Improving Demonstration Using Better Interaction Techniques

Richard G. McDaniel and Brad A. Myers
14 January 1997
CMU-CS-97-103

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also appears as Human-Computer Interaction Institute Technical Report
CMU-HCII-97-100

Abstract

Programming-by-demonstration (PBD) can be used to create tools and methods that eliminate the need to learn difficult computer languages. Gamut is a new PBD tool that can create a broader range of interactive software, including games, simulations, and educational software, than other PBD tools. To do this, Gamut uses advanced interaction techniques that make it easier for a software author to express all needed aspects of one's program. These techniques include a simplified way to demonstrate new examples, called "nudges," and a way to highlight objects to show they are important. Also, Gamut includes new objects and metaphors like the deck-of-cards metaphor for demonstrating collections of objects and randomness, guide objects for drawing relationships that the system would find too difficult to guess, and temporal ghosts which simplify showing relationships with the recent past.

This research was partially sponsored by NCCOSC under Contract No. N66001-94-C-6037, ARPA Order No. B326, and partially by NSF under grant number IRI-9319969. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC, ARPA, NSF, or the U.S. Government.

Keywords: Programming-by-demonstration, inductive learning, programming-by-example, application builders.

1. Introduction

Gamut is a innovative tool for building interactive software like games, simulations, and educational software. Much of the effort involved in producing a game or an educational tool is not in programming the game's logic but in providing the engaging background, artwork, and gameplay that keeps players interested. Artists and educators with the talent and ideas to produce good material are often unable to program computers. Thus, tools which eliminate the burden of programming while maintaining flexibility are desirable.

Traditional development tools for producing interactive software require extensive programming knowledge. Programming graphics in common environments like Visual C++ or MacApp can be difficult for even seasoned programmers. Tools such as interface builders can help software authors design the visual appearance of an application but still require programming to make the interface actually work. Application builders such as Klik & Play [9] eliminate programming but impose severe limits on the kinds of programs that can be created. Authoring tools like AuthorWare [1] or Director [10] are similarly limited and cannot easily produce complex behavior and player interactions.

One method for simplifying the programming process has been programming-by-demonstration (PBD). Instead of using a textual notation, the software author builds the program by providing examples of the intended interactions between the user and the application. Examples are demonstrated using the same interface normally used to create and manipulate the application's data. The system uses the examples to infer the author's intention and assembles code to execute the program.

Our research is aimed at significantly improving and expanding what can be accomplished using PBD. Gamut has the ability to infer complex relationships that other PBD cannot through the use of improved interaction techniques. The game author is able to give Gamut more information more easily than is possible in other systems without requiring the author to learn programming concepts. These interaction techniques provide several benefits:

- A simplified method for producing examples.
- An understandable way to create negative examples.
- The ability to give the system specific and direct hints.
- Objects and metaphors that can describe complex behaviors concisely.

2. Domain

Gamut can make games and simulations similar to board games. These are two-dimensional with a board-like background and playing pieces that represent the game state. The domain extends well beyond Chess and Monopoly, however. By having objects react autonomously and by adding player interaction, one can create video game behaviors such as moving monsters and shooting aliens. Educational games like Reader Rabbit [17] and Playroom [5] and video games like PacMan can all be made in Gamut.

The board game domain provides new challenges for a PBD system:

- The interface is interactive and needs to be updated continuously. Other PBD systems assume that the inferred program manipulates static data with little or no user input.
- Board games have a large number of states and modes. Game behavior can be triggered by a variety of

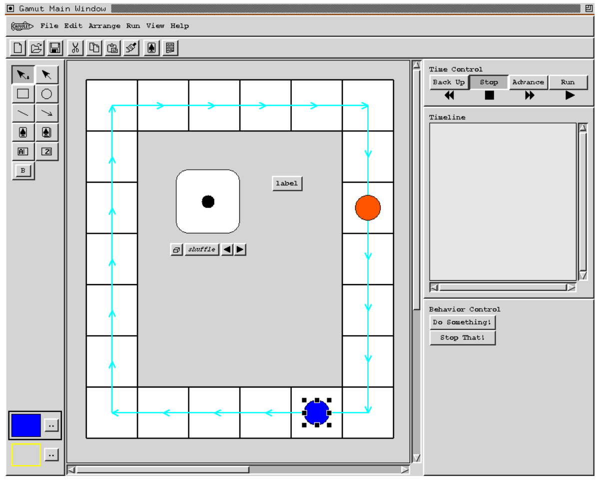


Figure 1: Gamut's Main Window

events and can have complicated relationships.

- Relationships between objects and actions are often formed as long chains of other relationships which build upon each other.

