Getting More Out Of Programming-By-Demonstration

Richard G. McDaniel and Brad A. Myers
HCI Institute, School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213 USA
{ richm, bam }@cs.cmu.edu

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 PBD tool that nonprogrammers can use to create a broader range of interactive software, including games, simulations, and educational software, than they can with other PBD tools. To do this, Gamut provides advanced interaction techniques that make it easier for a developer to express all aspects of an application. 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 demonstrating relationships that the system would find too difficult to guess, and temporal ghosts which simplify showing relationships with the recent past. These techniques were tested in a formal setting with nonprogrammers to evaluate their effectiveness.

Keywords
End-User Programming, User Interface Software, Programming-by-Demonstration, Programming-by-Example, Application Builders, Inductive Learning, Gamut.

INTRODUCTION
Gamut is an innovative tool for building interactive software like games, simulations, and educational software. Much of the effort involved in producing software in this domain is not in programming the application's logic but in providing the engaging background, artwork, and gameplay that keeps the users interested. Artists and educators who could produce such material are often unable to program computers. Thus, tools which eliminate the burden of programming while providing a wide range of capabilities are desirable.

Traditional development tools for producing interactive software require extensive programming knowledge. Programming graphics in common environments like Visual C++ or Visual Basic can be difficult even for seasoned programmers. Tools such as interface builders can help developers design the visual appearance of an application but still require programming to make the interface actually work. Application builders such as Click & Create [3] eliminate programming but impose severe limits on the kinds of programs that can be created. Authoring tools like AuthorWare [1] or Director [8] are similarly limited and cannot produce complex behaviors and player interactions without using their built-in scripting languages.

To appear in Proceedings CHI'99

One method for simplifying the programming process has been programming-by-demonstration (PBD). Rather than using a textual notation, the developer 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 developer's intention and creates the 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 through the use of improved interaction techniques. The interaction techniques allow the developer to give Gamut all of the required information without resorting to a written programming language. 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.

DOMAIN
Gamut can create games and simulations similar to board games. These are two-dimensional games with a board-like background that uses playing pieces to represent the game's 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 [16] and Playroom [6] and video games like PacMan can all be made using Gamut.

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

- The created games are interactive and require player input. Some PBD systems only assist in editing static data such as a text document.
- Board games have a large number of states and modes. Game behavior can be triggered by a variety of events and can have complicated relationships.

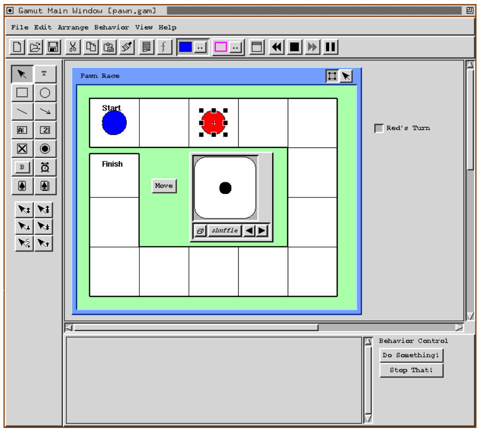

Figure 1: Gamut's main window. On the left are the tool palette and mouse icons. Along the bottom is the behavior dialog area.

- 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 such as 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 developer has to show the system all relationships upon which a behavior depends.

EXAMPLE

To motivate the design of Gamut's interaction techniques, we will show how to build the simple board game application shown in Figure 1. This game was also used as a task in the usability study discussed later. In the game, two pieces colored red and blue follow the path of squares around the edge. The first piece to reach the end wins. The pieces alternate turns and move the number of spaces as shown on the die in the center. As an added complication, whenever one piece lands on another, the landed-on piece must go back to the beginning.

As each of the following interaction techniques is presented, we will show how the developer uses that technique to build this board game.

