Gamut: Creating Complete Applications Using Only Programming-by-Demonstration

RICHARD G. MCDANIEL and BRAD A. MYERS
Carnegie Mellon University

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 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 along with improved inferencing that makes use of these techniques. The interaction techniques include a simplified way to demonstrate new examples, called *nudges*, and a way to give the system *hints* by highlighting 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 showing relationships that the system would find too difficult to guess, and *temporal ghosts* which simplify showing relationships with the recent past. Gamut can infer behaviors that other PBD systems would require a developer to code by hand. Gamut infers behaviors incrementally from multiple examples and never requires the developer to edit code. Gamut revises code automatically using a recursive procedure that efficiently scans for the differences between a new example and the original behavior. Differences that cannot be resolved by generating a suitable description are handled using *decision tree learning* providing a significantly greater ability to infer complex relationships. These techniques were tested in a formal setting with nonprogrammers to verify that they are effective.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques--*user interfaces*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces; I.2 [**Artificial Intelligence**]: Applications and Expert Systems--*games*

General Terms: Design, Human Factors

Additional Key Words and Phrases: End-User Programming, User Interface Software, Programming-by-Demonstration, Programming-by-Example, Application Builders, Inductive Learning, Gamut.

1. INTRODUCTION

Today's software applications are produced by expert programmers. However, much of the content that computer users desire is not provided through programmers. In computer games and educational software, artists, musicians, and educators provide the contents upon which these applications and their users depend. Similarly, only analysts and other business professionals know what is really needed in a financial application. Typically, these people are talented in their field, but are not programmers, yet they often have the knowledge and desire to create useful software. However, the current techniques for building most software requires significant programming ability. Therefore, techniques that reduce the burden of programming and allow nonprogrammers to create software would be useful.

The most common methods for producing software involve programming languages. Traditional languages like FORTRAN and C++ require years of effort to become proficient. Likewise, common libraries for producing graphical software like MFC [Microsoft 1997] and Motif [Ferguson and Brennan 1993] are equally difficult to learn and have been known to confound even expert programmers. Tools such as interface builders that allow a programmer to lay out graphical interfaces using direct manipulation help reduce some of the programming effort. However, even with these tools, developers are still required to program a significant portion of the application. Similarly, authoring tools such as HyperCard [Apple 1993] and Director [Macromedia 1996] can help a developer create some applications without programming. However, these tools tend to limit significantly the kinds of behavior their applications can produce and often require the developer to learn a complicated scripting language in order to produce more sophisticated applications.

The goal of our research is to extend the range of software that can be produced without programming. We wanted to design a tool that could be used by nonprogrammers that was general-purpose and could be use to produce many kinds of behavior. The resulting prototype is a working system called Gamut which is a tool for creating game-like applications. Gamut uses programming-by-demonstration (PBD), where instead of using a language, the developer gives examples of the desired behavior and the system uses learning techniques to infer the correct program. Gamut is essentially a drawing editor with the ability to learn behavior through multiple, demonstrated examples. The developer edits the application's interface to demonstrate how it works. For instance, if the developer wants the application to move a piece in a board game, then the developer would move the piece manually in an example.

The inferencing algorithms in prior PBD systems have been limited and could not create entire applications. For instance, some systems could only infer linear constraints among the numerical properties of its objects. Others could only refer to objects in the interface as literal constants and could not form procedures for describing objects. Finally, some systems possessed no inferencing capabilities at all and used PBD as a method for recording macros that the developer would later have to modify by hand.

The research in this project shows that many of the behaviors that previously required hand modification can be produced using only programming-by-demonstration. Gamut uses new interaction techniques that allow the developer demonstrate examples easily. These techniques let the developer point out important information that the system needs along with an understandable method for providing negative as well as positive examples. Gamut also has more sophisticated inferencing algorithms than previous PBD systems which allow Gamut to generate complex behaviors. It can produce conditional behaviors where the reasons an action occurs can be independent of the action's result. Gamut can also describe objects indirectly by chaining together procedures that describe how objects are related to one another.

