Using Better Communication To Improve Programming-by-Demonstration

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

Abstract

My research focuses on how a user communicates with a programming-by-demonstration (PBD) system. In particular, I am experimenting with new nonverbal, direct manipulation techniques that will enhance a user's expressiveness and subsequently will make it possible to infer a broader range of application behavior. The techniques include a new form of demonstrational interaction called *nudges* used to specify behavior. 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. Other 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. Finally, using *guide objects* is a technique for demonstrating constraints and hidden connections between objects. Complementing these new techniques will be an inferencing algorithm sufficiently powerful to convert data from all these sources into application behavior. Special attention will be given toward how hints are used to regulate inferencing. By fostering better communication between the user and the system, these techniques should allow the user to create highly interactive software with minimal programming expertise.

Introduction

Creating interactive applications 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.

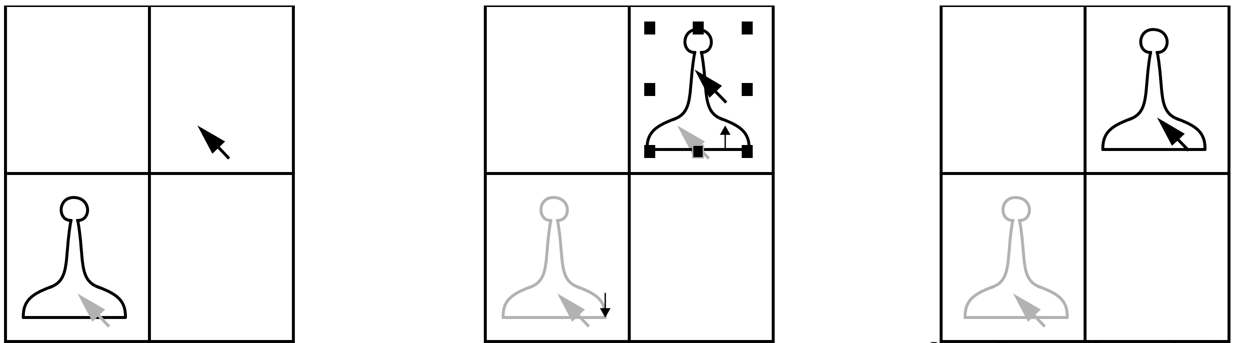
Researchers have applied programming-by-demonstration (PBD) techniques to reduce the amount of programming required for interactive software. In a PBD system, an author draws the interface components and demonstrates their behavior by mimicking the desired program responses to various events. The system records and analyzes the demonstrations using inductive learning to produce a program that performs the desired behavior. Since an author can only demonstrate what he or she has ability to express in the PBD interface, communication between the author and the system must be expressive and powerful.

The goal of my research is to invent and test techniques that allow an author to express semantic relationships that affect application behavior and provide inferencing capable of using this data. 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 using fewer examples. By using these techniques, an author can create a wider variety of software than can be produced using current PBD systems.

I will incorporate the techniques 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 such as computerized versions of Monopoly and Chess as well as computer games and educational software like Number Munchers, PacMan, and Playroom. Developing board games provides a number of challenges for a PBD tool:

- There are a large number of conditions and modes. This requires that the system find and track the variables that control when operations happen.
- An object's history is an important part of its context. To determine an object's direction of travel or next legal move often requires that the past state of the object be accessible and manipulable.
- 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.

Gamut's role will be to create applications that referee the interactions among human players. Gamut will not necessarily be able to play the game, itself. Expressible behaviors include those that control player interaction as well as perform autonomous actions such as controlling monsters and other animated behaviors. Naturally, not all behaviors will be expressible in Gamut. Behaviors that use special forms of interface interaction like 3-D rendering and behaviors that require the system to infer complex equations such as physics simulations will not be supported. Gamut will place an emphasis on inferring the conditions that cause behavior to act differently in different contexts. When an object changes



Author wants a piece to move when dragged. While program is running, author clicks down on piece, drags, and lifts button at destination.

Nothing happens, so author switches to Response Mode and "nudges" Gamut by dragging the piece manually. (Grayed objects are temporal ghosts.)

Later, Gamut will move the piece automatically.

Figure 1: Using Nudges To Demonstrate An Interaction

state, the author will be able to instruct the system why the behavior has occurred as well as what the behavior does.

