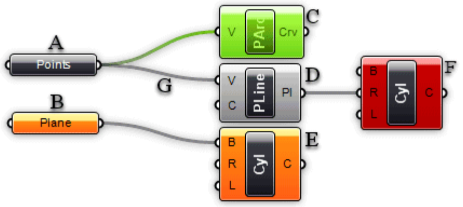
The experiment

In this project, we brought the concept of lists, as a programming element, to the model view in Generative Components (GC) software. Then we ran an experiment with two groups of participants, expert GC users and non-GC users, in which they performed a task that involved making lists of points, using three different methods to make these lists:

- Script window: Typing the names of the points in the list in the script window (GC's existing technique)
- Script window/ Model view: Selecting the points by using a combination of control key and mouse click in the model and then modifying the list in the script window. (GC's existing technique)



(a) Dimensioning in Solidworks



(b) Visual programming in Rhino's Grasshopper

Figure F.6: Visualizing programming in existing CAD systems

- Model view: Using the list objects and adding points to the lists in the model view. (Our prototype)

The data shows some differences between the techniques, but nothing significant were found. However, there is more that needs to be done to improve the experiment and continue the project.

This paper and project were done in IAT 812 course, Cognition, Learning and Collaboration, at School of Interactive Arts and Technology, Simon Fraser University and is part of a larger project, the author's PhD thesis.

Methods

Hypothesis

Our hypothesis in this experiment was *"It is positively effective to bring the task of making lists to the model view."*

The null hypothesis was *"It is not positively effective (either negatively effective or not effective at all) to bring the task of making lists to the model view."*

To test our hypothesis, we set up an experiment to compare three ways of doing a task that involved making several lists, *in the script window*, *in both the script window and the model view*, and finally *in the model view*. We predicted that the combination of script and model view was better than scripting alone, and the model view technique was better than both other techniques.

Participants

There were two groups of participants in this study. The first group of participants was selected from a community of expert Generative Components users, in a workshop and conference on computational geometry (Smart Geometry Workshop and Conference, March-April 2009, San Francisco). Participants in this group were asked to finish three tasks and their performance was recorded (as explained later), but their written and oral feedback was more important for us than their performance in the experiment, because we knew that their expertise with the software would affect the results. Users who are familiar with a software are usually used to its existing techniques and interface and are fast and comfortable with them. If we introduce a new way of doing something, it takes them more time to do a task using this new technique than using existing techniques that they already know.

Participants in the second group had never used Generative Components before the experiment. We chose this group to avoid the effects of participants' expertise on the results. The non-user participants were randomly selected from a group of graduate students in a multidisciplinary school. They received more instructions than the first group for the first two conditions. The first group were already familiar with them. (The words *user* and *non user* are used to refer to these two groups of participants in the remaining sections of this paper.)

There were 10 participants in the first group (GC users), 3 females and 7 males, and 9 participants in the second group (non-GC users), 5 females and 4 males. Participants had different occupational and educational backgrounds, mostly architects in the first group and computer scientists and engineers in the second group. Participation was voluntary and participants signed informed consent forms at the beginning of the experiment. The experiment was approved by Simon Fraser University ethics committee through IAT 812 course, spring 2009.

Materials

Participants were asked to create a surface over a set of 9 points. This task required them to make a list of three lists with three points in each list. So they had to make three regular lists and one nested list. Here is an example of the final list used to make a surface: {{point03, point04, point08},{point02, point06, point01},{point11, point13, point10}}

Currently, Generative Components provides two ways of making a list of objects. One is by simply typing their names in the required field in the edit window or script window, in a couple of curly brackets, e.g. {point01, point02, endPoint05, myPoint02}. This is the most common way of making lists in most textual programming languages. In this technique, the user needs to know the name of the objects in the list beforehand and type them correctly.

The other way of making a list in Generative Components is by holding down the Control key on the keyboard and hovering the mouse over the point that user wants to select. By doing that, the name of the point is added to the list in the edit window. User needs to repeat that for each point in the list. During the selection process, user has to have an eye on the edit window as well as the model view to see the list and make sure the names are correct and the brackets are in the right places. In order to make a list of lists, user has to open the script window and modify the list and add brackets in the right places. These two techniques are the first two conditions of the experiment.

For the third condition, we created a prototype in Generative Components that allowed the user to select the members of the list in the model view, by moving a *list maker* object over them. When the list maker is close enough to an object, the object is added to the list and the list is displayed beside the list maker object. User can use multiple list makers and list makers can select other list makers to create nested lists. During this process, user does not need to look at the edit view or the script window and can only attend to the model view. At the end, user puts the name of the final selector object in the required field in the edit window.

Experimental design and procedure

Before starting the experiment, we asked each participant about their past experience with the software, in order to know which group they belonged to and adjust the instructions accordingly.

For this experiment, participants completed the same task of making a surface in 3 conditions (as explained above). The time that it took them to finish each task were recorded on the spot by the researcher. In order to control the effect of the order of the techniques, half of the participants did the experiment in the regular order and the other half did it in the opposite order. We did not intend to do any between group tests. Therefore, there was no need to balance the groups of expert and novice users. The effect of the first group's participants' expertise with the software were considered in the interpretation of the results.

Experiments started with a general description of the software, different windows of the interface, and a quick demo of how to create a shape in GC. At first, we explained the technique of creating a list in that condition, in the script window, in both the script window and the model view, and in the model view (see Materials section for more details). For each condition, there was a practice phase and the main experiment. They practiced each technique by creating a curve that required creating a list of points. The experimenter was available to answer their questions and help them if needed. Some of the expert GC users requested to skip the practice for some of the conditions. Then we explained the process of making a nested list of points and a surface on that list, using oral and written instructions and diagrams. We skipped some of these instructions for the expert group, as they didn't need them. The main task for that technique followed, when they created a surface on 9 points, as fast as possible.

A questionnaires were filled out by participants. Some general questions such as age, gender, and occupation were answered at the beginning, as well as rating participant's expertise in programming, CAD systems, and Generative Components on a scale of 0-10. After each condition, participants were asked to rate that technique for ease of use and clarity on a scale of 1-10. At the end of the experiment, they were asked to choose one of the three techniques as the easiest to use, the hardest to use, the most confusing and their preference.

Results

The quantitative data that results from this study can be categorized into two groups: users' performance in the task and users' feedback collected in a questionnaire during and after the experiment. Each group of data was presented in graphs and also analyzed separately using SPSS software.

Users' Performance

For each condition (script, script/model, and model), users performed a task of creating a surface on a list of 9 points. Time was recorded for each task as a measure of participants' performance in each condition. Figures F.7 and F.8 show the average time in seconds for each condition and for the two groups of GC users and non GC users.

	Time	
	User	NonUser
Script	158.0	133.8
Scrip/Model	85.6	89.2
Model	116.1	99.3

Figure F.7: Average time (in seconds)

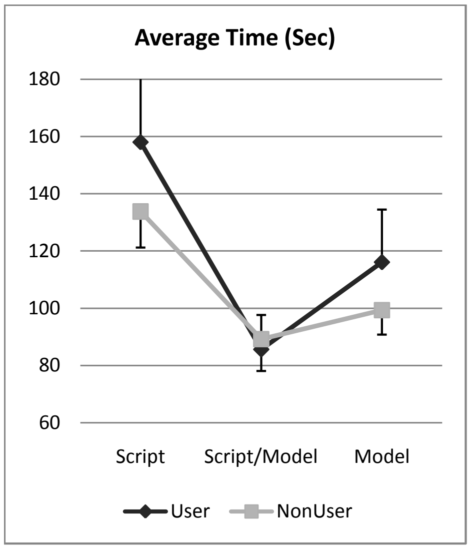


Figure F.8: Average time (in seconds), along with the standard error of the means

Both groups performed better in the script/model condition and worse in the script condition. None of the groups performed as we expected in the model condition. The data was analyzed using a repeated measure ANOVA (Analysis of Variance). Here is a summary of the results:

- Users: The ANOVA shows that there is a marginally significant difference between the means for this group at a 0.1 level of confidence. The Post Hoc test shows that the difference is between the script and the script/model conditions.