In the following sections, we will explain how the developer interacts with Gamut as well as the mechanisms that Gamut uses to infer behavior. We will also describe Gamut's domain and the range of behaviors that it can produce. Finally, we will show how Gamut was used by nonprogrammers when it was tested in a usability study. Gamut has also been described in other publications. The full account can be found in the first author's thesis [McDaniel 1999]. Shorter accounts can also be found in the proceedings of IUI'98 [McDaniel and Myers 1998] which discusses Gamut's inferencing and CHI'99 [McDaniel and Myers 1999] which describes the interaction techniques and the usability study.

2. DOMAIN

Gamut is used to make interactive graphical applications like games. This can include some educational and simulation software that act much like games. Gamut's applications are similar to computerized board games. These applications are two-dimensional and usually consist of a fixed background scene (or board) with graphical pieces that move on top. Figure 1 shows an example game that was created with Gamut. This game is based on the video game, Q*bert. In this version, the player controls the character who can jump on the tops of the cubes in the background. Green balls fall down from the top of the screen and if one hits the character, the player loses a life. If the player collects all the markers on each of the cubes, the player wins. This game can be considered a board game by Gamut's definition. The cubes in the background form the board whereas the character and the balls are the pieces that interact with the board and with each other. Gamut has been used to make several applications including video games like Pacman and shooting games as well as board games such as Parcheesi, Tic-Tac-Toe, Hangman, and others.

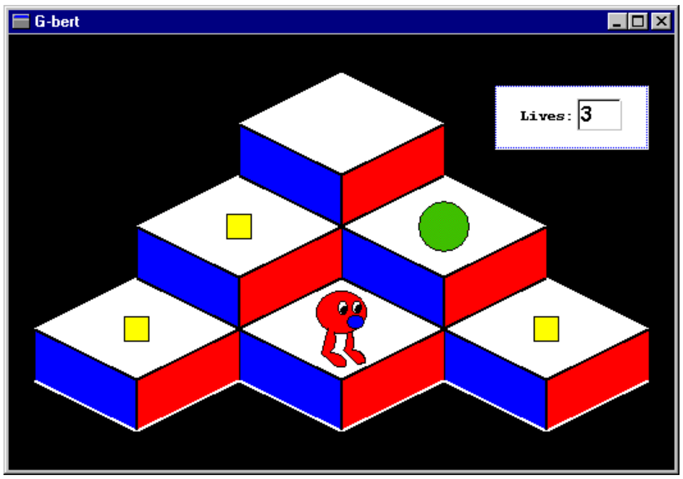


Fig. 1. This Q*bert-like game was created entirely with Gamut. The player moves the character by clicking on the tops of the cubes. Balls drop down from the top and move randomly down the pyramid. This task was also used in the usability study as the optional third task.

The board game domain provides challenges that have not been addressed by other PBD systems. For instance, the behaviors in board games tend to be conditional. Behaviors depend on modes such as the current player's turn. Figure 2 shows a different kind of mode in Pacman. A monster in Pacman can either run towards or away from Pacman depending on whether or not the Pacman has eaten one of the larger "power dots." The decision does not depend on the monster or even the Pacman but on a completely independent factor which in the figure is represented by a toggle switch. (This mode might also be represented by noting the color of the monster.) Gamut is able to incorporate independent objects in its behaviors to produce the dependencies between the pieces and the various modes in the game.

