Programming in the Model: Contextualizing Computer Programming in CAD Models

Maryam M. Maleki, Robert F. Woodbury
School of Interactive Arts and Technology,
Simon Fraser University, Surrey, BC, Canada
(mmaleki,robw)@sfu.ca

Keywords: End-user programming, scripting, visual programming, direct manipulation

Abstract

Programming in the model locates programming elements and tasks contiguous with computer aided design (CAD) models. It aims to reduce the separation between acts of programming, modeling and design, using both spatial coincidence to reduce task shifting and common CAD techniques to simplify the expression of code. Using techniques from visual programming, parametric modeling and CAD selection we demonstrate how programming in the model can express the core steps in a very simple simulation algorithm.

1. INTRODUCTION

Designers use programming in computer aided design (CAD) for several reasons including exploring unconventional designs, reducing repetitive work and enabling change within the design process [Aish, 2003; Woodbury, 2010]. In recent years, such programming has become a normal part of design in innovative firms worldwide. Almost all CAD systems now provide extensive programming interfaces and such organizations as Smart Geometry conduct regular workshops and conferences that focus on programming within the design process. These designers are end-user programmers: they have little formal programming education and program as part of their job, that is to complete tasks [Woodbury, 2010]. Therefore they experience the same issues that other end-user programmers face in domains such as accounting in spreadsheets and analysis in structural engineering. One of these issues is the separation between programming and the task (design, analysis, etc.). In CAD, designers have to switch between the programming environment and the model view several times to make even the most simple modifications and evaluate their effects on the model, which results in a loss of focus and efficiency.

The literature emphasizes the importance of a close fit between the programming world and the domain to reduce novice and end-user programmers' cognitive load [Green and Petre, 1996; Pane and Myers, 1996], as well as to minimize or simplify the use of syntax and code to reduce non-programmers' difficulties in writing code [Kelleher and Pausch, 2005]. Direct manipulation of objects in the interface is a way of reducing the distance between what users think and what the system does and giving users a greater sense of engagement [Shneiderman, 1983; Hutchins et al., 1985]. Visual programming represents the program with visual elements and allows direct manipulation of the program in graphical form (e.g. Rhino's Grasshopper). However, the program is still separate from the model and in order to observe the result of any manipulation of the visual program, users need to go back and forth between the model and the visual programming environment.

2. PROGRAMMING IN THE MODEL

In this ongoing project, we propose *programming in the model* as a technique that uses designers' spatial and visual capabilities and their familiarity with the 2D/3D model and contextualizes programming concepts and subtasks directly in the model. In this approach to programming in CAD, object properties as well as dependencies between objects are represented in the model and can be accessed, modified and assigned by direct interaction in the model view. In addition, programming constructs such as functions and loops are created and modified in the model directly where they are needed and linked to the objects in the model to get their inputs [Maleki and Woodbury, 2010]. In this approach, we use both direct manipulation and visual programming. Contrary to visual programming where the program is represented visually in an independent window in the interface, programming in the model embeds the visual program in the 2D/3D model to reduce the need for designers to move their attention back and forth between multiple windows. Further, debugging is too often at yet another level removed from the design task. Effective programming in the model must include both programming and debugging.

The domain in which we work always presents a collection of objects as its model. Objects in the model are essentially global variables, even if namespaces are used. Programs tend to operate over a subset of the model. This makes practical a strategy of spatial localization of code near the subset so considered. We consider programs that (1) create, (2) change, and/or (3) report object properties (have side-effects). We aim for those constructs comprising a general programming language, i.e., constants, variables, expressions, statements (including control statements) and functions, as well as needed debugging facilities.

The product of programming in the model is a textual program, readable by machine and both readable and editable by people. The reason for this is that we envision programming in the model as only one of the productive modes of programming work. Program visualization and text provide complementary views and insights into code. Over fifty years of work on textual programming should not be ignored. Programming in the model complements, rather than replaces textual programming.

Textual programs are abstractions from the concrete domain over which they compute. In contrast, programming in the model is domain-proximate: it locates and demonstrates code working on specific objects, in this case within the CAD domain. A goal, therefore, is that programming in the model should provide a way for novice CAD programmers to progress from operations over specific objects to general functions that apply to a class.