For example, the destination square where a piece is moved in Monopoly could be described as "the square that is the dice's number of squares away from the square where current player's piece currently resides." This description depends on the configuration of the board, the number on the dice, and the player whose turn it is. Each object in the relation forms a link in the chain. Furthermore, an object like the turn indicator is not necessarily graphically or temporally connected to the other objects. Current PBD systems cannot infer this form of relationship.

Gamut can be taught the rules of a game, but generally cannot create computer opponents. The difference here is the difference between rules and strategy. Playing a complex game well requires strategy which is often not easily encoded as a set of rules. Gamut is designed to assemble games for humans to play, not to play the games, itself. The game author has to show the system all relationships upon which a behavior depends.

3. Related Work

A number of tools make building games easier. Most construct a specific class of games such as Bill Budge's Pinball Construction Set [2] which makes pinball simulations. A recent product is Klik & Play created by Lionet and Lamoureux [9]. In Klik & Play, the game author first draws the game objects and classifies each as background, characters, or other objects. Then the author assigns behavior to the characters by picking from a list of stock behaviors. These behaviors can be customized by changing some parameters, but in general, the author is limited to built-in methods for user interaction and game play.

Gamut most resembles Marquise [15]. Like Gamut, Marquise's goal was to create whole applications. Marquise had the ability to recognize palettes of objects and could quickly infer operations such as selecting and dragging. Marquise's major deficiency was an inability to correct guesses by demonstration. The only method for correction was editing the inferred code using a set of unwieldy dialog boxes.

PBD systems such as Wolber's Pavlov[20] and Frank's Grizzly Bear [4] have shown that simple heuristics can be used to infer many forms of graphical constraints and simple behaviors. Both of these systems infer linear relationships between objects with numeric parameters (like an object's screen position).

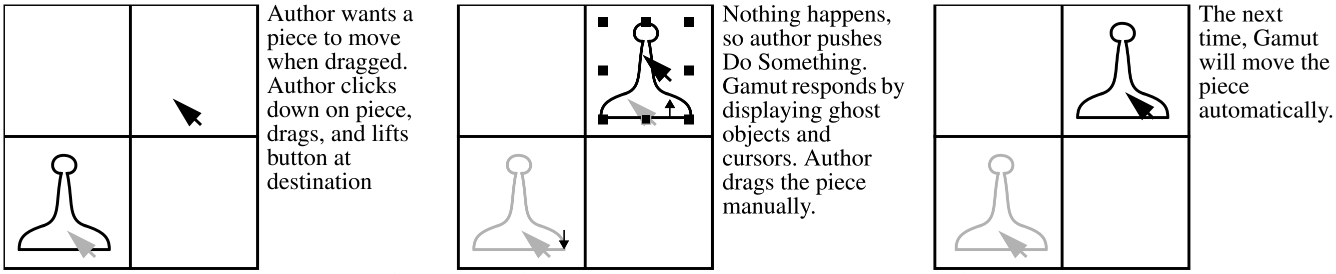


Figure 2: Using Nudges To Demonstrate A Behavior

Unfortunately, these heuristic methods start to fail when faced with the task of building a whole application. In particular, descriptions that are hierarchical, nonnumerical, or nonlinear cannot be inferred in these systems. Pavlov requires users to annotate their demonstrated behaviors with conditional guard statements in order to overcome these problems.

Gamut's inferencing ability has many factors in common with Maulsby's Cima system [12]. Cima also has the ability to learn from hints and can learn concepts incrementally. Cima's description language is not as powerful as Gamut's. Cima's statements are restricted to logic statements in disjunctive normal form (DNF) which it uses to recognize passages in a body of text. Cima currently cannot do work on behalf of the user: it just recognizes strings of text. It is unclear what the interface to Cima will be like as it was still in a prototype stage at the time of this writing.

Gamut and Cima use inductive learning techniques to learn from examples. In inductive learning, algorithms extract the salient features from the examples and generalize them to form rules. Gamut uses two forms of induction taken from AI literature. The first is plan recognition [19] which Gamut uses to determine the differences between new examples and previously learned behaviors. Plan recognition takes a chain of events and maps a higher level description upon it that explains why each event occurred. Gamut uses this to infer where the software author means to create new actions, conditions, and loops. The second form of induction is decision trees, which Gamut uses to learn the complex relationships between modes. A decision tree algorithm like ID3 [16] uses statistical measurements to map the values of a set of attributes into a domain of concept names. In Gamut, the attributes are predicates on the state of the objects in the game, and the concepts are branches in a conditional statement.

4. Interaction Techniques

Gamut's interaction techniques make it possible to demonstrate not only the surface activity of the interface, but the semantics behind that activity. They allow the software author to express all the relevant relationships in an entire interactive application. The techniques can be divided into three categories: interaction methods, which includes nudges and hint highlighting; author generated objects, such as guide objects, cards, and decks of cards; and system generated objects, such as temporal ghosts.