- Non users: The result of the test shows that there is a significant difference between the conditions at a 0.05 level of confidence for the non user group. The Post Hoc test of Tukey for this group (considering the equality of variance) shows that there is a significant difference between the script condition and the script/model condition in regards to the time of the task at a 0.05 level of confidence.

Users' Feedback

Participants filled out a questionnaire during and after the experiment. The questions they were asked in between the sessions aimed for their feedback about each session, while the post experiment questions asked them to compare the three techniques.

Feedback during the experiment

After each condition, they answered two questions about that condition. The questions asked them to rate the technique in that condition for *ease of use* and *clarity* on a scale of 0 to 10. The average rate of each condition for each question is presented in figures F.9 and F.10.

- Users: This group found the scripting technique the hardest and most confusing technique. The model was the easiest for them to use, but not the most clear one.
- Non users: This group, however, found the model technique the most confusing and the scripting technique very clear. The script/model technique was the easiest for them to use.

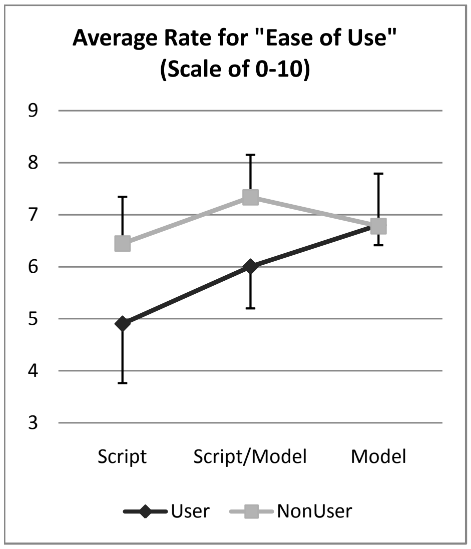
However, the ANOVA test shows no significant difference at a 0.05 (or 0.1) level of confidence between the three conditions in participants' response to these two questions in either groups.

Feedback after the experiment

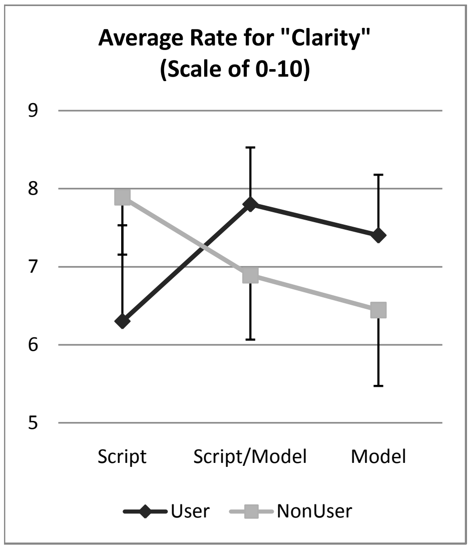
After the experiment, participants were asked to compare the techniques by choosing one of the conditions as *the easiest*, *the hardest*, *the most confusing*, and *the one they prefer*. Figures F.11 and F.12 show the results of this comparison.

	Ease of use		Clarity	
	User	NonUser	User	NonUser
Script	4.9	6.4	6.3	7.9
Scrip/Model	6.0	7.3	7.8	6.9
Model	6.8	6.8	7.4	6.4

Figure F.9: Average rates (scale of 0-10) in response to the questions after each session



(a)



(b)

Figure F.10: Average rates (scale of 0-10) in response to the questions after each session, with the standard errors of the means

- Easy/Difficult (Graphs (a) and (b))
  - User: Scrip/model seems to be the easiest and scripting the hardest technique.
  - Non User: Again, script/model is the easiest. It's not clear which one of the other two techniques is the hardest for this group.
- Clear/Confusing (Graph (c))
  - User: Scripting is chosen as the most confusing technique and model as the most clear one.

  - Non User: This group, on the other hand, found the model technique very confusing and scripting the most clear one.

- Preference (Graph (d))
  - User: Scripting is the least preferred technique among the three for this group.
  - Non User: There is no difference between the three techniques.

	The easiest		The hardest		The most confusing		The preferred one	
	User	NonUser	User	NonUser	User	NonUser	User	NonUser
Script	2	2	5	3	5	1	2	3
Scrip/Model	4	4	2	2	3	2	4	3
Model	4	3	3	4	2	6	4	3

Figure F.11: Total number of times each technique was chosen in the post experiment comparison

Discussion and Conclusions

Limitations

To test the idea of programming in the model in CAD systems, we needed to choose a CAD system as our platform. We chose Generative Components (GC) because we had access to the community of GC users and also a research relationship with GC developers in Bentley systems that provide us with support for our projects.

We didn't have access to GC's source code in order to manipulate its functionality and interface. So we used GC script, which is designed mostly for creating models, to create a prototype for our project. GC script did not allow the prototype to reach its desired functionality. For example, there were very limited editing capabilities after the lists were created. Also it was not possible to use any mouse buttons to interact with the model, so participants had to spend more time navigating the list makers through the model. These problems may have resulted in some of the unexpected results in this experiment, as they make the prototype uncomparable with GC, and if solved (probably by using another programming software) may in fact change the results of future studies.

Discussion and future work

As noted in the result section, we did not find any significant statistical results from this experiment to prove our hypothesis. The small number of participants, (10 in one group and 9 in the other) could be an important factor, as well as the technical problems with the prototype, as explained above.

Another important problem with this experiment was the choice of participants in the non user group. Most of the participants in this group were in fact computer programmers (7 computer scientists, 1 engineer, 1 designer) and had many years of experience with several different scripting and programming languages. Due to this fact, they preferred writing scripts to interacting with the model. As we proposed programming in the model to be an effective approach to programming for architects and designers, participants with architectural or design background and no prior experience with GC would have been a better choice for this study.

The participants in the user group had prior experience with GC and were familiar with the script and script/model techniques. It takes some time and practice to get familiar with a new technique and this might be the reason they did not perform well with the model technique, in spite of having design and architectural background. This, again, suggests further studies with non GC user designers.

Most of these GC users also had experience with programming in CAD. That is one of the reasons they attended the Smart Geometry workshop and conference as tutors, presenters, or participants. It would be interesting to investigate the effects of programming in the model on designers who do not have any programming experience. This technique may in fact work better for them in learning end-user programming in CAD, than designers who are already familiar with scripting.

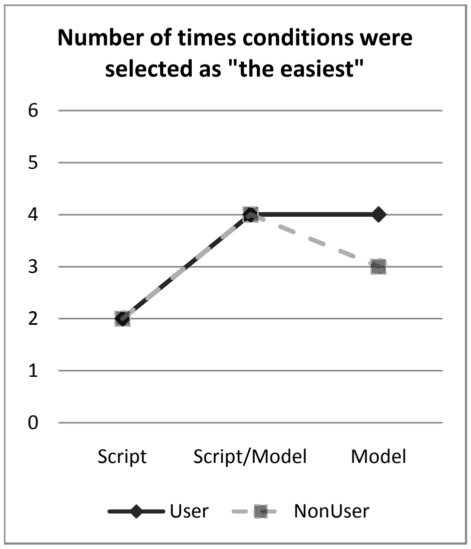
The concept of lists in programming was chosen for this study as the first step towards bringing programming to the model. However, lists are not the most complicated and troublesome programming concepts for CAD users. There are more confusing programming elements to study, such as functions and loops. The effects of programming in the model and the difference it makes in users' performance may be more significant for more challenging concepts. This can also be a possible future path for this project.

