Improving Communication In Programming-by-Demonstration

Richard G. McDaniel
HCI Institute, School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213 USA
+1-412-268-3066
richm@cs.cmu.edu

ABSTRACT
The range of PBD systems can be significantly expanded by improving the user's expressiveness when communicating with the system. The techniques in my research include a new form of demonstrational interaction called *nudges*. Complementing nudges is a special form of selection which is used to give the system *hints* by identifying significant objects. A new *deck-of-playing-cards* metaphor is also introduced for specifying useful effects such as randomness and sequencing. The final techniques use objects for annotating examples such as *behavior icons* for manipulating and editing behaviors, and *temporal ghosts* to allow explicit references to past states. By fostering better communication between the author and the system, these techniques should allow the user with minimum programming expertise to create highly interactive software.

Keywords
User interface software, application builders, programming-by-demonstration, programming-by-example, inductive learning

INTRODUCTION

Creating interactive software such as games and educational software can be an onerous programming task even when using state-of-the-art user interface tools. Requiring a software author to use a textual programming language on interface-driven software like video and board games seems unnecessary. Furthermore, potential authors who are able to draw pictures or write music but cannot program are prevented from using their talents for making software.

Researchers have applied programming-by-demonstration (PBD) techniques in order to reduce the amount of programming required for interactive software. A PBD system uses inductive learning to generate a program from example demonstrations. To produce a program, the author draws the various interface components and then demonstrates their behavior by mimicking the desired program response to various events. The system records and analyzes the demonstrations, converting the low-level demonstrated actions into a higher-level description of the program behavior.

The goal of my research is to invent and test techniques that allow an author to express the semantic relationships that affect behavior. Direct manipulation techniques to annotate and manipulate examples will give an author the means to point out interesting objects and make important abstractions explicit so that the system can infer a broader and more useful range of behavior. By using these techniques, an author can create a wider variety of software than can be produced using current PBD systems.

The techniques are being incorporated into a new software tool called Gamut with which an author will be able create a variety of complete, interactive games. These games will be in the style of two-dimensional board games and include computerized versions of Monopoly and Chess as well as computer games and educational software like Number Munchers, PacMan, and Playroom. The board game domain provides a number of challenges:

- There are many conditions and modes. This requires the system to find and track the variables that control when operations happen.
- An object's history factors into its context. To determine an object's direction of travel or last legal move requires that the past state of the object be accessible.
- The number of objects with certain characteristics can affect the behavior. Keeping score and determining game phases often requires that the system be able to count things.

TECHNIQUES

Gamut defines new forms of interactions including *hints*, *nudges*, a *deck-of-playing-cards* metaphor, *guide objects*, *behavior icons*, *temporal ghosts,* a *timeline* view, and a *counting sheet*.

Hints and Nudges

Authors demonstrate system behavior by running the game. When the author first performs an event, the system does not respond. The author then corrects the system by switching into response mode and modifying the objects as they are supposed to be affected by the program. In essence, the author *nudges* the system to perform the correct behavior. If the system continues to perform incorrect behavior, new examples are demonstrated by nudging the system again.

The system defines two nudges for certain situations. First, the *Do Something!* nudge is used when a behavior has been demonstrated to the system, but an object does not respond. When the unresponsive objects are selected and

Do Something! is pressed, the system will search for a previously learned behavior that can be generalized to include the selected objects. Second, the *Stop That!* nudge is used when a behavior performs undesired actions on objects. The system compares the current state to previous times when the behavior succeeded and determines how to make the behavior conditional.

To point out important objects that relate indirectly to the demonstrated behavior, the author will use highlighting for hints. Hints are especially useful when the author performs a *Stop That!* nudge. Objects can be highlighted even if the reason for the hint will not be apparent until later. Maulsby studied hints in his thesis as well [2], but Gamut will be the first to apply hints in an interactive domain.