INTERACTION TECHNIQUES

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

Guide Objects

Guide objects are graphical objects and widgets that are visible while the developer is creating an application but are hidden when the application runs. Gamut supports two kinds of guide objects. The first is derived from Maulsby's Metamouse [9] and Fisher *et al*'s Demo II [4] which allowed certain graphical objects to be made invisible on demand. *Onscreen* guide objects show graphical relationships between other objects on the screen, visible or invisible. Onscreen guide objects can be used to demonstrate distances, locations, and even speeds.

For instance, in the example game, the path that each piece follows around the board can be represented with arrow line guide objects as shown in Figure 2. Guide objects are drawn in pastel colors so they will be distinct from application objects. At any time, the developer can make the guide objects invisible by switching a mode. Without the path, the system would not know how the squares around the board were connected, and might not even be able to see that the squares exist at all. Allowing the developer to draw the graphical connections saves the system from having to provide sophisticated machine vision heuristics to achieve the same effects.

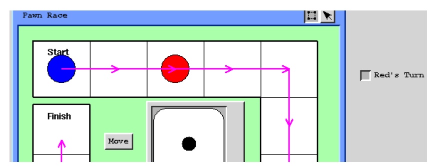

Figure 2: The developer uses arrow lines as guide objects to represent the path the pieces follow.

Other guide objects are placed o*ffscreen*. The player's view is shown as a blue window frame in the middle of the developer's drawing area as seen in Figure 1. Objects that are drawn outside of the blue frame cannot be seen by the player and are offscreen. Offscreen guides objects are used to represent the application's data that is not stored directly on the board. Timers, counters, toggle buttons, and other widgets are all used as offscreen objects.

In our example game, the developer needs to represent the player turn order. The developer decides to use a checkbox and places one outside of the frame window and labels it "Red's Turn." (In the Motif look-and-feel, a "checkbox" looks like a raised or lowered rectangle next to a text label.) The widget begins with its checkmark on.

The purpose of guide objects is to enable the developer to show relationships that are nearly impossible to infer. In AI, this is called the *hidden object problem* [18]. A hidden object is a dependency or variable upon which a behavior depends that is not included as part of the application's visible state. Gamut cannot infer hidden objects without the developer's help because it has no way to determine what such objects could be. The number of possible things a hidden object could be is virtually infinite. However, it is possible to recognize when a relationship requires more than the developer has shown. Gamut's inferencing algorithm can detect when relationships have not been fully specified and asks the developer to tell the system about missing objects.

Deck Widget

Cards and decks are the two major data structures in Gamut. Many modern board games use decks of cards to simulate a large variety of behaviors. In 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. In Gamut, decks may be used to represent lists of numbers, objects, colors, etc., and they provide a randomization feature (shuffling) which is useful for constructing random behaviors.

A deck may also be used to produce video game behaviors. For example, a deck can provide alternating images for an animated character. To make a character move randomly, its position can be tied to a deck containing an arrow for each direction that gets shuffled each time the character moves.

Gamut's deck of cards is not the same card metaphor found in HyperCard [7]. 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 in Gamut, one drags objects into it. The deck will store and maintain the order of all objects it contains.

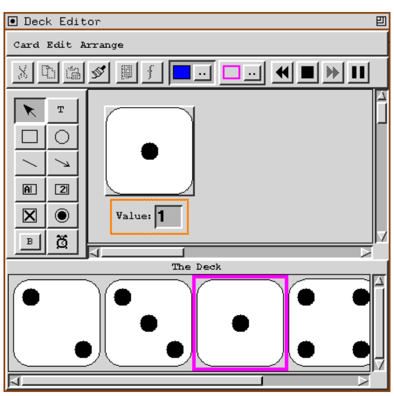

Figure 3: The card and deck editor. This shows the contents of the deck in Figure 1. Each face of a die is presented as a card. The current face is shown in the main drawing area.