Participants written and oral feedback on the three techniques can also provide some interesting insight for future work, especially on the difference between designers/architects and computer scientists in their preference between script and model. Here are a few of those comments. A more comprehensive and detailed qualitative study is necessary for any conclusions.

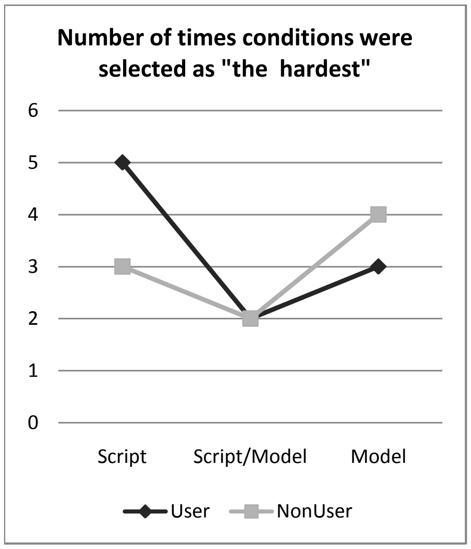
- Architects/designers:
  - "Typing all of that is hell." (Refers to the scripting technique.)
  - "Typing the names correctly and remembering the syntax is very difficult."
  - "[The model technique] is very easy and user friendly."
  - "[The model technique] is visual, fun, and clear."
- Computer programmers:
  - "It's easier to use the method I am used to, [scripting], rather than interacting with the mouse in GC model."
  - "Typing is so easy and direct."
  - "Copying and pasting [in script] is easier for me than moving a circle in a model full of points."
  - "[Scripting] gives me a sense of coding. Like I am actually creating something."
  - "I hated the model technique."

Conclusions

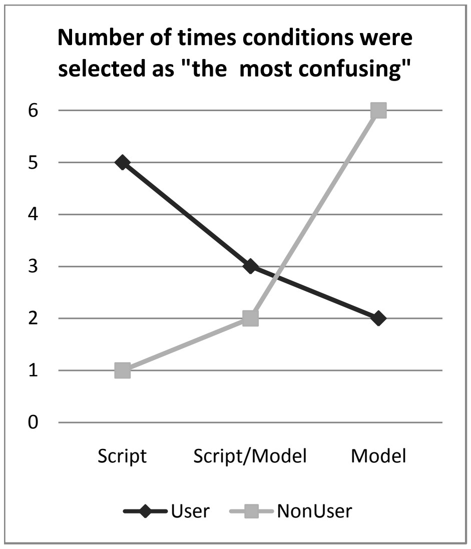
This experiment was designed to test the hypothesis that bringing end-user programming to the model in CAD systems positively affects designers' experience and performance by using their expertise and understanding of the model for programming. However, due to the limitations in the prototype and the choice and small number of participants, we didn't find any significant results to prove or reject the hypothesis. We suggest further investigations on this topic with improved prototypes and experimental design, in order to understand the effects of programming in the CAD model for non-programmer designers.



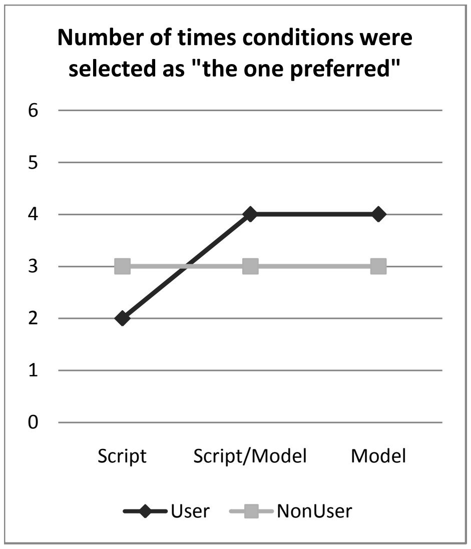
(a)



(b)



(c)



(d)

Figure F.12: Total number of times each technique was chosen in the post experiment comparison

Appendix G

Visualizing Lists in CAD Models

Abstract

Lists of points are used in some Computer Aided Design systems to create simple or complex shapes such as lines, curves, and surfaces. During the task of making these lists, users can benefit from visual feedback that shows them different attributes of the lists including membership, order, and hierarchy. Visualizing these attributes presents a challenge that requires exploration and investigation of several different visualization design principles. A few of those principles are presented in this project in the form of four brushing techniques. In addition, a prototype was implemented for this project that allowed the simulation of different situations and comparison between these brushing techniques.

Introduction

Architects and engineers use Computer Aided Design (CAD) systems to design and develop 2D and 3D models of their final design, whether it is an engine, a house, or an airport. In more complex projects, designers sometimes have to use computer programming in CAD to do the tasks that can only be done or can only be done efficiently by using programming. In those cases, designers become end-user programmers: non-professional programmers who need to write code to get a task done in the domain of their own expertise. As it is known in the literature, there are many issues in end-user programming in general and also issues that are specific to CAD.

For my PhD thesis, I am working on end-user programming in CAD. More specifically, I am working on *programming in the model* in CAD as a means of using designers' visual and spatial abilities in end-user programming. In other words, we are working on bringing computer programming closer to the model in CAD to minimize the mental shift between the two tasks of design and programming.

In a parallel course, IAT 812 Cognition, Learning, and Collaboration, I explored this idea by choosing *lists* as a programming element and implementing a prototype in Generative Components (a CAD system from Bentley Systems) to study the effects of bringing the task of making lists to the model in CAD. In this course, IAT 814 Knowledge, Visualization, and Communication, I looked at the same problem from a visualization perspective, in order to find out some of the challenges in visualizing lists in the model and explore some of the solutions.

For this purpose, a prototype was created in Processing that enabled us to explore different brushing techniques in several different situations. In this paper, we explain the project and some of the visualization challenges specific to this project. We then introduce some of the visualization principles that can be used to address those issues. Then a brief description of the prototype is included, along with thorough explanation of the brushing techniques used in this project, followed by conclusions and ideas for the future works.

Visualization challenges

In some CAD systems (Generative Components in particular), lists of points are used to make shapes such as line strings and curves. Nested lists are used to create surfaces. The difference between a list and a set is that order is important in the list. A curve created on points 1, 2, and 3 looks different from the curve created on points 2, 1, and 3.

In this project we bring the task of making lists to the model in order to investigate *programming in the model* as an approach to end user programming in CAD. In other words, users make the lists in the model by simply selecting the points and adding them to the lists, without having to worry about the syntax (brackets and commas) in the script editor. However, without any visual feedback, users cannot keep track of the members of the lists and their order and it is equally difficult to modify lists, as users have to use the textual representation of the lists and the names of the points to connect them together.

There is important information about the lists that needs to be displayed to the user. Here are the three most important ones:

- *membership*: Users need to know which points belong to a list and also what lists a point belongs to.
- *order*: As mentioned above, order is important in the lists, as it affects the shape of the final model. Therefore we have to show the order (or the history of selection) of the members of each list.
- *hierarchy*: Nested lists (or two dimensional arrays) are commonly used to make more complex geometries. So there may be lists that contain other lists and this hierarchy among lists has to be visually represented.