As designers develop skill and as programs grow in size and complexity, there are tasks that will be more amenable to writing textual programs than to direct manipulation. Also, typical CAD systems provide multiple programming environments, each of greater capability and ending in a full development environment. Making textual programs available and editable at every level in a system can aid the process of laddering to more complex programming tasks and environments.

The current generation of new CAD systems are almost all parametric—they provide for the representation and maintenance of geometric and other data relationships within a model. In such systems programming both takes on a new role and become an inextricable aspect of work. No system can support the myriad of relationships that a designer might envision, so user-extension of these relationships is the norm. The current solution, provided by all systems, is programming. In a parametric system, writing small programs, distributed throughout a model is a normal aspect of the modeling process.

Following is a summary of some features of programming in the model. This is a work in progress; we present only a subset of the eventual features.

- Making lists: A designer makes lists of objects in the model using the list making feature and receives immediate visual feedback on the list in the model. The lists created by this method have the correct syntax based on system's programming language. The designer can modify the lists in the model by clicking on objects to be added to, removed from, or reordered within the list.

- Predefined operations and their inline customization: These are the operations that are applied to lists and individual objects. The system resolves indexing depending on the type of the operation and its operands.

- Implied indexing: The customization of the operations in a regular CAD system requires using the loops to access individual items of the lists. Programming in the model removes the loop by allowing implied indexing of the lists.

- Object properties and relationships: In parametric modeling, objects use other objects as inputs. These creates a network of dependencies among them. Some systems show these relationships independently from the model in a symbolic representation. Programming in the model also presents relationships in the model where the objects are. In addition, object properties are accessible in the model in short or extended modes and are editable.

We explain a sample of these features in the next section.

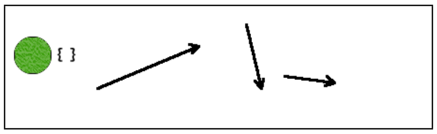
3. SIMULATION WITH PROGRAMMING IN THE MODEL

In this section, we demonstrate programming in the model through a very simple simulation system. Simulation is seldom provided in conventional CAD interfaces, yet designers often need to understand the effect of an environmental force over time or aim at design involving concepts of adaptation, in which parts of a design "respond" to other parts. For such goals, simulation is an almost-necessary tool.

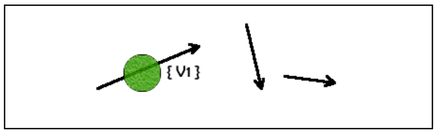
Beginning with the basic operations of vector addition, scaling and point subtraction (which themselves could be further decomposed into addition, subtraction and multiplication over real numbers), we develop all the logic needed for a simulation step. Our intention in providing an example that is both simple and complete is to show that programming in the model is a general concept, rather than a limited convention.

The model in the simulation example is a set of points called *force points* and a single point called the *target point.* The force points and the target point define a set of vectors. The simulation (explained at the end of this section) uses relaxation to move the target points so that the sum of these vectors is zero. We use a convention of variable replication [Aish and Woodbury, 2005] in which a variable of a particular type can carry either a collection or an individual object. While it would be usual in mathematical notation to distinguish between collections (using, for instance, uppercase letters) and objects (diacritics and/or lowercase letters), in the following we use single uppercase letters for all variables (P,Q,R,S), characters followed by a single number to distinguish objects destined for a collection ($P1,Q4,R3,S2$) and use the prime diacritic (P',Q',R',S') to distinguish objects clearly derived from earlier versions.

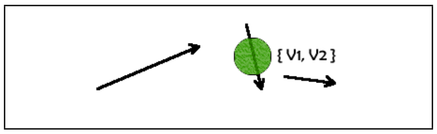
In this example, as in most parametric modeling tasks, we need to make lists of objects to use as inputs for functions or other objects. In programming in the model, we provide tools for users to make and use lists in the model and provide visual feedback in the model for each list. In addition, list objects take care of the specific syntax of the lists for the user (commas, brackets, etc.). Figure 1 shows a list of three vectors that is made by moving the list object in the model and clicking on the vectors to add them to the list. This list object then will be used as input to the operations performed on these vectors.