Techniques

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

Nudges and Hints

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. An example of nudging is shown in Figure 1.

The system defines two nudges for certain situations. First, the *Do Something!* nudge will be used when a behavior has been demonstrated to the system, but some objects do 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 will be used when a behavior performs undesired actions on objects. The system compares the current state to previous times when the behavior succeeded and determine how to make the behavior conditional.

Several prior systems such as Frank's Grizzly Bear [2] have used an extended macro-recorder metaphor for collecting demonstrations. A macro-recorder stores operations by having the author transfer into a special mode where the system will remember each action the author performs. The extended macro recorder adds a second mode to this scheme for recording stimulus events. First, the author switches on stimulus mode to demonstrate the event, then he or she switches to response mode to demonstrate the actions. This makes creating a behavior a two-step process, both phases requiring the author to anticipate and set up the environment so that the demonstration will proceed without a mistake.

The purpose of nudges is to simplify the way that an author specifies behavior. Instead of demonstrating stimulus and response separately, the author only shows responses. The stimulus is inferred by observing what events the author performed most recently before demonstrating a response. Also, by using *Stop That!*, the author generates negative examples implicitly. The macro-recorder technique requires the author to indicate first that a negative example is about to be shown, then the author must purposely force the system into a state in which it is not allowed to be. This requires that the author know what situations the system has guessed incorrectly before the system has shown any problems. Using nudges, the author must only identify that the system made a mistake after it has happened. Correcting the mistake can be accomplished immediately and does not require the author to formulate a new demonstration from scratch.

To point out important objects that relate indirectly to the demonstrated behavior, the author will use highlighting for hints. Maulsby's Turvy experiments studied hints as well [4], but Gamut will be the first to apply hints in an interactive domain. Hints are needed to distinguish the causes for conditional behavior. Even in a small application, the number of possible conditions that could affect a given behavior are too numerous to generate and test. By allowing the author to highlight objects that are significant to the behavior, a system can narrow the number of choices to arrive at a reasonable guess. In Figure 1, the author highlights the two mouse pointer ghost objects (described later) to note that the cursor affects which object is moved and where it goes. Hints are especially useful when the author performs a *Stop That!* nudge because *Stop That!* indicates that a behavior is conditional. Objects can be highlighted during any demonstration even if the reason for the hint will not become apparent until later.

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 2.

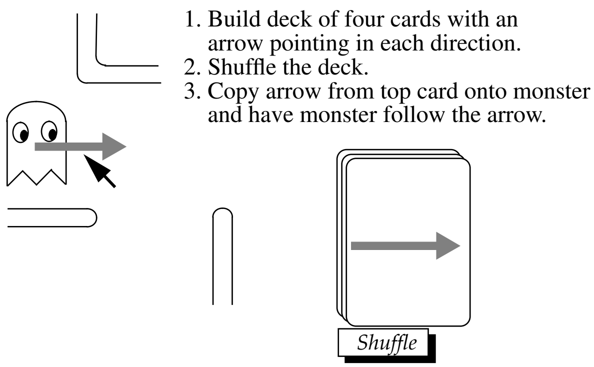


Figure 2: Using Cards to Move a Monster

Playing cards can be used to perform many common programming algorithms that use a list of elements data structure such as stacks, queues, and dictionaries. By presenting this data structure as a deck-of-cards, its operations become more concrete. The metaphor also helps the author understand and control aggregate operations such as sorting, searching, and aggregating properties. For instance, one might sum the `VALUE` property of each card in a player's hand to determine a final score.

Annotation 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, mode switches, behavior icons, and temporal ghosts will all be annotation objects in Gamut.

Guide Objects

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 called *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. Lines and arrows can be used as directions and paths for moving objects as shown in Figure 2.

Mode Switches

Mode switches allow the author to easily specify different application modes. For instance, a mode switch can be used to control which player's turn is current. To build Pacman, a mode switch can be used to control changes between the normal mode when monsters are dangerous and the mode where monsters turn blue and can be eaten by the Pacman.

To build modal behavior, the author first creates a mode switch and specifies how many states the switch will have. To change the mode, the author switches the mode switch and demonstrates how the application objects are affected. For instance, in Pacman, when the mode switch is set to blue mode, the author colors the monsters blue, deletes the monsters' "run towards" behavior, and replaces it with a "run away" behavior.