4.1. Nudges

Gamut's method for recording demonstrations is called "nudges." The idea behind a nudge is that when the system makes a mistake or needs to learn new material, the author gives the system a little nudge. Basically, the application that the author is constructing is always running. When the running application is supposed to do something but does nothing, or does something when it is not supposed to, the author notices the problem and corrects the behavior as soon as the problem happens.

There are two kinds of nudges. The first is called "Do Something." The software author uses Do Something when the system sits idle when it is supposed to be performing some action. When the author sees the system miss a cue, the author selects the object that was supposed to act and pushes the Do Something button. If the system had previously seen that object perform an operation in a similar situation, the system has the opportunity to make a guess. Gamut will search behaviors that use the same starting event and behaviors that recently executed to find actions that will affect the selected object. If it finds a match, it will perform those actions as its guess. The author can accept a guess, ask the system to guess again, or reject the guess and demonstrate manually. If Gamut finds no match, it will ask the author to demonstrate the behavior manually.

The second nudge is called “Stop That.” Stop That is an indication to the system that incorrect actions just occurred. The author, when noticing a deviant action, selects the object that was affected and presses the Stop That button. The system immediately undoes all actions recently performed on that object. If the object was supposed to do nothing, the author is effectively done at this point. If the object was supposed to perform a different action, the author may ask the system to make a guess as it would for the Do Something nudge. The author may also modify objects manually.

For problems that do not fit neatly in the Do Something or Stop That category, pushing either button will work. When an object performs an action, but is supposed to do something else, selecting it and pressing Do Something will cause the system to search for a new action for that object to perform. Some guesses may not undo the original problem but others may. Pressing Stop That would cause the old action to be undone and from that point the author can demonstrate or have the system guess the correct action.

In an abstract sense, Do Something and Stop That represent positive and negative examples. New demonstrations are positive examples and are performed using Do Something. Stop That is a negative example since it asks that no action be performed in that given instance. Evidence from Frank [4] suggested that people found negative examples difficult to understand, but we suspect that those users had difficulty with the demonstration techniques in that particular system. That system and others require the author to demonstrate negative examples explicitly using special modes which draw attention to the example and away from the overall task. They also require the author to create a completely new demonstration. This assumes that the author understands why a bug exists and what events caused it to happen. Gamut only assumes that the author knows what was supposed to have happened and can alter the application objects to reflect that state.

Making negative examples easy to demonstrate has the secondary advantage that program structure is easier to infer. Negative examples permit the learning of disjunctive logic statements which in turn permits program structures such as if-then statements to be learned without the author having to create conditions explicitly.

Using nudges is an especially simple way to enter and edit examples. Demonstrating new examples and editing learned behaviors becomes the same process. Since the proportion of the actions needed to be changed or enhanced in a given behavior is likely to be small, new examples tend to require few operations. Furthermore, correcting mistakes becomes an immediate process. When a bug is seen, the author can correct it right away.

Nudges also reduces the number of system modes. First, the Run/Build mode distinction is eliminated. A prototype study mentioned later in this paper showed that people did not like switching between Run mode and Build mode and would often forget which mode they were in. In other systems, one can only create new objects and edit objects in Build mode. In Run mode, though, editing is turned off and the program’s behaviors are turned on so that the author may test how the application works. In Gamut, editing is always possible, and any behavior the author shows becomes active immediately. Toggle switches are used in situations where the running and editing behaviors overlap. For example, normally a button can only be pushed in Run mode and resized and moved in Build mode. Instead, Gamut uses a toggle between two selection pointers which are part of the tool palette in Figure 1. When the normal selection is used, buttons are pushed when clicked. Using strict selection, though, causes buttons to be selected so they may be edited. Other, non-widget, objects like rectangles and lines are selected normally in both modes. Similar toggle switches are used to demonstrate input events, hide guide objects, and for hiding Gamut's system windows.

Some PBD systems require a separate recording mode to enter stimulus events. This is part of the Stimulus/Response mode distinction advocated by Wolber in Pavlov [20]. A Stimulus/Response style interface is normally implemented as an extended macro recorder. With a macro recorder, pressing "record" causes the system to record all subsequent actions. "Playing" the macro later will execute those actions in a new context. In an extended macro recorder, an additional event recording phase is added. First, the author records an event. This is often troublesome since sometimes the event is part of a compound event. Consider recording just the down click from the mouse. To get an down click, one must later release the button in order to move on. Other systems have solved this problem by using timers [4] or by using buttons to disambiguate [20]. The stimulus is not ambiguous in Gamut because the author is expected to nudge the system immediately when the system errs. If the event is masked by later events, the author can retrace the last few program steps using undo. This makes the last event to occur the correct stimulus and there is no need for an extra mode.