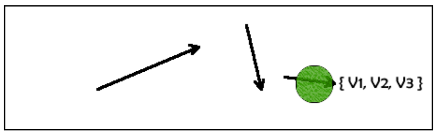
(a)



(b)



(c)



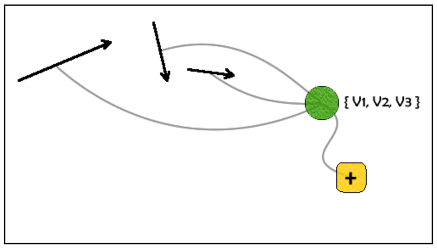
(d)

Figure 1. Making a list of three vectors.

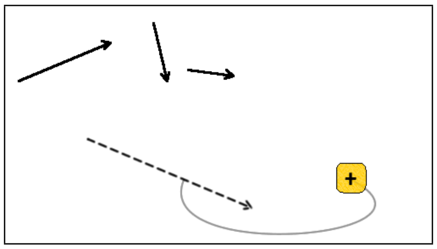
There is a set of predefined operations for each type of object in programming in the model. For vectors the operations are vector addition and negation (which yield vector subtraction), point-vector addition, point subtraction, and scaling. Using *replication* [Aish and Woodbury, 2005], each of these operations applies to lists as well as individual objects. In Figure 2 an addition operation is applied to the list of vectors by bringing the corresponding node to the model and linking the list to it. The result of this operation is a vector that is the sum of the three input vectors.

Let V be a list of vectors, and V' be a new vector, then $V' = +V$ means

```
for (int i = 0; i < V.count; i++)
{
  V' = V' + V[i]
}
```



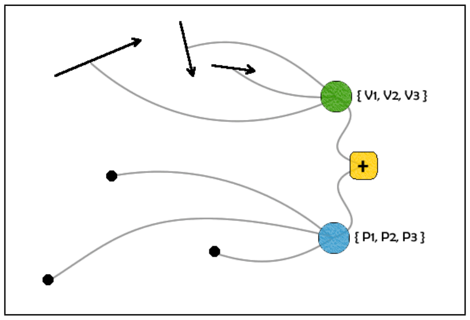
(a) Inputs.



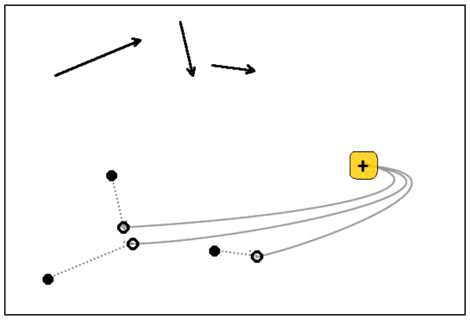
(b) Outputs.

Figure 2. Using addition operation on the vectors in a list.

Figure 3 shows the addition of a list of vectors to a list of points by making a list of points and a list of vectors and linking them to the addition node. The output of his node is a list of points that are the results of adding each vector to each point in the same order they appear in the lists.



(a) Inputs.



(b) Outputs.

Figure 3. Adding a list of vectors to a list of points using the addition operation.

Let V be a list of vectors, P be a list of points, and P' be a new list of points, then $P' = P + V$ means

```
for (int i = 0; i < min (P.count, V.
    count); i++)
{
  P′[i] = P[i] + V[i]
}
```

However, the user may want to add the vectors to the points in a different order. In most CAD systems, using a loop is the only way of doing such thing. Here, we eliminate the loop with *implied indexing* by allowing the user to define the order in a single line added to the addition operation using indexing. By doing that we relieve the user from having to deal with the syntax of a loop, as well as list counts and out of bound indices. The system forms and tests the needed conditional statements. Figure 4 shows a customized addition operation in which each member of the vector list with index i is added to the member of the point list with index $i+1$. The user can customize the operation by opening a text box and typing the desired operation.