Card Widget

Gamut's card widget acts as a large drawing surface, independent from the main window. Naturally, cards may be placed in decks, but they can also be used on their own. The developer uses Gamut's card editor (see Figure 3) to draw on a card. The raised frame in the card's drawing area shows what is visible within the card's widget. Objects drawn outside the visible region act as offscreen guide objects similar to offscreen objects in the main window.

In the example game, the developer uses a deck to represent the die in the center of the board (see Figure 1). The deck editor shows each item in the assembled deck which in this example is a set of cards each representing a face of the die (see Figure 3). The developer can demonstrate "rolling" the die by shuffling the deck. In the drawing portion of the deck editor, the developer draws the pips for each face in the card's visible region. Below the visible region, the developer adds a number box and types the numeric value of the die face. By including the number box, the system will not have to count the dots in order to infer the value of the card.

Demonstrating Behavior

Gamut introduces a new way to demonstrate behavior which we call *nudges*. The idea is that when the system makes a mistake or needs to learn new material, the developer gives the system a "nudge" telling the system immediately where it went wrong. In other words, when the application is supposed to do something but does nothing, or does something when it is not supposed to, the developer nudges the system and corrects the behavior.

Gamut defines two kinds of nudges. The first is called "Do Something." The developer uses Do Something to demonstrate new behaviors. When the developer sees the system miss a cue, the developer pushes the Do Something button. The system then becomes ready to accept the developer's new example permitting the developer to modify the application's state appropriately.

The second nudge is called "Stop That" which tells the system that one or more objects did something wrong. The developer, when noticing a deviant action, selects the affected object and presses the Stop That button. The system immediately undoes all actions just performed on that object. If the object was supposed to do nothing, the developer is finished at this point. If the object was supposed to perform a different action, the author may modify objects to show the system the correct behavior.

In our example, the developer wants to demonstrate that when the player pushes the application's "Move" button, the game will respond by moving the current player's piece. The developer first pushes the Move button (it is to the left of the die in Figure 1). Though the developer pushes the button, the application will do nothing because no behavior has yet been demonstrated. So the developer pushes Do Something at which point the system prepares to accept a new example. The system displays *temporal ghosts* to show how objects have changed from the previous state, activates *hint highlighting* so the developer may give hints, and presents a dialog asking the developer to complete the example and press the Done button when complete.

The application is supposed to roll the die and move the piece the corresponding number of places. It also must update the player turn. The developer pushes the "Shuffle" button on the die's deck, moves the red piece the corresponding number of squares, and toggles the turn indicator to be unchecked. The view would now look like Figure 4a. At this point, before pressing "Done" to finish the example, the developer should give the system *hints*.

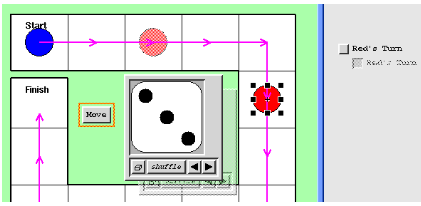

Figure 4a: The developer has just moved the red piece which used to be in the middle of the top row. The red piece's ghost shows where it was.

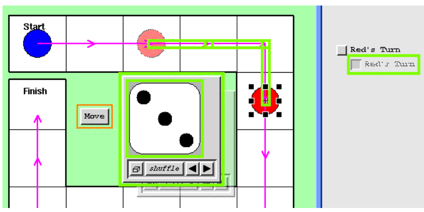

Figure 4b: The developer has highlighted the arrow lines, die, and ghost of the toggle switch as hints.

Hint Highlighting
A "hint highlight" 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. Maulsby implemented a similar feature in Cima [10] by having the user select a word or phrase and use a menu command to make it a hint. Hints are used to reduce the size of the inferencing algorithm's search space. The number of features upon which a single relationship may depend can be immeasurably large. Finding the correct features can require an exponential amount of search time without hints. Providing hints can reduce the search to near constant time.