Deck of Cards

An author will be able to use the deck of playing cards metaphor to specify a variety of effects. Cards can be used to perform random events like the Chance deck in Monopoly or to move pieces as in PacMan. First, the author builds an appropriate deck by assigning graphics and properties to each card. The author can then demonstrate how to use the deck such as when to shuffle or where to play a card as the game progresses. By having the system play cards autonomously, objects like video game monsters can be made to move on their own, as shown in Figure 1.

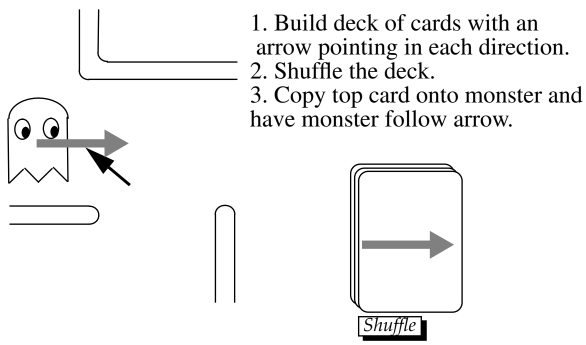

Figure 1: Using Cards to Move a Monster

Guide Objects

Annotation objects are objects that an author uses during editing to build or manipulate relationships and behaviors. They appear as visible graphics while the program is edited but disappear when it is run. *Guide objects*, *behavior icons*, and *temporal ghosts* are all annotation objects in Gamut.

Lines and rectangles can be used as annotations to show connections and graphical constraints between objects. Demo II by Fisher *et al*.[1] used a similar technique calling the objects *guidewires*. By placing the guide objects in important locations, positions and relationships can be maintained even when no visible objects are involved. For instance, one can configure a table for a card game by laying out rectangles at places where cards are placed.

Behavior Icons

Gamut represents behaviors with small icons placed near the objects they affect. These icons can be selected and highlighted just like any other object. By using cut, copy, and paste operations, the author can transfer a behavior from one object to another. The system will infer how the behavior must change to work in the different context.

Temporal Ghosts

Many rules in board games rely on the prior state of objects. For instance, a piece's legal moves may be relative to its current position. To allow the author to make explicit connections to the past, Gamut will show a dimmed, translucent image of the object in its prior state. A similar technique was used to record cursor positions in Marquise [3] so that the author could specify how objects behave with respect to cursor events.

Timeline and Counting Sheet Views

The author will be able to control the number of ghost objects as well as run the application back and forth in time with the timeline view. In the timeline, events and actions are displayed, permitting the author to highlight and manipulate them.

To keep score or keep track of other numbers, the author will use the counting sheet. Like a spreadsheet for data descriptions, the counting sheet will describe various numeric properties about the game. The numbers can be used as events to initiate other behaviors such as switching to a new round or ending the game.

INFERENCING

Gamut's inferencing is most like the artificial intelligence field called plan-recognition. As in Gamut, a plan recognition system watches a sequence of actions and labels the sequence with a higher-level description that explains the reason why the actions occurred. Since plan-recognition already exists as a field, it is likely that Gamut's inferencing algorithm can borrow from existing systems.

Though good inferencing is crucial for a system like Gamut to be successful, the ability for the author to communicate effectively will reduce the amount of guessing the system must perform, making its inferences more accurate. Similarly, effective communication will make the system's incorrect guesses easier to fix.

CONCLUSION

By designing techniques to foster better communication between author and system, it will be possible to make more powerful application behavior with fewer examples. Having these techniques in a single tool will provide authors with a powerful environment for building interactive software like games and educational software while requiring minimal programming expertise.

REFERENCES

1. Gene L. Fisher, Dale E. Busse, David A. Wolber. Adding Rule-Based Reasoning to a Demonstrational Interface Builder. *Proceedings of UIST'92*, pp 89-97.
2. David Maulsby. *Instructible Agents*. Ph.D. thesis. Department of Computer Science, University of Calgary, Calgary, Alberta, June 1994.
3. Brad A. Myers, Richard G. McDaniel, and David S. Kosbie. Marquise: Creating Complete User Interfaces by Demonstration. *Proceeding of INTERCHI'93: Human Factors in Computing Systems*, 1993, pp. 293-300.