Let V be a list of vectors, P be a list of points, and P' be a new list of points, then $P'[i] = P[i+1] + V[i]$ means

```
for (int i = 0; i < V.count; i++)
{
 if( (i+1) < P.count)
  {
   P′[i] = P[i+1] + V[i]
  }
}
```

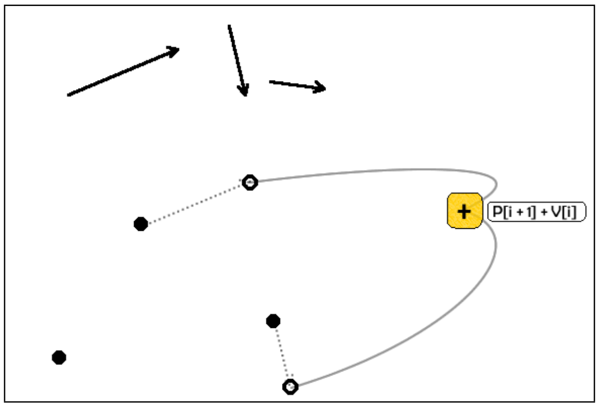


Figure 4. Addition operation is customized so that every $V[i]$ is added to $P[i+1]$.

This method of accessing list items without using a loop can be very helpful in simplifying a collection of loops and conditionals into a single line of code. Here is another example of its use.

Let L and K be lists of real numbers. Then $L[i] = K[i-1] + K[i] + K[i+1]$ is an inline operation in programming in the model that gives the user access to members in the K list by only typing a single line of code. However, in traditional programming languages, the following loop is required for such an operation.

```
for (int i = 1; i < K.count-1; i++)
{
  L[i-1] = K[i-1] + K[i] + K[i+1]
}
```

Notice that the loop starts from $i = 1$ instead of $i = 0$ and ends at $i < K.count - 1$ instead of $i < K.count$. Consequently, the sum of three members of K is assigned to $L[i-1]$ in order to start the list L from the 0^{th} member. All of these irregularities are sources of hard mental operations, confusion, and error for designers [Green and Petre, 1996].

Now we use these primitive operations to create the relaxation algorithm. A *relaxer node* is a composite node built of the above operations that applies the scaled sum of force vectors to the target point. The output of this node is a single point. First we make a list of the force points by using a list object. This list F, the target point P, and a scale factor S are inputs for the relaxer (Figure 5). As displayed in Figure 6, the first operation in the relaxer node is subtraction of the target node from each of the force points to make a list of vectors V. Then we add these vectors together to make a sum vector V'. We scale V' by multiplying it by S. At the end, we add this scaled vector to the target point to get the result point.

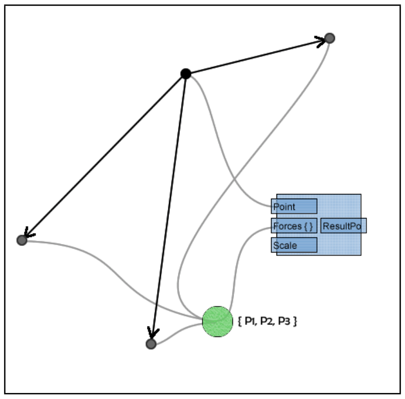
Let F be a list of force points, P be a target point, and V be a new list of vectors, then $V = F - P$ means

```
for (int i = 0; i < F.count; i++)
{
    V[i] = F[i] - P
}
```

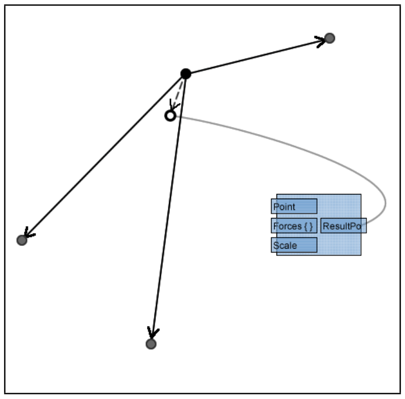
Let S be a scale factor and V' and V'' be new vectors, then $V' = +V$ and $V'' = S * V'$ mean

```
V′ = 0
for (int i = 0; i < V.count; i++)
{
    V′ = V′ + V[i]
}
V″ = S * V′
```

Let P' be a new point, then $P' = P + V''$, which is the addition of a vector to a point, creates the result point. Details of some of these operations are shown in Figure 7.



(a) Inputs.



(b) Output.

Figure 5. Feeding a list of force points and a target point to the relaxer node results in a new point made by addition of the scaled sum of the vectors to the target point.