4.2. Hint Highlighting

In his thesis, Kosbie suggests that a "focus hint" may be useful for improving inferencing in PBD [7]. A focus hint is a special form of selection where the author points out key elements that are important to a demonstration thereby "focusing" the system's attention on those objects. Kosbie did not actually implement this feature or show how it should be used, but Gamut actually uses this notion as its primary means for soliciting hints from the author.

The number of features upon which a single relationship may depend can be immeasurably large. Without a focus hint, finding the correct features requires an exponential amount of search time. Hints provide direction to guide heuristic search. Good hints can reduce search to near constant time.

When the right mouse button is pressed over an object, Gamut hint highlights it by drawing a green rectangle around it. Highlights around lines are seen as a thin rectangle that follows the line's direction. Highlighting is different from normal selection which is caused by the left mouse button and presented as a conventional set of square handles. Selection is used to move, resize, and recolor objects. It is common for the author to want to hint highlight an object, perform other operations, and then highlight other objects.

Maulsby studied how users generate hints in the Turvy experiment [12]. The results showed that some people give hints readily and prodigiously. Other people, however, did not provide any hints at all until the system got stuck and forced them to act. Hint highlighting provides for both styles of hint giving. Highlighting is permitted any time the author is demonstrating a new example. If the author fails to highlight objects that the system cannot guess, the system will ask the author a specific question about the situation and request that the author give a hint by highlighting something.

4.3. Deck of Cards

The vast majority of modern board games use a deck of cards. Game players know that cards can be used to simulate a large variety of behaviors. For games like Monopoly, cards are a source of random events like the Chance deck as well as the means for storing game state such as knowing which player owns each property. Cards and decks of cards are the main data structure in Gamut. Decks may be used in the manner that one uses them in actual games, or they can serve as sources of data and randomness.

Gamut's deck of cards is not the same card metaphor found in HyperCard [6]. In HyperCard, cards are the whole application. In general, a HyperCard "stack" is a set of screen displays with links between them to denote the method and order in which displays are presented. A Gamut deck is a widget within the application. To use a deck one drags objects into it. The deck will store and maintain the order of all objects it contains.

Decks have several operations. Dealing the top card is accomplished by dragging the object off of the deck. Shuffling the deck is performed by pushing a button. To find a specific card, the author uses arrow buttons to view each card in order. Putting objects into the deck is performed by dragging the object over the deck's area. A deck may also be opened and closed which controls whether items can be added or removed. This is important to align an object over top of a deck without putting it into the deck.

A deck may be used in a variety of ways not found in board games. For example, a deck can provide alternating images for an animated character. By stacking decks within decks, one can arrange to have images of a character walking in each direction. Shuffling a deck can be used to induce randomness into any behavior. An example later in this paper uses a deck to move a monster around randomly.

4.4. Card Widget

Though any object may be put into a deck, Gamut also provides a card widget. A card begins as a blank area on which the author can draw graphics. The card has a viewport creating a visible portion and an offscreen portion. Any object drawn over the visible region appears on the card widget. The visible region can be adjusted and moved by direct manipulation.

The common way to make a card is to draw the card's appearance in the visible region and to put numeric, textual and other data in the offscreen region. A Chance card in Monopoly might have a picture and description in the visible portion and the amount of money that is added or subtracted from the player's account in the offscreen part.

The windowing effect provided by cards has another use. There are many kinds of games where a character moves about a large space. In this scenario, often the background moves behind the character while the character remains relatively still. One can create this effect using a card. First, the author demonstrates how the character moves through the scene by actually moving the character in the most convenient way leaving the background alone. Then one transfers the whole scene to a single card. By attaching the card's viewing region to the character (using the group command for example), the view on the card will show the character as stationary and the background moving behind.

4.5. Guide Objects

Objects that are visible while the author is creating an application but are hidden when the application runs are called "guide objects." Maulsby's Metamouse [11] and Fisher *et al*'s Demo II [3] used these as well. A guide object shows a graphical relationship that would otherwise be invisible. Lines and arrows are used to show paths and connections. Rectangles can serve as placeholders. Guide objects can be used to demonstrate distances, locations, and even speeds.

Offscreen objects are another form of guide object. These are objects that are drawn outside of any visible surface like a window or a card's viewing region. Timers, counters, buttons and other widgets are all used as offscreen objects. These objects keep state information that is not stored directly on the board. For instance, a counter can be used to track the number of moves. A timer can be used to alternate images in an animated icon.

The purpose of guide objects is to enable the author to show the work involved with all behavioral relationships. In AI, this is called the “hidden object” problem [19]. In essence, a hidden object is a dependency or variable that the author has not included in the examples being shown. Any hidden object can be demonstrated by including the proper guide object(s) to represent it. Since a guide object can be effectively anything at any place, it is virtually impossible, in general, to guess what a guide object should be. However, it is possible to recognize when a relationship requires more than the author has shown. It is also possible to recognize a number of situations where authors commonly forget guide objects. Gamut’s inferencing algorithms can detect when relationships have not been fully specified. When guide objects are not highlighted or need to be created, Gamut asks a question designed to stimulate the author to create the needed objects.