In Gamut, the developer highlights objects as hints by pressing the right mouse button over them. Gamut marks hinted objects with green rectangles. Highlight marks around lines are seen as a thin rectangle that follows the line's direction as seen in Figure 4b. Highlighting is different from normal selection which is caused by the left mouse button and presented as a conventional set of square handles. (The circle on the right of Figure 4a and b is selected.) Selection is used to move, resize, and recolor objects. Since it is common for the developer to want to hint highlight an object and still perform other operations, highlighting is made an independent operation from selection.

In the example, the developer knows that the path of the piece is important as well as number shown on the die; so, the developer highlights the lines in the path and the die object. The developer will also need to highlight something that shows what value to set the current player checkbox. The checkbox only toggles back and forth between true and false so its only dependency is its own value. Thus, the new value of the checkbox depends on its original value. To highlight the original value, the developer highlights the checkbox's *temporal ghost* which is shown on the right in Figure 4b.

Temporal Ghosts
A common problem for hint highlighting arises when the objects that need 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. Though the concept of ghost objects is not new, Gamut is the first to use it for PBD.

Gamut displays temporal ghosts as 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 below the object but offset so the developer can still see it.

When the developer toggles the turn indicator in the example, a ghost appears below it, offset to show that it used to be checked (see the right portion of Figure 4a). The developer highlights this ghost to show that its value is important. Gamut will be able to see that the new value is different from the old value and use that to describe how the toggle changes. Finally, the developer pushes "Done" because the example is finished.

For the second example, the developer pushes the "Move" button again. The system is able to incorporate enough information to know how far to move the pieces but it does not know that it is supposed to move the blue piece and so it moves the red piece. The developer notices that this behavior is wrong, selects the red piece and presses "Stop That." Gamut immediately undoes the move action performed on the red piece. Stop That places the system into the same mode as Do Something so temporal ghosts are displayed once again and hint highlighting is made active. The developer moves the blue piece the correct number of spaces and also changes the player turn indicator if the system did not already do so. When finished, the developer presses "Done." This time, the developer did not highlight any objects. As a result, when the system finds ambiguities, it will ask the developer questions in order to resolve them.

Question Dialogs
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 in the behavior dialog region of the window. The questions ask about the objects or values that are immediately affected by the behavior. Developers have three choices for response. First, they 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, they may choose the "Replace" button in order to directly replace the old value with the new. Replace is used to correct mistakes or to modify a behavior from its original form. Finally, the third button is called "Wrong" and is used in cases where Gamut asks a question that makes no sense. Gamut generates its questions using heuristics which can sometimes fail. The Wrong button tells the computer that it has generated a bad question and that it should try a different line of reasoning.

In the developer's second example in the example application, the system sees the blue piece move instead of the red piece but it does not know why. The system asks the developer to highlight the object that best describes why the blue piece has moved and not the red. The developer highlights the ghost of the turn indicator and presses Learn. Highlighting the turn indicator tells the system that the blue piece moves when the checkbox is unchecked. To finish this portion of the example, the developer would need to test the Move button one more time and correct the system (without needing further highlighting) to complete the behavior.

Mouse Input Icons

The example application does not directly use mouse input in the window since the button widget handles the mouse automatically. However, it is worth mentioning how demonstrating mouse events is accomplished in Gamut. Gamut has a palette of mouse events below the main tool palette (see Figure 1). To create a mouse event, the developer selects an event and drops it onto the window as though it were a graphical object. This allows the developer to demonstrate mouse events without entering a special mode. The system will respond as though the player had just produced the selected event. Keyboard and other sorts of events, though not implemented, could be included in a similar way.

Gamut uses the same icons to represent mouse events as we used in our Marquise system [14]. Clicking events are shown with an arrow pointing up as well as down whereas button down events only point down. Double clicks are shown as two arrowheads pointing down and moving/dragging uses a wavy line. The icons are shown in Figure 5.

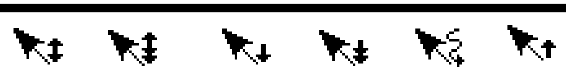

Figure 5: Icons that are used to represent mouse events.