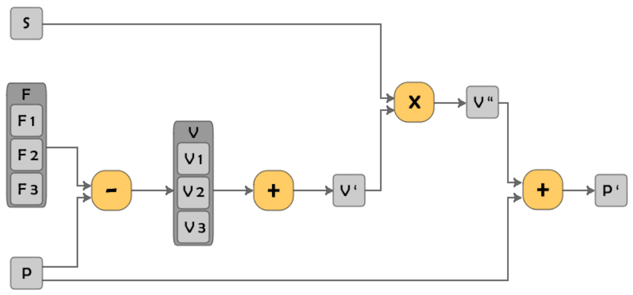


Figure 6. Inside a relaxer node.

The simulator is a node that is independent from the propagation graph and runs only once (Figure 8). The simulator takes the target point and the result point, moves the target point to the location of the result point to get the model closer to the final, balanced state. It then updates the graph. By doing that the relaxer algorithm is run and the result point moves to a new location based on the new state of the model. The internal loop in the simulator repeats these actions until the target point is coincident (within a tolerance) to the result point and the model is ”balanced.”

$$V = F - P \begin{cases} V1 = F1 - P \begin{cases} V1.X = F1.X - P.X \\ V1.Y = F1.Y - P.Y \\ V1.Z = F1.Z - P.Z \end{cases} \\ V2 = F2 - P \begin{cases} V2.X = F2.X - P.X \\ V2.Y = F2.Y - P.Y \\ V2.Z = F2.Z - P.Z \end{cases} \\ V3 = F3 - P \begin{cases} V3.X = F3.X - P.X \\ V3.Y = F3.Y - P.Y \\ V3.Z = F3.Z - P.Z \end{cases} \end{cases}$$

(a) Subtraction of the target point (P) from the list of force points (F) results in a list of vectors (V).

$$V' = +V \begin{cases} V'.X = V1.X + V2.X + V3.X \\ V'.Y = V1.Y + V2.Y + V3.Y \\ V'.Z = V1.Z + V2.Z + V3.Z \end{cases}$$

(b) Applying an addition operation on a list of vectors (V) results in a sum vector (V').

Figure 7. Details of subtraction and addition operations in the relaxer.

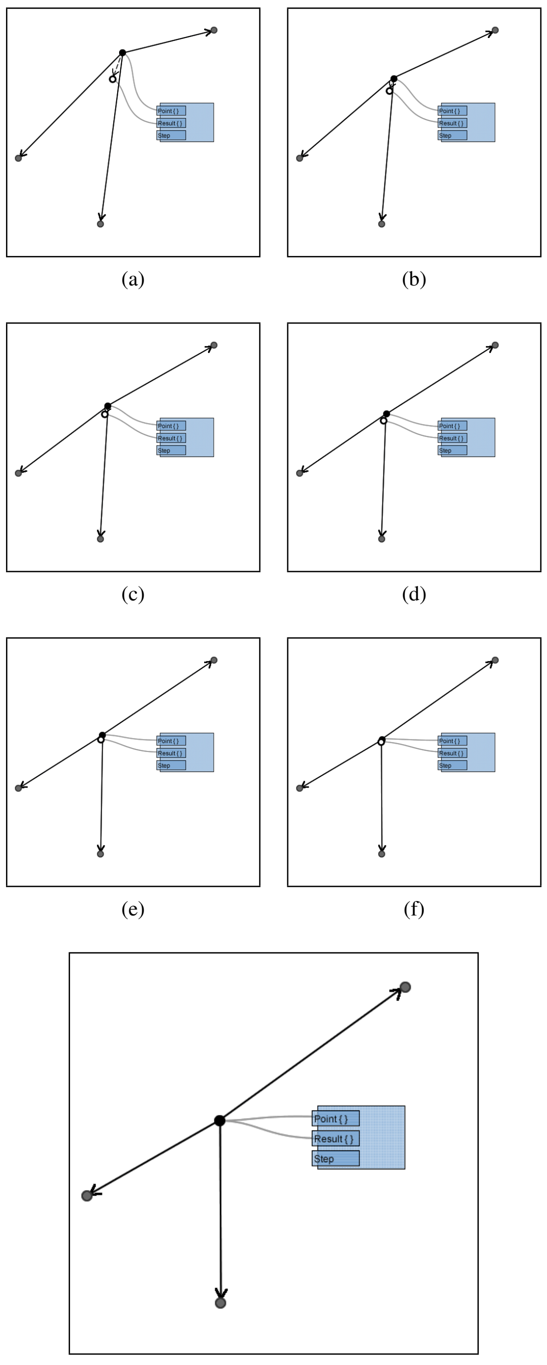
4. DISCUSSION

The simulation example above raises issues and questions that need to be addressed and provides an opportunity for future research.