4.6. Temporal Ghosts

A common problem for focus hints arises when the objects to be highlighted do not exist anymore. Interactive games are dynamic: objects are created, moved, and destroyed constantly. Temporal ghosts are a technique for keeping objects that change onscreen so that they may be highlighted. Ghosts also make the recent past visible so that the author can understand what changes have occurred.

Temporal ghosts are dimmed, translucent images of objects seen in their past state. If an object is moved, a ghost will appear in the object’s original position. If the object changes color, the ghost will appear directly on top of the object but the changed portion will appear as a color dithered halfway between the beginning and end colors. Other graphical effects will likewise show as ghosts of the original appearance. Mouse input events appear as ghost cursor arrows in the scene with small letters and symbols nearby to indicate the direction of the button click or letter of the keypress as in Marquise [15].

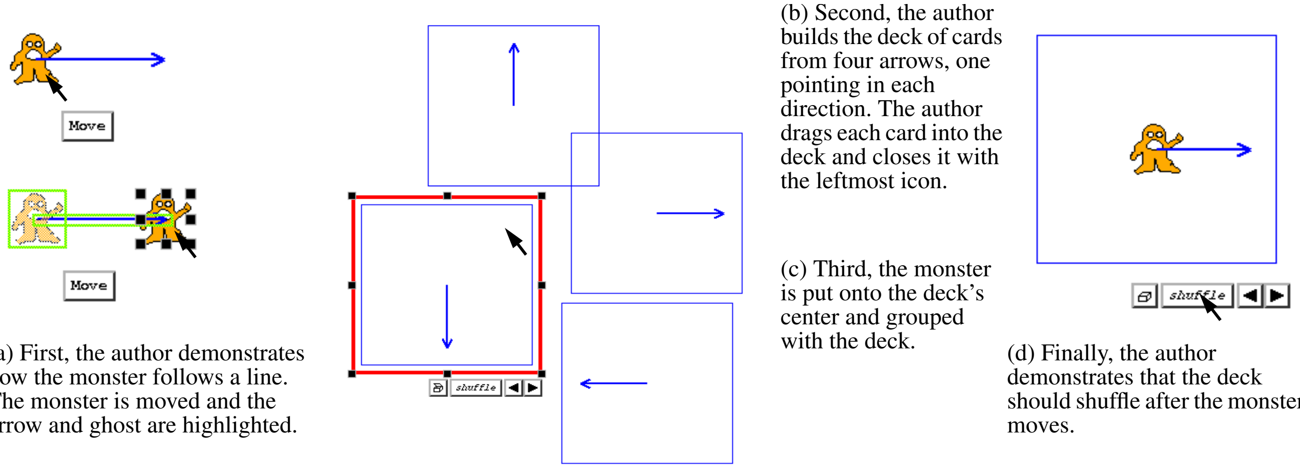
Since ghosts are predominately used for highlighting, ghosts only appear during demonstration. At the moment the author presses Do Something or Stop That, the ghosts will appear showing the state immediately prior to the last event. Showing the ghosts at other times is not desirable since so many would be present that the screen would be cluttered and confusing.

The use of dimming has been used by other systems to show state changes as well. Kramer’s Translucent Patches system used dimming and transparency to layer multiple sketches in a small area [8].

4.7. Dialogs

The nudges technique for recording demonstrations is designed to deliver a sense of freedom and openness. One can build new demonstrations in virtually any situation. Likewise, hints can be given as early as one wants. For beginners, this freedom can be overwhelming. Thus, Gamut provides structure during the demonstration recording modes to give authors direction.

Gamut has three phases of dialog feedback during recording mode. Each phase has an associated color which the author may but is not required to notice. The zeroth level would be simply the Do Something and Stop That buttons which have no particular color. The first level, colored blue, is the state acquisition phase. In this phase, the author is only expected to modify the state of the objects to correctly represent how they are meant to change. An expert user at this point will also highlight the objects on which the behavior depends. Should the system find enough information to incorporate the example, the recording session will end and the application with continue. Expert demonstrators need never see the later dialog



(a) First, the author demonstrates how the monster follows a line. The monster is moved and the arrow and ghost are highlighted.

(b) Second, the author builds the deck of cards from four arrows, one pointing in each direction. The author drags each card into the deck and closes it with the leftmost icon.

(c) Third, the monster is put onto the deck's center and grouped with the deck.

(d) Finally, the author demonstrates that the deck should shuffle after the monster moves.