In addition to membership, order, and hierarchy, there are other issues that we need to consider. For instance:

- How does this visual feedback combine with the CAD model? Does it interfere with the model? Which technique works better with wireframe models and which one works better with a rendered or shaded model?
- How do these lists work with each other in the model when they share points and overlap?
- How does the visualization of the lists connect with other types of representation, such as list icons or list text?
- How does it scale? Does it work for large lists? Does it work for large or complex models?

In order to find the answer to some of these questions, we explored several brushing and linking techniques to visually represent lists in the model view. Different information visualization design principles were considered for this purpose. Some worked and some didn't. Here is a brief survey of those design elements, mostly cited from Ware (2005) and Bartram (2009):

- *Proximity:* Proximity cannot be used in this project simply because the position of the points in the model are important to the geometry and cannot be changed to group the points together.

- *Similarity:* Similarity in shape, color, and size can be used to group the points. We explore similarity in color in this project. However, similarity in shape was not used for the points due to issues with overlapping lists when one point has to have more than one shape. Shape was used in combination with connectedness for arrowheads.

- *Connectedness:* Connectedness is argued to be a fundamental Gestalt principle and more powerful in grouping objects than color, size, or shape. It is used here in the form of lines that connect members of a list. However, connectedness alone is not enough for distinguishing the lists from each other. When a point belongs to more than one list, there is no way to know which line to follow from that point. Color, shape or other elements should be used with connectedness to make it effective.

- *Closure:* Closed contours divide the space into inside and outside. The objects inside or outside the contour are perceived as a group. We use a contour for each list that wraps around the points in the list. Contours combine with each other very well and are still distinguishable. Color and texture can be used for more complex models with multiple overlapping contours.

- *Transparency:* If contours are filled regions, transparency is necessary to prevent them from occluding each other. It can also be used with shapes, when several shapes (arrows) are in the same place.

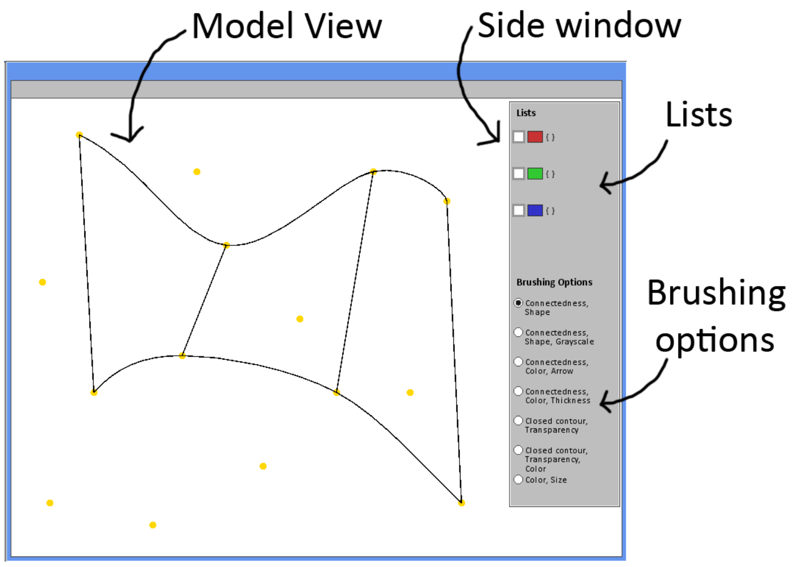
- *Color:* Color can be a powerful visual cue in grouping objects, if used correctly. It can also be used to link the lists in the model view and other representations of them by simply giving them the same colors. In this project we only use three colors of red, green, and blue for the lists. More investigation is required to determine the effect of color and to find out what colors work better for this visualization problem. Color deficiency options must be considered as well.

- *Size:* Size can be used to display order in each list. It can be in the form of point size or line thickness.

- *Direction:* Direction is used here in combination with connectedness to show the order in the lists.

The prototype

To further investigate the challenges of visualizing lists in the model view in CAD systems, we implemented an interactive prototype in Processing. The interface and functions are similar to Generative Components, so that we can compare them in future studies. The model (points, curves, ...) is displayed in the model view. The side window is where list options and brushing options are located (Figure G.1).



Lists

☐ 🟥 { }

☐ 🟩 { }

☐ 🟦 { }

Brushing Options

- [✓ Connectedness, Shape]
- Connectedness, Shape, Grayscale
- Connectedness, Color, Arrow
- Connectedness, Color, Thickness
- Closed contour, Transparency
- Closed contour, Transparency, Color
- Color, Size

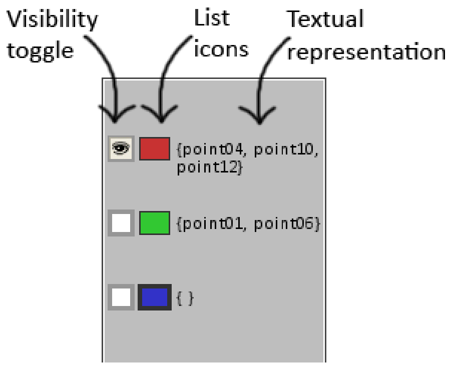
Figure G.1: The prototype

In order to interact with a list, it must be activated by clicking on its icon in the side window (Figure G.2). A thick dark border appears around a list icon that is activated. Only one list can be activated at any time. So clicking on another list will deactivate the first list. A list can also be deactivated by clicking on its own icon.

When a list is activated, points in the model view can be added to it by holding down the control key and clicking on the points. Right clicking on the points removes them from the active list. The textual representation of each list is displayed next to the list icon.

Lists can also contain other lists and make two dimensional lists. Control click on a list icon adds it to the active list and right click removes it from that list.

Brushing techniques are used to visually represent the lists in the model view. The visibility toggle icon next to the list icon can be used to turn the list on and off. Hovering



{point04, point10, point12}

{point01, point06}

{ }

Figure G.2: List options in the prototype

the mouse over the list icon turns the list on temporarily. Additionally, hovering the mouse over a point in the model view shows the lists it belongs to in addition to its name.

Different brushing techniques are provided in the prototype for comparison. These techniques can be explored by using the radio buttons in the Brushing Option section (Figure G.2).

The prototype can be accessed at http://www.sfu.ca/˜mmaleki/iat814/FinalProject/. Unfortunately, the prototype is not bug free. A list of known bugs is provided at the end of this paper.

Brushing techniques

The brushing techniques that we explored in this project can be categorized into three major groups, based on the visualization principle used to link the members of the lists. The three principles are color, connectedness, and closed contour. However, connectedness alone does not suffice to separate the lists from each other, because when two lines go out of a point, there is no visual clue which one to follow for each list. Therefore, we used shape and color as additional visual cues with connectedness. The final categories are:

- Color
- Connectedness and shape
- Connectedness and color
- Closed contour

In this section, we describe each technique and how it does or does not address the visual problems of the lists and also the specific issues in each technique.

Color

- **Membership**

  In this technique, the color of the points is defined by the list they belong to (Figure G.3).

- **Order**

  Size of the points is used to show order in the list. The position (index) of a point in the list determines its size. The smallest point in a list is the first member of that list (Figure G.3).