Contrasting Nudges With Other Systems' Techniques

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 signifies a negative example since it asks that no action be performed in that given instance. Evidence from Frank [5] suggested that developers found negative examples difficult to understand, but we suspect that those users had difficulty with the demonstration techniques in that particular system. Frank's system and others require the developer to demonstrate negative examples using special modes. This requires the developer to understand *a priori* when a negative example is required and it draws attention to the example instead of to the behavior that it represents.

With nudges, the developer stays focused on the behavior of the application. Each nudge provides an incremental improvement to the behavior that the developer is testing. The developer does not have to know whether a particular example is positive or negative, and must only tell the system when objects are not behaving correctly.

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 by manually changing the inferred code.

Another advantage of nudges is that they reduce the number of system modes. Some PBD systems require a separate recording mode to enter stimulus events. This is part of the Stimulus/Response mode distinction used in various system including Pavlov [19]. 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. An extended macro recorder adds an extra *stimulus* phase where the developer demonstrates the event of the behavior.

Revising behaviors can be more tedious using the extended macro recorder technique than Gamut's nudges approach. Since the macro recorder requires the developer to perform the event during the stimulus recording phase, the developer must know beforehand what he or she intends to demonstrate. If the developer were to find a problem while testing the application, the stimulus event that occurred during the test would have to be recreated for the macro recorder. Furthermore, the initial state for all the objects involved would have to be reset. With the nudges approach, the developer can create new examples as the need occurs. When a problem appears during testing, the developer can immediately nudge the system and use the current application state in the example.

INFERENCING

To manage the various interaction techniques, Gamut needs an inferencing algorithm that can understand them. The algorithms must handle multiple examples incrementally while incorporating hint highlighting of various objects, guide objects, and temporal ghosts. Furthermore, the algorithms must be able to generate the behaviors characteristic of board games like chains of relationships with conditional components. For a more complete description of Gamut's inferencing algorithm see our previous paper [11].

Gamut's action language is based on the *command object* structure found in Amulet [13], the development environment used to implement Gamut. 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 stage of inferencing (see Figure 6).

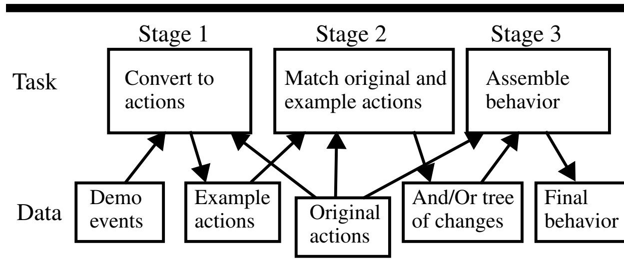

Figure 6: Stages in Gamut's inferencing algorithm.

In the first stage, the commands from the undo history are reduced to a canonical form which removes repetition and order dependencies. The events found in the undo list are often complicated. A group command, for instance, not only creates a group object but also moves and reparents the objects being grouped. This stage converts the events into a small set of basic actions. This reduced set of commands is passed on to the matching stage.

The matching stage uses a plan recognition algorithm as its basis [18]. If the application already has behavior defined for the event being demonstrated, the old behavior is used as a template for recognizing features in the new example. (If there is no previous behavior, the algorithm moves to stage three.) The algorithm can follow chains of descriptions 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 set of changes is captured in an And/Or tree which represents the various ways the changes might be applied. And-nodes represent changes that must occur together while Or-nodes are used to store alternatives.

The final stage of inferencing resolves the differences found in stage two and describes new parameters. This stage uses the objects highlighted by the developer to search for descriptions which resolve differences stored in the And/Or tree. The algorithm employed by this phase is heuristic and is based on the algorithm in Marquise [14]. When Gamut does not find a suitable description, it asks the developer a question in the behavior dialog. The text of the question is based on the current unresolved difference and refers to the value in the original behavior, the new value in the example, and some context information stating what the value affects. The developer is asked to highlight the appropriate objects to answer the question.