Figure 3: Making a Monster Move Randomly

phases. Other operations available during the blue phase include causing objects to Stop That and asking the system to guess what an object should do.

The second phase, colored magenta, is used to ask the author questions. Questions occur when the system finds a contradiction or suspects that there is a relationship where an object was not highlighted. The system will generate one question at a time based on the situation that occurred. The questions are specific and discuss objects that are immediately affected by the behavior. The system does not try to generate general questions as they tend be vague and confusing. When each relationship question is brought up, the author has three choices for response. First, one may highlight the object upon which relationship depends and press the Learn button. The system then tries to incorporate this new information to generate a description. Second, one may choose the Replace button in cases where the original relationship or behavior was a mistake. Sometimes the author will have a change of mind and want to change a behavior drastically. The Replace button removes the old description and replaces it with a new one entirely. Finally, the third option is used in cases where the computer is confused and asks a question that is completely wrong. The Wrong button tells the computer that it has generated a bad question and that it should try a different line of reasoning.

The final phase of dialog is the yellow help phase. These dialogs come up if the system detects that the author is providing no helpful information. These dialogs offer suggestions based on the current question the system is asking. Suggestions range from telling the author how to highlight an object to pointing out what kinds of relationships the system can understand and suggesting that the author can create a guide object.

5. Example

Here is a short example showing how to produce a simple but effective behavior in Gamut. This example has been simplified considerably and shows how a behavior may be created after gaining considerable experience. A beginner would probably demonstrate something entirely different.

The example is making a monster walk randomly about the screen (see Figure 3). The strategy for this behavior will be to use a deck of cards to choose between a set of four directions. The monster will move by following the direction set by the deck.

First, the components are created and laid out in the main window. The monster is a bitmap image loaded from disk. The deck is created, and also a button widget labelled “Move” to generate events. To make the monster move repeatedly, one can replace the button with a timer widget. To represent the four directions the monster may travel, we draw four arrow lines, one pointing in each direction.

After the parts are created, one demonstrates how the monster should move along the arrow (Figure 3a). The monster is placed at the foot of one arrow line. The desired behavior will be that when the Move button is pressed, the monster will travel to the end of the line. Pushing the button at first causes nothing. The author selects the monster and pushes Do Something to generate the first example. Since no behaviors have been described, the system makes no guess and asks the author to modify the monster object manually. The author complies by moving the monster to the end point of the line. As this occurs, the system leaves a ghost object of the monster at the foot of the arrow. The author highlights the arrow because the arrow describes the location where the monster should be placed. The author also highlights the ghost of the monster since the starting point of the monster determines which arrow the monster should follow. Completing these hints, the author pushes the Done button ending the example. To test this new behavior, the author moves the monster to a different arrow line and pushes the button causing the monster to follow the line. Had the author not given any hints, the system would fail to move the monster correctly for the different line. At that point, the author would select the monster, push the Stop That button, and move the monster to its proper destination. Questions the system would ask the author would be about dependencies for the monster’s location and how to choose which arrow the monster is meant to follow.

Simply placing the arrows in the deck is not quite satisfactory since the starting points of the arrow lines would not line up. First, each arrow is grouped with a rectangle so that the start point is placed in the center. Then these groups are stored in the deck (Figure 3b).

Grouping the deck and the monster together (Figure 3c) is an effective way to cause the lines to follow the monster. When objects are grouped, they are permanently attached and cannot be broken apart except by ungrouping. Thus, moving the monster will also move the lines. We also direct the monster/deck group to move smoothly so that it will not make sudden jumping movements.

The final step will be to demonstrate shuffling the deck after the monster has finished taking a step (Figure 3d). Pushing the Move button once, the monster steps. The author selects the deck and pushes Do Something. The system finds no useful behavior to apply to the deck so it asks the author to demonstrate. The author responds by pushing the deck’s Shuffle button and pressing Done to complete the example. Pushing the Move button now causes the monster to move and the deck to shuffle, making the monster move about randomly.

6. Paper Prototype User Test

With innovative interaction techniques comes a heightened need for user testing. To test out ideas early, we performed a paper prototype study [18] in which a computer interface was simulated with paper cutouts and manipulated manually by the tester. The goal of the study was to find high level problems, with special attention paid to whether subjects could demonstrate examples effectively and what objects they would highlight. Between each subject, improvements were made in the paper interface to try to solve the problems that the last subject encountered.

Most portions of Gamut’s user interface translated well to the paper version. The background consisted of a drawing of the board with extra paper space for offscreen areas. Game pieces were represented by cardboard cutouts. Buttons and dialog boxes were also cut from card stock. Index card were used to represent cards and decks of cards. To indicate shuffling, the subject actually physically shuffled the deck. To hint highlight, the subject placed a penny on the object of interest.