Behavior Icons

Gamut will represent 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. The icons may also be selected and expanded to present extra information about their behavior. This can be used to view a textual representation or to modify the behavior's parameters.

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 as in Figure 1. A similar technique was used to record cursor positions in Marquise [6] so that the author could specify how objects behave with respect to cursor events. Gamut generalizes this to all objects.

Timeline and Counting Sheet Views

The author will be able to control the number of ghost objects as well as make the application run back and forth in time with the timeline view. In the timeline, events and actions are displayed graphically, 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 will be able to be used as events to initiate other behaviors such as switching to a new round or ending the game.

Feedback And Representations

A major concern for PBD systems is how should it tell the user what it has inferred in an understandable and concise way. Some systems in the past have provided no feedback at all. Demonstrations would be inferred without the author knowing what would happen until the program was tested later. Other systems provide a secondary window that displays the system's inferences as programming code. Graphical representations have been shown to be one of the most understandable forms of feedback for novice users [5]. This is why Gamut avoids using language representations as much as possible and uses graphical representations for the majority of its techniques.

Gamut uses a variety of feedback methods. Behavior has many facets and different views are needed for each. One basic requirement is that all data in a Gamut program be completely visible and accessible. This same principle was embodied in Gould and Finzer's Rehearsal World [3] with some success and has since been common in many PBD environments. As in Rehearsal World, Gamut's data will be exposed as widgets. Some of the widgets are grouped together such as the counting sheet where multiple number-based widgets are collected in one place, and the deck-of-cards which collects multiple card widgets. All data is exposed so that the author can see the state of the program at a glance and see potential problems as soon as possible. Changes in the background data do not always affect the visible screen objects immediately which leads to confusion later when the screen objects do misbehave. By making the data visible, some errors can be corrected sooner and more directly because the author can point directly to the errant data before it causes problems.

Just as important as being able to see the data is being able to examine the behaviors that affect the data. This is an area of active research and one of the areas where Gamut's new techniques are used. Each kind of annotation object can be considered a form of graphical feedback in Gamut. Guide objects can show what path an object will follow or how objects are interrelated. Behavior icons are a graphical representation for behavior and make it possible to see which objects have associated behaviors and which do not. Temporal ghosts show the relationship between the previous and current state of the program. And mode switches are a graphical representation for sources of conditions.

The graphical annotations in Gamut are meant to suggest to the author how objects will respond to behaviors. Unlike a language-based technique, the annotations are not precise in that they are not necessarily self-describing. Seeing a guide line connecting two objects in a program could mean potentially anything to someone who is viewing it for the first time. Often, an annotation's use becomes apparent when the program is run, but not always. One solution is to provide a language that describes what a behavior will do. This language can be viewed upon request so that nonprogrammers need not look at it if not desired. One potential problem arises, though, if Gamut allows the author to modify a behavior using this language. Originally, the code would have been generated by Gamut making it feasible for the system to reason about the behavior and change it when necessary. If the author hand modifies the code too much or in a unexpected way, it can become too difficult for Gamut to reason with the code any longer.

Feedback must also be provided to tell the author what will happen when a behavior is run. The temporal ghosts show the previous state and the timeline view provides a fine-grained history of past actions and events. The author can see precisely what has happened by reading the actions on the timeline. Though the timeline is essentially a textual form of feedback, it is stripped of all procedural code such as conditions and loops so that it can be read as a simple list of actions. The timeline allows the author to replay events and is the chief means for controlling program execution. Common debugging operations such as single stepping are also provided by the timeline by replaying only some of the actions at a time.

Implementation

Converting the data expressed using Gamut's techniques into a usable application is the job of the inferencing system. Inferencing involves two components: the knowledge representation which dictates the form in which inferences will be generated and the inferencing algorithm which converts the demonstrations into the knowledge representation.

Knowledge Representation

There are two forms of knowledge that will be used in Gamut. The first form represents the learned game behavior. It consists of programming constructs such as operations, loops, and conditions which form the structure which causes the game to respond to the user. The kinds of operations Gamut must support include: creating and deleting objects, moving objects, and changing object properties.