If Gamut still fails to find a suitable description, it will use a decision tree to choose between the old and new values. The precise way that Gamut applies decision trees is beyond the scope of this article. The basic idea is that Gamut generates attributes for the decision tree algorithm using the objects the developer highlights. The algorithm (specifically ID3 [15]) will, in turn, decide which attributes to apply and in what order. This allows Gamut to generate conditional expressions with objects that are not directly affected by the behaviors they control.

USER TESTING

We tested Gamut under formal conditions to see how well the techniques would be understood by nonprogrammers. In a short three-hour session, the test participants had to learn the system and build two tasks using Gamut. One participant was also invited back to attempt a third task which was longer and required mouse input. Overall, Gamut performed fairly well. Three of the four participants were able to complete the tasks and the one person who attempted the third task completed it as well.

Participants

The study's participants were contacted through electronic bulletin boards and email at Carnegie Mellon University. The subjects were required to be nonprogrammers. Specifically, the subjects were allowed to have taken a low-level class in programming, but were not allowed to program computers for a living or as a hobby. Participants were required to be familiar with typical computer interface metaphors as well as drawing editors.

Tasks

In the three hour sessions, the participants had to complete an hour-long tutorial and two tasks. We asked the participants to use a "think-aloud" protocol [17] to articulate their thoughts. The sessions were videotaped and an experimenter was present to answer the participant's questions.

The first task was based on a matching test similar to some educational games like Reader Rabbit and is shown in Figure 7. It was called Safari and it consists of two decks of cards. One contained a list of animals like "Zebra," and the other contained a list of questions about animals like "Does it have stripes?" The goal of the task was to put guide objects into the deck that would tell Gamut the correct answers and to demonstrate the behavior of a pair of buttons labelled "Yes" and "No."

The second task was the board game task that we used in this paper as an example and can be seen in Figure 1. The participants did not have to draw the board or create the die, but did have to create guide objects to represent the path the pieces followed, create something to represent the turn indicator, and demonstrate all the behaviors.

The third task which only one participant attempted was based on the video game Q*bert and is also shown in Figure 7. It had a character which jumps from cube to cube in a pyramid and collects objects. There is also an enemy ball which falls down from the top of the pyramid in random directions. The third task required a longer time to complete than the others so it had its own session. The participant was given a shortened tutorial which described the new interaction techniques required in the third task.

Observations

The purpose of the study was to see whether nonprogrammers could use Gamut to demonstrate behavior. Few PBD systems in the past have ever been tested with actual users so we were mostly interested in proving that PBD could actually work.

As shown in Table 1, each participant used a different set of Gamut's techniques in order to complete the tasks. For instance, some participants preferred to use Do Something exclusively, others preferred to use Stop That. Only one participant was unable to learn enough of Gamut's techniques to build the applications.

One of the problems the participants had concerned highlighting ghost objects. Most participants showed a peculiar reluctance to highlight a ghost object. Several preferred to highlight the original object and not its ghost even though the original was modified and displayed a different state from what the participant wanted the system to learn. The "Highlight Ghosts" line on the table shows that only one participant was truly comfortable highlighting ghosts while the others would choose not to most of the time.