Five subjects with little or no programming background were tested using three tasks. Sessions were restricted to a maximum of two hours. The first task was a two player board game where the players throw dice to determine how far to move their pieces. The second was a PacMan-like game where the subject demonstrated how to make a monster chase PacMan. The third was a Space Invaders-like game where the subject demonstrated how to shoot a randomly moving UFO. All subjects completed the first task and most completed the second task. Two subjects completed all three tasks with extraordinarily little difficulty.

As a result of the study, we eliminated many mode problems found in the original nudges design. The present method with no Run/Build mode distinction followed directly from these experiments. We also found that most people are reluctant to give hints to Gamut at first and will wait until Gamut asks a question. The subjects became more willing to give hints ahead of time after learning the objects that Gamut needs to have highlighted. One especially positive aspect the study suggested that subjects will create guide objects if properly requested. When subjects were presented with dialog boxes that asked direct, specific questions about the objects in the example, most were able to create a sufficient guide object that answered the question.

7. Inferencing

To manage the various interaction techniques, Gamut needs inferencing algorithms which can accept that data. The algorithms must handle new example actions, hint highlighting, and guide objects. Furthermore, they must be able to generate the behaviors characteristic of board games like chains of relationships and a large number of modes.

The Gamut's action language is based on the "command object" structure found in Amulet [14], the development environment in which Gamut has been implemented. A command object represents an atomic action such as Move, Create, Cut, Copy, and Paste. In Amulet, user actions are queued onto an undo history which Gamut uses as the input to the first phase of inferencing.

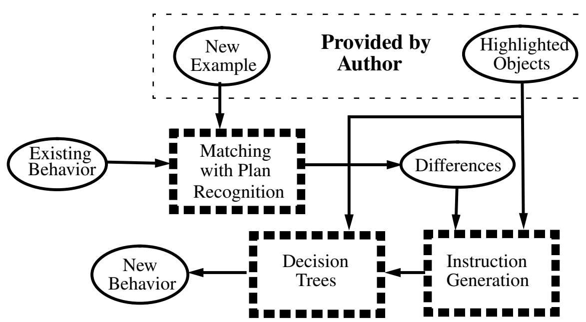


Figure 4: Phases of Inferencing

First, the commands are reduced to a canonical form which removes repetition and order dependencies. This step eliminates the need for authors to demonstrate specific actions to accomplish a goal. Some PBD systems make a distinction between actions like "move the start end of the line" and "move the whole line" and will fail if the author demonstrates the same result using different actions. Gamut is forgiving when the author uses a variety of actions or changes the actions' order. This reduced set of commands is used as the new example which is passed on to the matching phase.

The matching phase uses a plan recognition algorithm as its basis. The author's last input event determines the behavior that becomes the "plan" to be recognized. The algorithm can follow chains of relationships in the original behavior and determine which of the parameters should be changed in order to make the old behavior perform the actions in the new example. Output from this phase is the set of differences from the existing behavior and the new example.

The next phase of inferencing corrects the behavior for the differences found in the previous stage. New actions and relationships are incorporated into the old behavior. This phase uses any hint highlighting the author gives by searching for highlighted objects which can resolve a difference. The algorithm employed by this phase is heuristic and is based on the inferencing algorithm in Marquise [15]. When the system does not find a relationship, the system asks the author a question. The question is formed as "this behavior used to do X, but just now you showed that it should do Y, please highlight the thing that causes this to happen." The question encourages the author to highlight an object that will resolve the difference.

If Gamut still fails to find a relationship, it will use a decision tree to choose between the values that the author has already demonstrated. A decision tree is a data structure for choosing a value based upon a set of attributes. Each internal node in a decision tree is a test on one of the attributes and the leaves of the tree are values in the domain. The values in Gamut's decision trees denote actions for the behavior to perform or relationships in an action's parameters. Attributes for the decision tree are predicates on the state of the application. Common predicates include tests on the value of object slots or whether two objects graphically overlap. Given a set of highlighted objects, Gamut can generate a large set of predicates based on the objects' parameters. The decision tree algorithm will choose the predicates that best represent the relationship. The other predicates will eventually be thrown away after the tree has been used effectively several times. If the author is being especially ambiguous and the highlighted objects do not generate useful predicates, Gamut resorts to help dialog boxes to try to explain the relationships that Gamut understands and suggest that the author highlight other objects or create new guide objects to better describe the behavior.

8. Status And Future Work

Gamut is implemented using Amulet [13] and will run on Unix, Window, or the Macintosh. We expect to finish work on Gamut soon in order to begin formal user testing. The interaction techniques mentioned in this paper are all functional, but there are still areas that could be improved. For instance, Gamut's dialog generator could create much better English text to describe behavior differences. Also, we would like to incorporate several more forms of relationships so that Gamut may generate more behaviors with fewer examples.