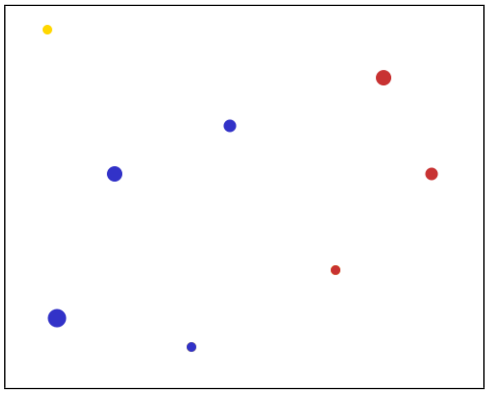


Figure G.3: Color: Membership and order

- **Hierarchy**

  At this point, we do not have any way of showing hierarchy of the lists in this technique. A possible solution can be an additional larger point under the point (it looks like a ring) that represents the nested list. However, the size of that point has to be adjusted to the size of the first point, base on the order of the list.

- **Connecting with the textual lists**

  The color of each group of points matches the color of the list they belong to. So it is easy to make a connection between the list in the model and the list in the side window and its textual representation.

- **On a wireframe model**

  The smaller points may get lost in a complex model and larger points may interfere with the model itself or get confused with the elements of the model, such as the columns. They work in less complex models with colors that are different from the colors used in the model (Figure G.4).

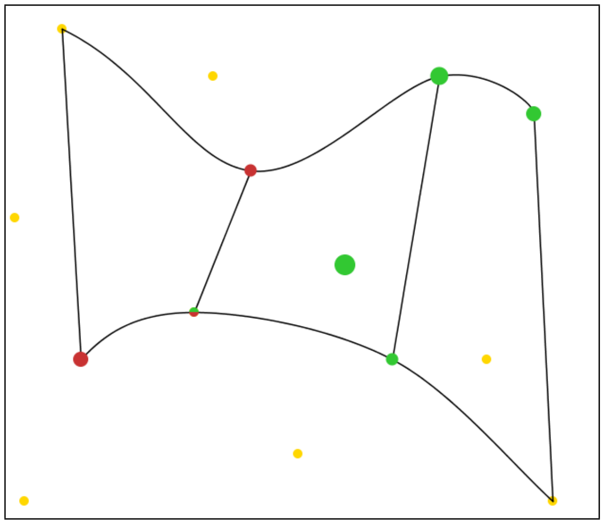


Figure G.4: Color: wireframe model

- **Scaling**

  This technique works for lists of up to 10 points. It does not scale well for larger lists, as the size of the points gets too big and make the model unreadable (Figure G.5).

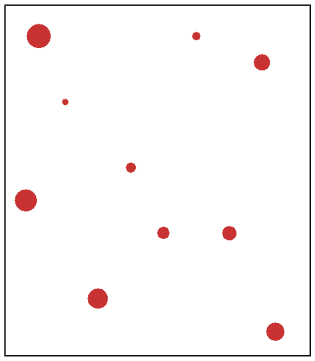


Figure G.5: Color: scaling

- **Combining with other lists**

In this technique, when a point belongs to more than one list, the area of the circle is divided into several sections depending on the number of lists that share that point. Each section follows the color and size of the point in one of the lists. However, this method does not scale well for more than 3 or 4 lists. It gets harder and harder to follow the membership and the order when the number of lists goes up (Figure G.6).

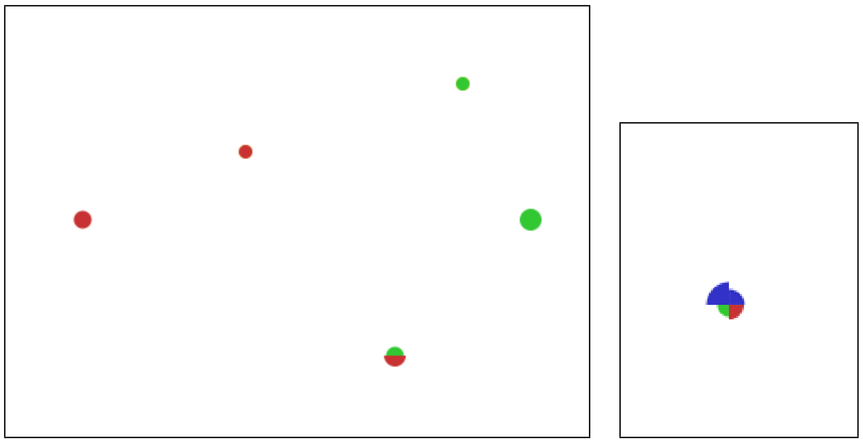

(a) Two lists share a point (b) Several lists share a point

Figure G.6: Color: Combining lists

Connectedness and shape

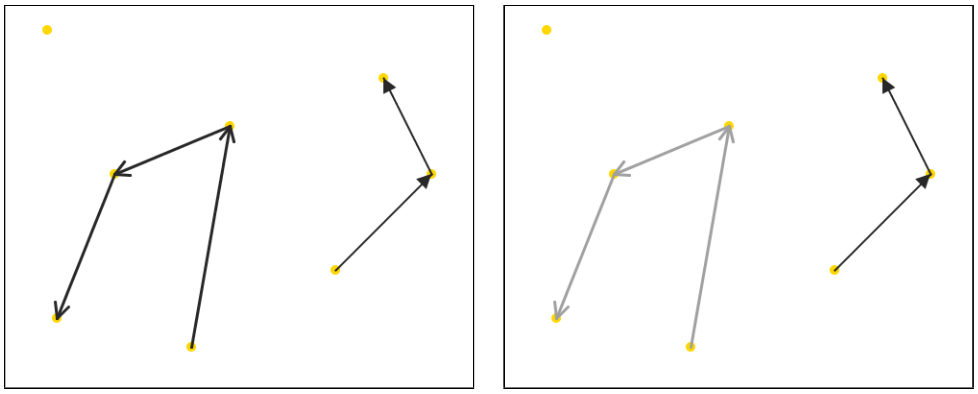
- **Membership**

  Members of a list are connected by a line. In order to distinguish between multiple lists, each list has a different shape associated to it that works like an arrow. Both lines and shapes define the membership of the points. Furthermore, we added a third element of color (only shades of grey) to this technique to see its effect on this brushing technique (Figure G.7).

- **Order**

  The shapes in this technique also carry the role of showing order in the lists. They are like arrows that show the history of user's selection in each list (Figure G.7).

- **Hierarchy**



(a) No color (b) With shades of grey

Figure G.7: Connectedness and shape: Membership and order

One way of showing hierarchy with connectedness is to use double lines for inside lists. However, this method does not scale well for lists with more depth (three dimensional lists and more) (Figure G.8). This technique is not included in the prototype.

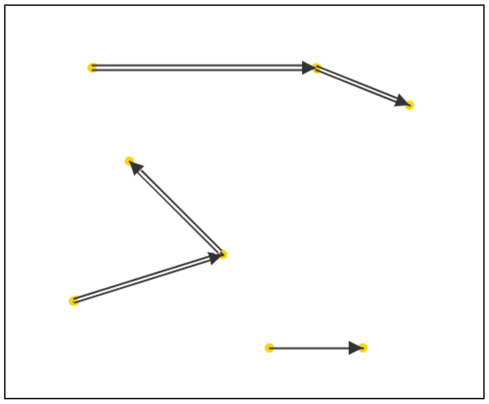


Figure G.8: Connectedness and shape: Hierarchy

- **Connecting with the textual lists**

  There is no good way of connecting the lists in the model with the lists in the side window and the textual lists, except for using the shapes to make some kind of legend. We don't know how clear it will be for the users without implementing and testing it.