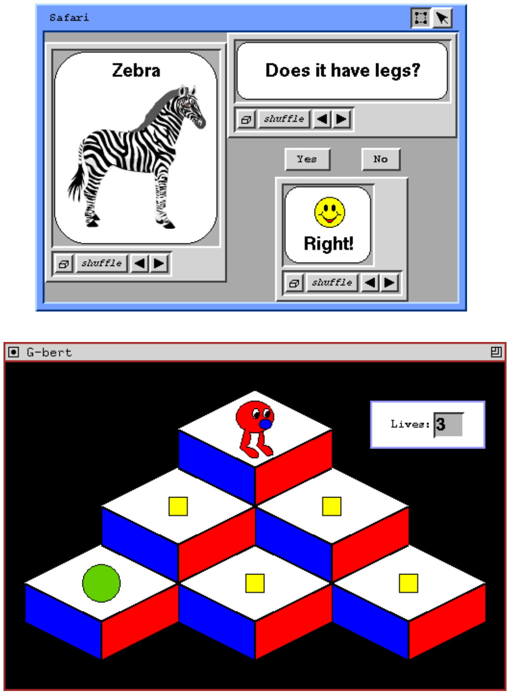

Figure 7: Screenshots of the first and third usability tasks. The top one is Safari which is based on an educational game. The lower one is G-bert which is a video game.

The most significant problem we discovered in Gamut concerns guide objects. Developers were always reluctant to create guide objects to use in their demonstrations. The pilot study showed we had to include specific instructions to tell the participant to draw guide objects. However, once told, participants were usually able to create objects that were suitable for the task they needed to demonstrate. The only successful participant who had difficulty creating guide objects after being told to do so was P4. It turns out P4 eventually did create all the needed guide objects, but only after asking the experimenter so many questions that it was not clear whether P4 had designed the objects herself or whether the experimenter had given away the answers.

A problem that all participants shared (though to different degrees) was highlighting inappropriate objects as hints when Gamut asked a question. Gamut requires that the developer provide a hint when it asks a question, but there seemed to be situations where certain objects were considered too obvious to highlight. Ghost objects seemed to be one instance. Visual paths and lines connected to objects seemed to be another.

When the system needed to have an object highlighted that was too obvious for the participant's taste, the participant would often choose to highlight an object which had a less obvious connection to the behavior. Gamut is designed to be resilient to badly highlighted objects so the participant usually had multiple opportunities to answer the same question, though the system's performance becomes considerably worse each time the participant gives a bad hint. Eventually, most participants would highlight the appropriate object and the system could proceed.

techniques	P 1	P 2	P 3	P 4
Do Something	X		X	X
Stop That		X		X
Highlights Ghosts				X
Guide Objects		X	X	
Cards and Decks		X	X	
Player Mouse Icons*		X		

Table 1: Different participants learned and used different sets of Gamut's techniques. All participants except the first were able to successful complete the tasks.

* Only participant two was given a task that required the Mouse Icons to complete.

The Cards and Decks and Player Mouse Icons lines of the table show that two participants used decks in their tasks and one used the mouse icons. The first two tasks came with decks of cards already prepared. All participants had little difficulty using the decks. However, P2 and P3 created and used an original deck of cards as their own widget. The Player Mouse Icons were only needed for task 3 and were only used by P2 who had little trouble.

It is not entirely clear why P1 was unable to use the system. According to the participant's own comments, he was "too tired" to learn how the system worked. The poor result may have simply been due to fatigue from a long day at work. The sorts of errors P1 would make were mostly caused by forgetting how to demonstrate using nudges. For instance, sometimes he might not demonstrate the event for the example and sometimes he would not push Do Something or Stop That.

RELATED WORK

A number of tools exist for building games. Most construct a specific class of games such as Bill Budge's Pinball Construction Set [2] which makes pinball simulations. A recent product is Click & Create [3] in which the developer 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 the author is limited to the built-in methods.

Gamut most resembles our previous system, Marquise [14]. 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. It also had a limited set of expressions for describing objects and locations which caused it to make poor inferences. The only means for correcting the system was editing the inferred code using a set of unwieldy dialog boxes.

PBD systems such as Wolber's Pavlov [19] and Frank's Grizzly Bear [5] 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). Unfortunately, linear constraints cannot be used to infer conditional expressions based on modes and many other kinds of behavior needed to build whole applications. Pavlov requires users to annotate their demonstrated behaviors with conditional guard statements in order to overcome these problems.

Gamut's inferencing ability is similar to Maulsby's Cima system [10]. 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. Also, Cima currently cannot do work on behalf of the user: it just recognizes strings of text.