Gamut still has several limitations such as environmental limitations where operations and objects do not exist in the Gamut's editor and system. For instance, 3-dimensional transformations and objects are missing, as are rotation and sound effects. These behaviors cannot be taught because the basic support for demonstrating them is lacking. Also, certain tasks are not appropriate for Gamut. Even when the environment permits demonstration of an inappropriate task, it would require a prohibitively large number of demonstrations. Gamut has been designed to accept a wide variety of game abstractions with the least difficulty. Tasks like editing a word processor document are not easy to teach.

The final destiny of Gamut will be to test these ideas in a set of formal user studies. We expect that nonprogrammers will be able to use Gamut to create behaviors of significant complexity. Beyond formal testing, Gamut may eventually be released for general use.

9. Conclusion

Gamut has the ability to infer complex behaviors which can be used to build complete interactive applications. This new capability derives from an innovative collection of interaction techniques coupled with inductive learning algorithms that can take advantage of the techniques. The nudges interaction simplifies example recording and provides a simple manner to create negative examples. Hint highlighting is a means for improving the system's guessing by allowing the software author to point out important objects in a behavior. The deck-of-cards metaphor and the card widget allow complicated behavior to be specified that can involve sets of data and randomness. Guide objects permit demonstration of objects and relationships which the system could not guess by itself. And temporal ghosts allow the author to directly form relationships with the recent past. By providing the author with the ability to give hints, negative examples, and use powerful objects and metaphors, Gamut allows people to make a broader range of applications with a minimum of programming expertise.

10. References

1. *Authorware*. Authorware Inc. 8400 Normandale Lake Blvd., Suite 430, Minneapolis MN 55437, 612-912-8555, 1991.

2. B. Budge. *Pinball Construction Set*. Exidy Software.

3. G. L. Fisher, D. E. Busse, D. A. Wolber. *Adding Rule-Based Reasoning to a Demonstrational Interface Builder*. Proceedings of UIST'92, pp 89-97.

4. M. Frank. *Model-Based User Interface Design by Demonstration and by Interview*. Ph.D. thesis. Graphics, Visualization & Usability Center, Georgia Institute of Technology, Atlanta, Georgia.

5. L. Grimm, D. Caswell, and L. Kirkpatrick. *Playroom*. Broderbund Software, 500 Redwood Blvd., Novato, CA 94948-6121, 1992.

6. *HyperCard*. Apple Computer Inc., Cupertino, CA, 1993.

7. D. Kosbie. *KATIE: Architectural Support for Programming by Demonstration*. Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, in progress.

8. A. Kramer. *Translucent Patches*. Proceedings of the ACM Symposium on User Interface Software and Technology. UIST'94. Marina del Rey, CA, 1994, pp 121-130.

9. F. Lionet, Y. Lamoureux. *Klik & Play*. Europress software, 1996.

10. Macromedia, *Director*, 600 Townsend Street, San Francisco, CA 94103, macropr@macromedia.com, http://www.macromedia.com/, 1996.

11. D. Maulsby. *Inducing Procedures Interactively: Adventures with Metamouse*. Masters thesis. Research Rept. 88/335/47. Univ. of Calgary, December, 1988.

12. D. Maulsby. *Instructible Agents*. Ph.D. thesis. Department of Computer Science, Univ. of Calgary, Calgary, Alberta, June 1994.

13. B. A. Myers, A. Ferrency, R. McDaniel, R. C. Miller, P. Doane, A. Mickish, A. Klimovitski. *The Amulet V2.0 Reference Manual*. Carnegie Mellon University Computer Science Department, CMU-CS-95-166-R1, May 1996.

14. B. A. Myers, D. S. Kosbie. *Reusable Hierarchical Command Objects*. Human Factors in Computing Systems, Proceeding SIGCHI'96, Denver, CO, April, 1996, pp 260-267.

15. B. A. Myers, R. G. McDaniel, and D. S. Kosbie. *Marquise: Creating Complete User Interfaces by Demonstration*. Proceeding of INTERCHI'93: Human Factors in Computing Systems, 1993, pp 293-

300.

16. J. R. Quinlan. *Induction of Decision Trees*. Machine Learning, Kluwer Academic Publishers, Boston, Vol. 1, 1986, pp 81-106.

17. *Reader Rabbit*. The Learning Company, 1987.

18. M. Rettig. *Prototyping for tiny fingers*. Communications of the ACM 37, 4 (April 1994). pp. 21-27.

19. K. VanLehn. *Learning One Subprocedure per Lesson*. Artificial Intelligence, Vol. 31, 1987, pp 1-40.

20. D. Wolber. *Pavlov: Programming By Stimulus-Response Demonstration*. Human Factors in Computing Systems, Proceeding SIGCHI'96, Denver, CO, April, 1996, pp 252-259