- **On a wireframe model**

Wireframe architectural and engineering models usually contain lines and curves that create forms. The lines used in this technique for brushing are very similar to those of the model and can easily be confused with them. It is hard to both follow the lines to find the lists and comprehend the model when the lists are visible (Figure G.9).

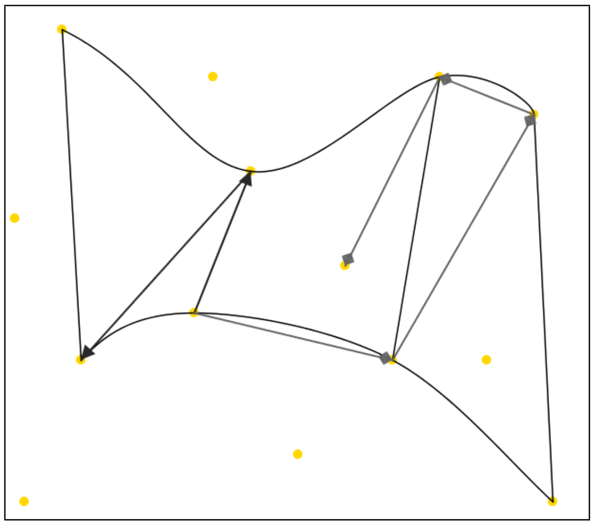


Figure G.9: Connectedness and shape: Wireframe model

- **Scaling**

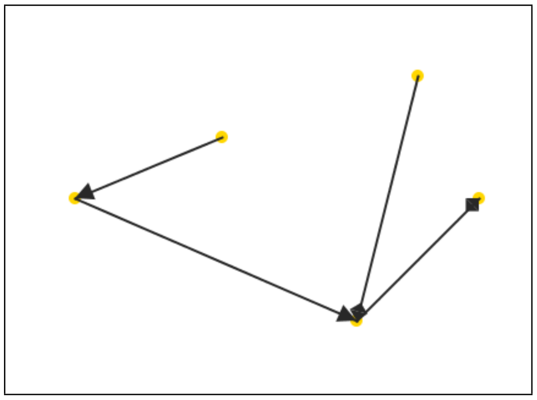
  The technique scales well for large lists. The only problem happens when there are many lists to brush, as we may run out of shapes to distinguish them.

- **Combining with other lists**

  When more than two or three lists share a point, the shapes get cluttered over each other and occlude each other, to a degree that they can be completely useless and unreadable, especially in the first method that everything has the same color (Figure G.10).

- **Other issues**

  A line is only meaningful when there are two points for it to connect. So when there is only one point in the list, there can be no line. In order to solve this issue we can put a single shape on the point (like an arrow without its line) to show which list the


(a) Two lists share a point

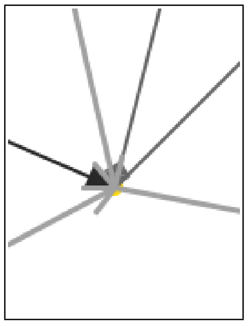

(b) Several lists share a point

Figure G.10: Connectedness and shape: Combining lists

point belongs to. As soon as the second point is added to the list, this single shape disappears and the points are connected and the shape follows the order of selection (Figure G.11).

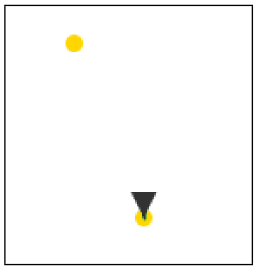


Figure G.11: Connectedness and shape: Single member lists

Connectedness and color

- **Membership**

  Membership is shown by a line that connects the members of a list. However, connectedness alone is not enough to separate the lists in the model. Color is added to the connectedness to assist in visualizing membership (Figure G.12).

- **Order**

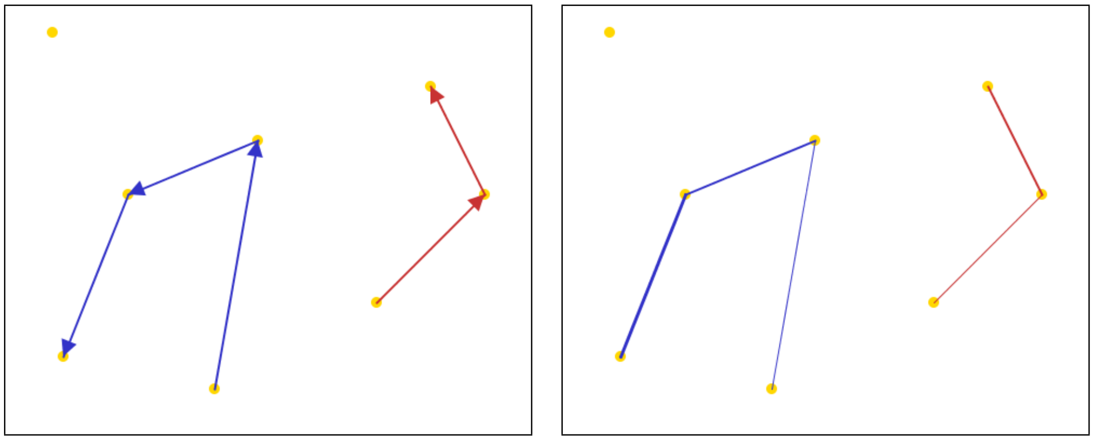

(a) Arrows show order (b) Line thickness shows order

Figure G.12: Connectedness and color: Membership and order

Two different techniques are used here to show order. In the first technique, arrows are used to show the history of selection and consequently the order of the lists. In the second technique, line thickness is used for that purpose. The thickness of the lines depend on their position (index) in the lists. The thinnest line is always the first member of the list (Figure G.12).

- **Hierarchy**

  Like the previous method, hierarchy can be shown by double and triple lines. The problem occurs for lists with more than two levels of hierarchy (Figure G.13).

- **Connecting with the textual lists**

  Color can be a good visual cue to connect the lists in the model with their corresponding lists in the side window and their textual representations.

- **On a wireframe model**

  Again, lines in this brushing technique can interfere with lines and curves in the model. I find it specifically harder to follow the line thickness in the second technique when combined with a wireframe model. However, if the wireframe model is in greyscale and no other color is used in it, the color of the brushing lines can separate them from the model better than the previous technique (Figure G.14).

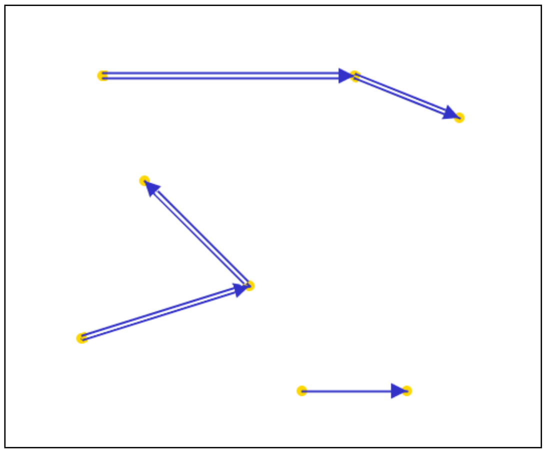


Figure G.13: Connectedness and color: Hierarchy

- **Scaling**

  There is scaling issue for the line thickness when the lists contain too many points and the lines get too thick (Figure G.15).