Another challenge is that pieces tend to follow paths and other connections with the board and these paths are not always visible. The system must provide a way for the developer to show the information that is normally hidden when the player sees the game. To be able to use paths and connections, the system must be able to refer to objects indirectly through a chain of descriptions. For instance, as shown in Figure 3, a description for the location where a piece moves in Monopoly might be, "the square that is the dice's number of squares from the current player's location." This one description refers to many different elements of the game including the configuration of the board, the number on the dice, and the mode for the current player's turn. Each element of this

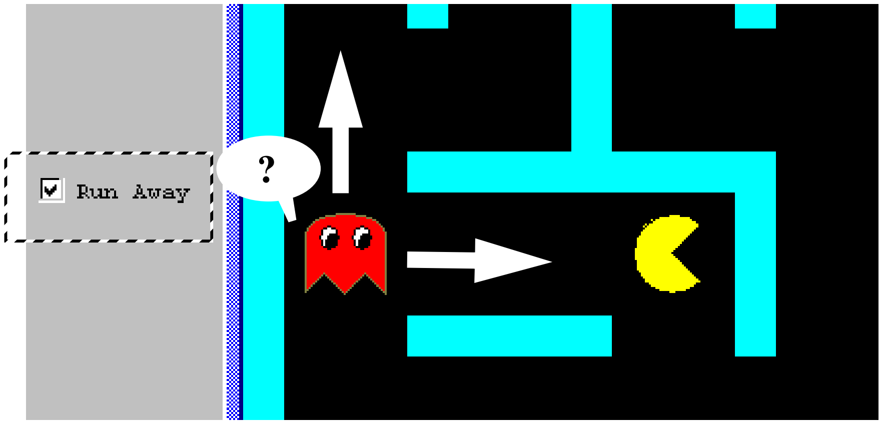


Fig. 2. The monster uses a mode to determine whether or not to chase the Pacman. The monster's decision is independent of the manner in which the monster moves.

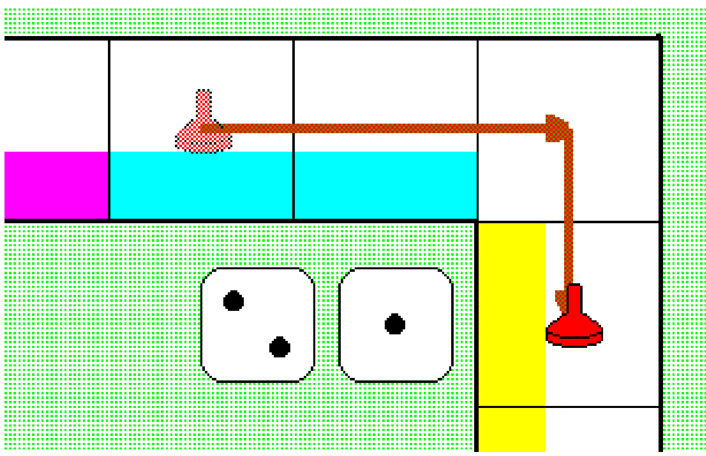


Fig. 3. To describe the location where a piece moves in this board game, the system must produce a chain of relationships. The location depends on the configuration of the board, the number on the dice, and the original position of the piece, and whose turn it is. Each element in this relationship depends on other elements.

description depends on the result of other objects. Gamut is able to chain together multiple descriptions in this way to form more complex descriptions.

Gamut can produce the rule-like behavior found in board games, but it cannot generally create computer opponents. Essentially, the computer can be taught to act as a referee enforcing the rules for human players. Playing some games would require the system to form complex strategies that would be very difficult to infer. Gamut is more like a programming system. The developer must give examples of every behavior in the application and must provide the information that the system uses to perform those behaviors. Some computer opponents are relatively simple to understand. For instance, a monster in Pacman mostly tries to move closer to the Pacman. The information the system needs to move the monster is readily available from the game board. However, the strategy that a Chess opponent might use is significantly more complex. Though it is theoretically possible for the developer to demonstrate a game tree or similar data structure for determining a player's move, it is not generally within Gamut's domain to learn behaviors so complicated.