The parameters for Gamut's operations are data descriptions which describe which objects the behavior affects. Gamut's data descriptions must be able incorporate many factors and be able to refer to objects indirectly. For instance in Monopoly, players roll dice to determine the next position for their pieces. The description for the next position might be "the square that is the dice roll's number of spaces away from the square where the current player's piece started." This description uses information from several sources including the dice, the board, who the current player is, and the current location of a piece. Furthermore, much of the information is described relative the other factors.

The second form of knowledge is the permanent store of rules that specify how board game elements are assembled. These rules are used to guess relationships between objects such as how objects are positioned or how to interpret a change between players. This knowledge is stored in the form of algorithms and methods associated with the various objects and property types that the author manipulates. For instance, shuffling algorithms are attached to deck-of-playing-cards objects, path algorithms are attached to guide objects, and number heuristics are attached to numeric properties. These algorithms and methods are invoked while the author manipulates data and demonstrates behavior to provide a source of information for the inferencing algorithm.

Inferencing Algorithm

Gamut's inferencing can be best likened to the artificial intelligence field called plan recognition [7]. A plan recognition system watches a sequence of actions which is normally provided by a human source. The goal of the system is to label the sequence with a higher-level description that captures the meaning of the actions and explains the reason why the actions occurred. Gamut watches actions performed in a graphical editor and analyzes the sequence as actions that would be performed in a game. At present, I do not know exactly which algorithms will work best for Gamut. Since this workshop will host a number of AI researchers, it will provide an opportunity to learn more about inferencing and discuss better algorithms for my own research.

Critical to Gamut's inferencing will be the ability to search prior behaviors for various attributes. When the author asks the system to *Do Something!*, the system will need to search for behaviors which use the same events as the one just demonstrated. Likewise, when a behavior icon is moved to a new object, the structure of the associated behavior will have to be searched for its relationship to the new object. Joining the branches of a conditional behavior requires that the two sequences be compared to find the point where they perform identical operations.

User Interface Experiment

Gamut is a relatively large system for a research project. This mandates that the first step be to test these ideas early before needless work is performed implementing techniques that users will not like or will not understand. Therefore, I will design an informal, paper prototype experiment to test Gamut's main concepts and techniques. In a paper prototype, one builds a mock-up of the computer interface using paper and cardboard cutouts. These pieces will be used to act out Gamut's techniques such as nudges, hints, deck-of-playing-cards, the timeline, and a limited set of annotation objects. Since the system must be acted out manually, behavior icons, the counting sheet, and mode switches will be left out to make the experiment easier to run.

For the experiment, subjects will be expected to demonstrate a series of behaviors. Some of the behaviors will require a degree of creativity on the subjects' behalf. If the techniques work well, the subjects should never be stymied by any of the problems, though they will likely have to think through a few problems. The goal is to see which techniques the subject understands best and how the subject uses them to solve a problem.

For the initial runs, subjects will be allowed to design their own widgets and interaction methods. Though this may just lead to fanciful ideas that cannot be implemented, I expect that allowing this will produce some insight into how users understand the problem. Because the experiment is informal, I will be able to modify the experiment between sessions to try out different ideas more quickly. The results from this experiment should be ready in time for the workshop presentation.

Conclusion

By designing techniques to foster better communication between author and system, it will be possible to make more powerful inferences with fewer examples. Part of the research will investigate how the techniques affect an author's perception of a programming task. Also, since this research uses a variety of techniques, it will be possible to investigate how authors learn and use Gamut's techniques as well as how authors conceptualize game behaviors in Gamut's framework. Having these techniques in a single tool will aid the investigation as well as provide authors with a powerful environment for building interactive applications 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] Martin Frank. *Model-Based User Interface Design by Demonstration and by Interview.* Ph.D. thesis. Graphics, Visualization & Usability Center, Georgia Institute of Technology, Atlanta, Georgia.

[3] Gene L. Fisher and William Finzer. *Programming by Rehearsal*. Palo Alto Research Center, Xerox Corporation, 1984.

[4] David Maulsby. *Instructible Agents*. Ph.D. thesis. Department of Computer Science, University of Calgary, Calgary, Alberta, June 1994.

[5] Francesmary Mudugno, T.R.G. Green, Brad A. Myers. Visual Programming in a Visual Domain: A Case Study of Cognitive Dimension. *Proceedings Human-Computer Interaction 1994, People and Computers*, Glasgow, August, 1994.

[6] 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.

[7] Stuart J. Russell, Peter Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1995. p. 654.