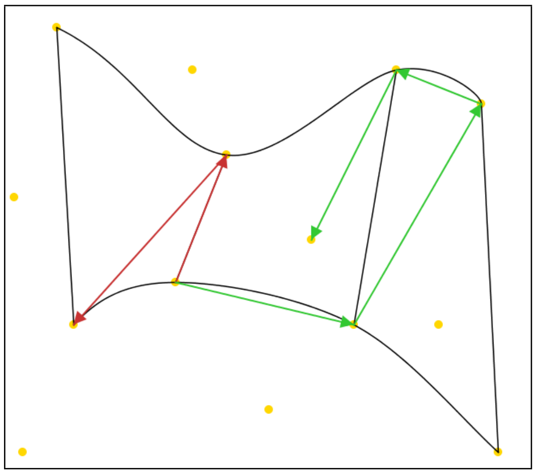
- **Combining with other lists**

  The arrows cover each other and become hard to read when more than one or two lists share a point. But the color helps make it a little better than the previous method (connectedness and shape) to follow the lists. Thickness seems to be much better in scaling in this situation as there are no arrows and lines are easier to follow (Figure G.16).

- **Other issues**

  The same problem applies here for the lists with just one member, as there is no way to produce a line with only one point. A single arrow can be used to show the membership for that point until the second member is added to the list (Figure G.17).

  However, the thickness method still does not work with two points. Line thickness is meaningful only when there are more than one line to compare. Point color can be used here to show the membership in single member lists and to show order in two-member lists (Figure G.18).


(a) With arrows

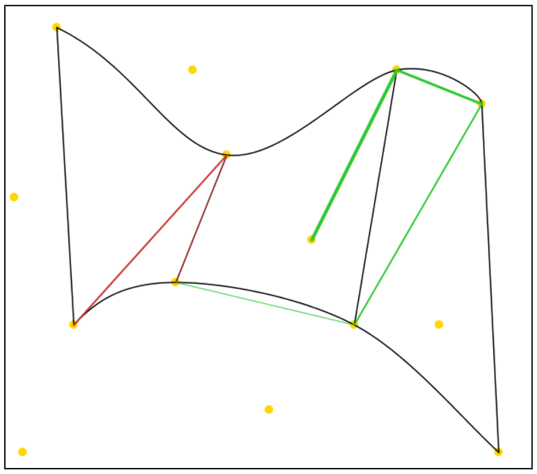

(b) With line thickness

Figure G.14: Connectedness and color: Wireframe model

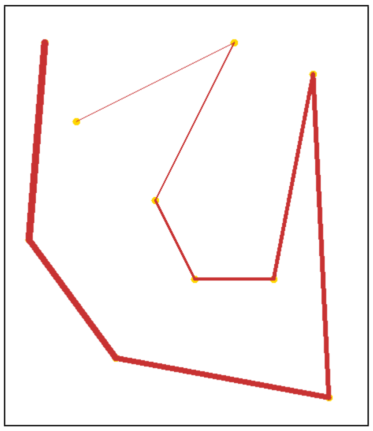
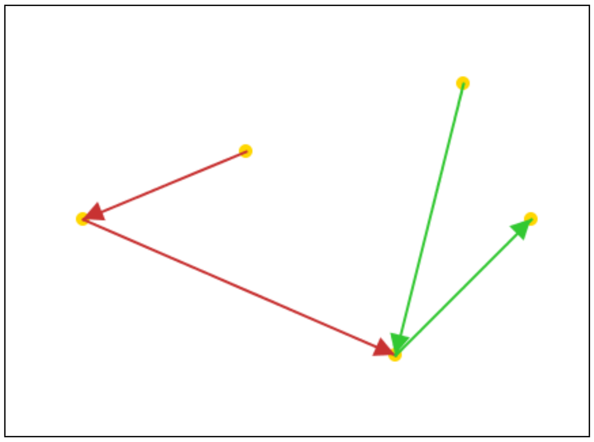
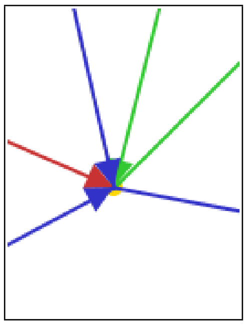


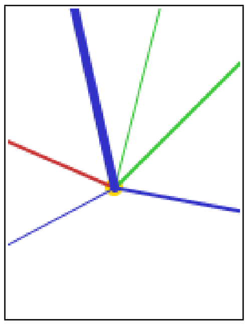
Figure G.15: Connectedness and color: Scaling



(a) Two lists share a point



(b) Arrows: Several lists share a point



(c) Thickness: Several lists share a point

Figure G.16: Connectedness and color: Combining lists

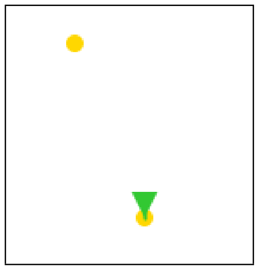


Figure G.17: Connectedness and color: Single member lists

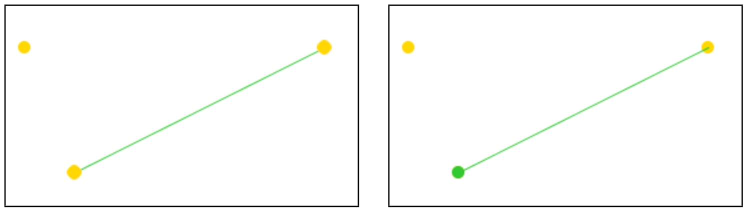


Figure G.18: Connectedness and color: Two-member lists and thickness

Closed contour

- **Membership**

  A closed contour wraps around the members of a list. The contours can have the same color or have different colors that reflect different lists. Both methods are capable of showing membership. Transparency is also used to allow the contours to overlap without occluding each other (Figure G.19).

- **Order**

  At this point, there is no way of showing order in this technique. A possible solution is adding numbers besides points that reflect their position (index) in the list.

- **Hierarchy**

  When a list contains another list, there is a contour for the inside list and another one that wraps around that list and any other member of the bigger list (Figure G.20).

- **Connecting with the textual lists**

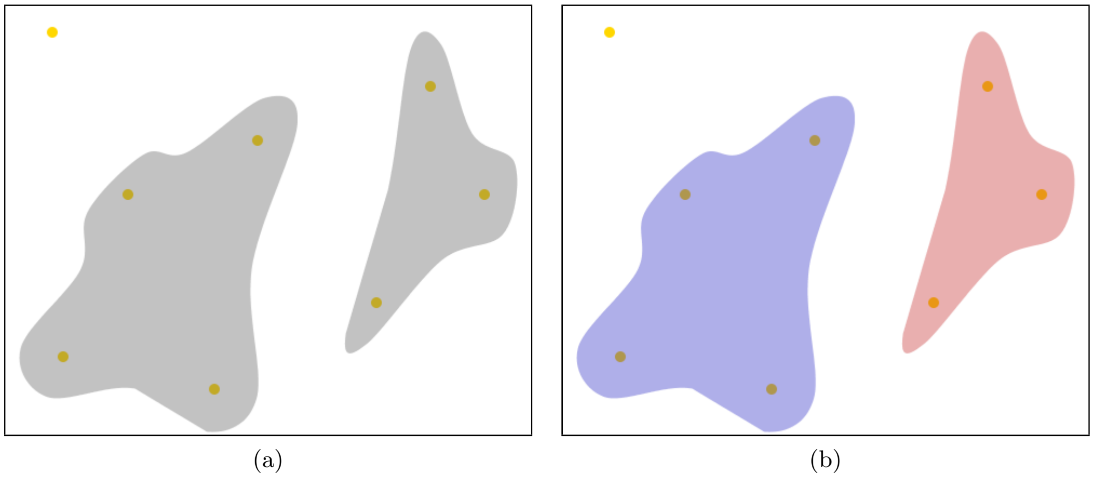


Figure G.19: Closed contour: Membership

The monocolor contours do a good job in linking the points in the lists and showing hierarchy as well as the colored ones do. But they do not connect the lists in the model window with the lists in the side window and the textual representations of them. Color works better in doing that.