An important aspect of programming in the model, inherited from the direct manipulation approach, is immediate visual feedback to user actions. This is important for the list making feature where the designer can benefit from having the list visually represented in the model by proper brushing and filtering methods. Modifying a list must immediately change its visual representation in the model. This brings up a challenge when the objects in the list are not simple points. Regular brushing techniques may not work for more complex objects such as surfaces and solids.

Representing programming elements in the model view can cause a cluttering problem, as shown in Figure 3, or may obscure the model all together. Transparency, layering, and on demand filtering can be useful but need to be tested in the context of design tasks.

Although inline indexing can eliminate the use of loops in simple customizations, it does not support every situation. For example, accessing the odd indices of a list requires every index to be checked against a condition. For situations where inline indexing is not enough and the use of a loop is required, how can we simplify the syntax and visualize it in the model for designers?



(a) (b) (c) (d) (e) (f)

(g) Final state of the model.

Figure 8. The simulator node takes the original target point and puts it in the location of the result point made by the relaxer node. This operation is repeated until the model is relaxed and the target point is in a balanced position.

The example of the relaxer node shows that a combination of a number of operations should be generalizable and abstracted into a node, so that with proper input, the desired output is achieved. This will lead to developing a method for representing functions and modules in programming in the model.

And more important of all, what do designers think about this? Does programming in the model help them program during the design task? Does it present too much interference? Is it easy to learn? Does it provide the means for learning the textual language? These questions remind us of the importance of user studies and user feedback sessions along the way.

5. CONCLUSIONS

This simple simulation example demonstrates two goals of programming in the model: simplifying the code, and bringing programming closer to the design.

Operations on some or all members of a list use loops (for, for each, and while loops) to access desired members. In programming in the model, we eliminate these loops by allowing direct access to members of a list through implied indexing as shown in Figure 4. The system takes care of the task of keeping the indices in bound that eliminates the use of conditionals (if statements) to check those indices. In doing so, we simplify the code for designers.

We hypothesize that representing object properties and relationships (visual programming) in the model and providing operations and inline customizations of those operations in the model (direct manipulation) may decrease the back and forth switching between design and programming environments, which in turn may allow the designers to focus their attention on the design task.

At the end, we need to emphasize that programming in the model is not intended to replace textual programming. Rather, the two work side by side and complement one another. Users can choose the form of programming based on their task and their programming experience and preferences.

This is a work in progress. We are currently working on other programming constructs such as functions and loops (when implied indexing is not enough) and how to present them to designers in the context of the model; as well as debugging. We are making working prototypes for future user testing of programming in the model, the results of which will be presented as they become available.

ACKNOWLEDGMENTS

This research is partially supported by the Canadian NSERC Discovery Grants program; Bentley Systems, Inc.; the NSERC Collaborative Research and Development program; and the MITACS Accelerate Canada program. This support is gratefully acknowledged.

REFERENCES

Aish, R. (2003). Extensible computational design tools for exploratory architecture. In *Architecture in the digital age: design and manufacturing*, pages 244–252. Spon Press, New York, NY.

Aish, R. and Woodbury, R. (2005). Multi-level interaction in parametric design. In *Lecture Notes in Computer Science: Smart Graphics*, pages 151–162. Springer Berlin / Heidelberg.

Green, T. R. G. and Petre, M. (1996). Usability analysis of visual programming environments: A 'Cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174.

Hutchins, E. L., Hollan, J. D., and Norman, D. A. (1985). Direct manipulation interfaces. *Human-Computer Interaction*, 1(4):311.

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

Maleki, M. M. and Woodbury, R. F. (2010). Programming in the model: Combining task and tool in computer-aided design. In *New Frontiers, Proceedings of CAADRIA 2010, Hong Kong*. Forthcoming.

Pane, J. F. and Myers, B. A. (1996). Usability issues in the design of novice programming systems. Technical Report CMU-CS-96-132, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA.

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

Woodbury, R. F. (2010). *Elements of Parametric Design*. Taylor and Francis. Forthcoming.