STATUS AND FUTURE WORK

Gamut is implemented using Amulet [12] and will run on Unix, Windows, or the Macintosh. Gamut is a prototype system implemented as the first author's thesis project. It is not a commercial software product and it is not available for release except for research purposes. Though Gamut is functional, more work would be needed to make it usable for typical developers.

We have used Gamut to demonstrate behaviors that other systems cannot produce. For instance, we have created a Turing machine emulation, complete Tic-Tac-Toe and Hangman games, various video game behaviors such as a PacMan-like monster. Behaviors from educational games such as matching words as in Reader Rabbit have also been created. Thus, Gamut has been used to demonstrate a broad range of behaviors without resorting to a written programming language at any point.

Though Gamut's input techniques work well, the interface still needs to provide better feedback. Currently, the system has only the behavior dialog to tell the developer about the behavior being demonstrated. More work is needed to inform the developer about what the system knows and what it can infer. The system needs to note graphical constraints and to have a dialog mechanism for displaying the inferred code in an understandable format.

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 allows complicated behaviors 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. Finally, temporal ghosts allow the author to directly form relationships with the recent past. These techniques were tested in a usability study where we found that nonprogrammers were able to use them to build realistic application behaviors effectively. Thus, Gamut's techniques are an effective method for demonstrating a broader range of applications with a minimum of programming expertise and would be appropriate for use in a wide range of future PBD systems.

ACKNOWLEDGEMENTS

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 the U.S. Government.

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. *Corel Click & Create*. Corel Corporation and Europress Software Ltd. 1996.
4. 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.
5. 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.
6. L. Grimm, D. Caswell, and L. Kirkpatrick. *Playroom*. Broderbund Software, 500 Redwood Blvd., Novato, CA 94948-6121, 1992.
7. *HyperCard*. Apple Computer Inc., Cupertino, CA, 1993.
8. Macromedia*, Director*, 600 Townsend Street, San Francisco, CA 94103, macropr@macromedia.com, http://www.macromedia.com/, 1996.
9. D. Maulsby, I. Witten. "Inducing Procedures in a Direct-Manipulation Environment." *Proceedings SIGCHI'89*, April, 1989. pp. 57-62.
10. D. Maulsby. *Instructible Agents*. Ph.D. thesis. Department of Computer Science, Univ. of Calgary, Calgary, Alberta, June 1994.
11. R.G. McDaniel, B.A. Myers. "Building Applications Using Only Demonstration." *Proceedings of IUI'98*. pp 109-116.
12. B. A. Myers *et al*. "The Amulet Environment: New Models for Effective User Interface Software Development." *IEEE Transactions on Software Engineering*, Vol. 23, no. 6. June 1997. pp. 347-365.
13. B. A. Myers, D. S. Kosbie. "Reusable Hierarchical Command Objects." *Human Factors in Computing Systems, Proceedings SIGCHI'96*, Denver, CO, April, 1996, pp 260-267.
14. B. A. Myers, R. G. McDaniel, and D. S. Kosbie. "Marquise: Creating Complete User Interfaces by Demonstration." *Proceedings of INTERCHI'93: Human Factors in Computing Systems*, 1993, pp 293-300.
15. J. R. Quinlan. "Induction of Decision Trees." *Machine Learning*, Kluwer Academic Publishers, Boston, Vol. 1, 1986, pp 81-106.
16. *Reader Rabbit*. The Learning Company, 1987.
17. M. Rettig. "Prototyping for tiny fingers." *Communications of the ACM* 37, 4 (April 1994). pp. 21-27.
18. K. VanLehn. "Learning One Subprocedure per Lesson." *Artificial Intelligence*, Vol. 31, 1987, pp 1-40.
19. D. Wolber. "Pavlov: Programming By Stimulus-Response Demonstration." *Human Factors in Computing Systems, Proceedings SIGCHI'96*, Denver, CO, April, 1996, pp 252-259