- **On a wireframe model**

  Since wireframe models only contain curves, lines, and points, filled closed contours seem to work well with those models. they do not interfere with the model and are easy to follow (Figure G.22). More investigation is necessary for other types of models, such as shaded and rendered models.

- **Scaling**

  Scaling issues may apply to very large lists, due to the large area of contours that covers large parts of the model.

- **Combining with other lists**

  Transparent contours seem to be more effective than connectedness and size when lists overlap in the model. Even without color separation, contours are easy to follow in such situations, as we are good at following curves (Figure G.23).

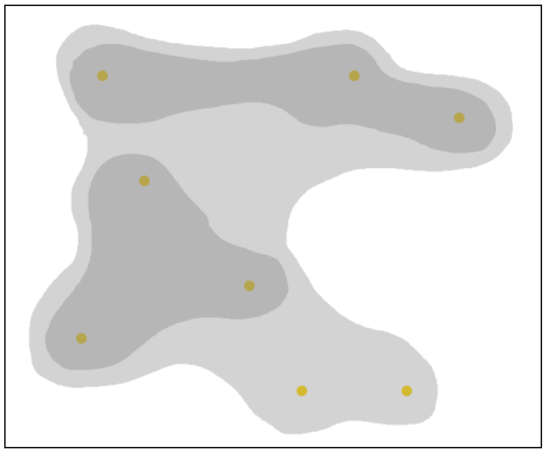
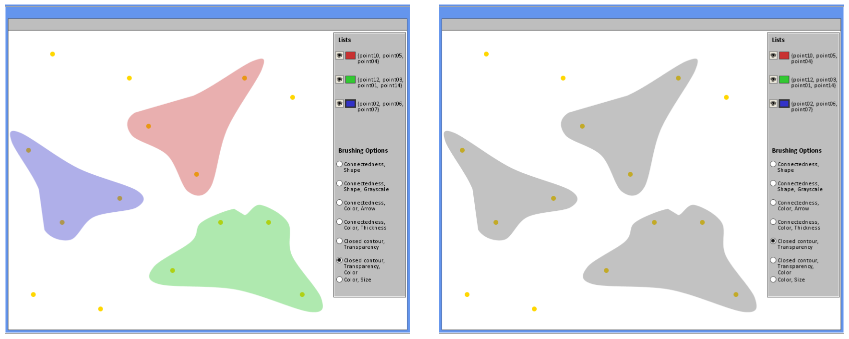


Figure G.20: Closed contour: Hierarchy



Lists

🟥 {point10, point05, point04}

🟩 {point12, point03, point01, point14}

🟦 {point02, point06, point07}

Brushing Options

(a)

- ○ Connectedness, Shape
- ○ Connectedness, Shape, Grayscale
- ○ Connectedness, Color, Arrow
- ○ Connectedness, Color, Thickness
- ○ Closed contour, Transparency
- [✓ Closed contour, Transparency, Color]
- ○ Color, Size

(b)

- ○ Connectedness, Shape
- ○ Connectedness, Shape, Grayscale
- ○ Connectedness, Color, Arrow
- ○ Connectedness, Color, Thickness
- [✓ Closed contour, Transparency]
- ○ Closed contour, Transparency, Color
- ○ Color, Size

Figure G.21: Closed contour: Connecting with other types of data

- **Other issues**

  When there are many objects in the model and lists contain points that are located far apart from each other in the model, contours may have to get strange and more complex shapes to avoid covering the points that don't belong to their lists. It may also become hard to follow them when they get too big.

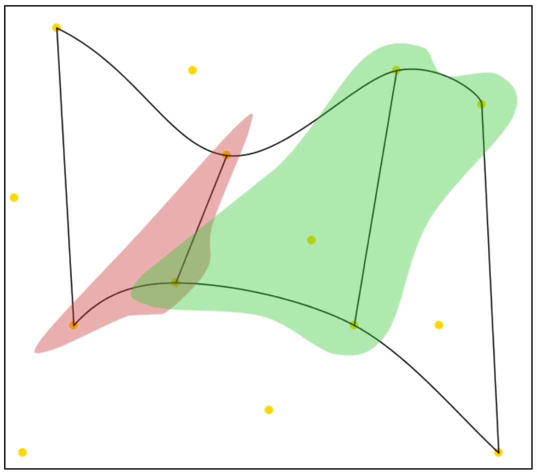


Figure G.22: Closed contour: Wireframe model

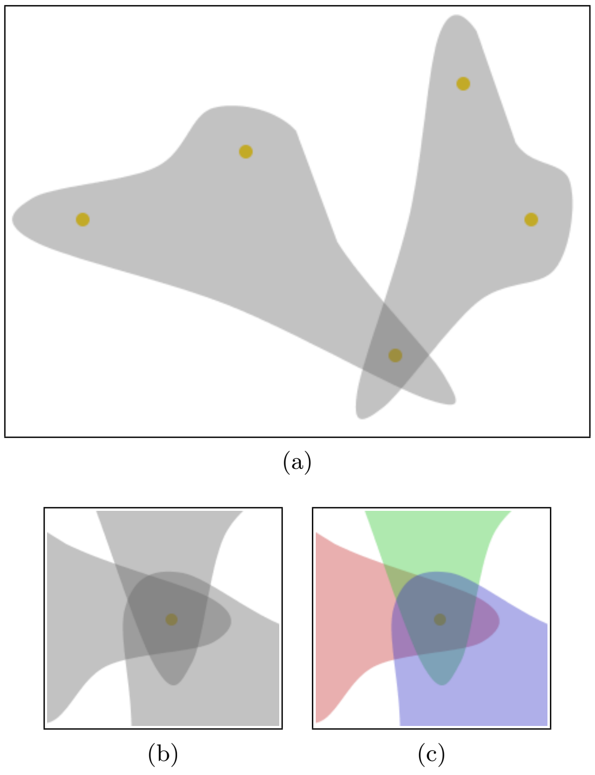


Figure G.23: Closed contour: Combining lists

Conclusions and future work

Our interaction with the prototype and observation of the brushing techniques show that they all have their strengths and weaknesses. Color works well in grouping the points and has the least interference with the CAD model, but does not scale well and also does not work for the combination of lists. Connectedness in general is a good way of showing membership, and order, but has the most interference with wireframe models. closed contour on the other hand works well with the wireframe model and also in combination with other contours, but does not scale and we still don't have a way of showing order in this technique.

In addition to our exploration of these techniques, we need to set up experiments in the future to test the techniques. In the experiment, we may ask participants to answer questions about membership, order and hierarchy in different situations. The prototype can be used to prepare the situations for these tests. It can also be used for interactive experiments in which participants make the lists themselves.

However, points are not the only members of the lists in CAD. Lines, curves, and other shapes can also be used in lists to create more complex objects such as solid bodies. Brushing these shapes may be even more challenging and needs to be investigated.

Another area for future research is visualizing lists in 3-dimensional models, as most CAD models are 3D objects and it is common for CAD users to interact with them in 3D views. Some of these techniques may not be feasible in 3D and some may be hard to use.

At the end, we can conclude that each technique has its own cons and pros and we have not yet found one that answers all our questions and solves all of the visualization issues in this project. Using a combination of these techniques may be the answer, in a way that each technique is used to its best purpose.