3. ASSUMPTIONS

In order to function, Gamut's techniques must incorporate some assumptions about the developer and how examples will be presented to the system. The techniques in Gamut are designed to allow the developer to demonstrate examples to the system quickly without having to learn too many new concepts. The goal is to make the process of demonstrating examples simple enough so that the developer will be able to create many examples. This is important since having many examples reduces the significance of any one example. It means that the system need not try to recognize every nuance of every example the developer presents. The system is permitted to ignore elements of prior examples if they did not seem relevant at the time. If an ignored element proves actually to be relevant, then the system can determine that later when the developer provides further examples.

Though it may seem disadvantageous not to revisit old examples, in fact, it is quite useful because the developer demonstrates behavior in Gamut incrementally. This means that the conditions on which a behavior depends might not even exist at the time the developer first demonstrates the behavior. Thus, examples will not always be completely correct. The missing pieces on which an example depends will only be learned as the developer provides examples that conflict with the currently inferred behavior. If Gamut tried to reconcile new examples with past examples, it would likely find more conflicts. Similarly, when the system asked the developer why such a conflict occurred, the developer would lack the proper context for the old example and might not even remember demonstrating it.

It is also assumed that the developer knows how the game actually works. This means that the developer knows all the information that the game uses including the hidden information that the player does not see. The developer must be able to represent that information using graphical objects that Gamut provides. Similarly, the developer must be able give examples showing how the game works. When the system makes a mistake executing a behavior that is not fully formed, the developer should be able to recognize that the mistake has occurred and show how the behavior was supposed to function.

4. CREATING AN EXAMPLE GAME

In order to motivate and present Gamut's various techniques, a small example will be used. The example shown in Figure 4 is a small board game in which a pair of game pieces colored black and gray follow squares around the edge of the board (originally the game used pieces with more lively colors, but since the figures in this publication are black and white, we had to restrict our palette). The players push the "Move" button in the center of the gameboard in order to take their turn. The application will automatically "roll" the die in the center and move the piece the corresponding number of squares. It will also automatically alternate players.

To make this game, the developer will first need to draw its components. The graphics and objects that the developer uses to compose the game will largely determine how the game works. Once the parts are created, the developer demonstrates what happens when the player uses the game's various controls. In this game, the developer has only one control: the button labelled "Move." Thus, the system must learn a single, albeit complex, behavior to make this game work.

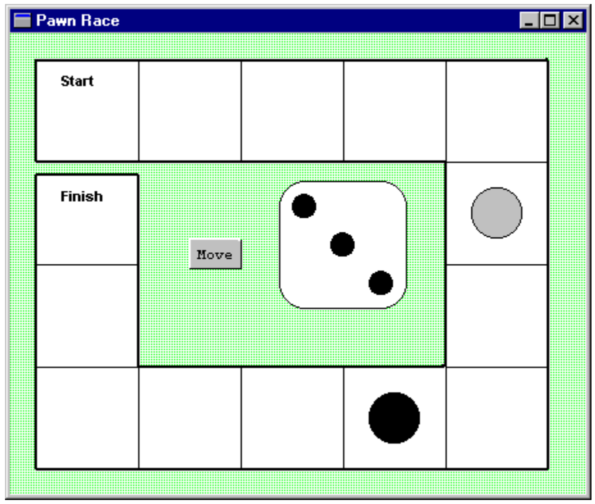


Fig. 4. The example game consists of a rectangular board with a path of squares along the border. Two pieces alternate turns moving the number of spaces shown on the die in the center. The player pushes the "Move" button to make the pieces move. This game was also used in Gamut's usability study as task two.

4.1. The Main Window

The developer will first need to draw the components of the interface. Gamut's main window is mostly a direct manipulation drawing tool as shown in Figure 5. The palettes and menus are used to create objects and manipulate them on the screen. As with other drawing editors, the developer draws object by selecting an item from the object palette and dragging out a region in the drawing area. The colors and fonts of an object can be changed by selecting it and using the appropriate menu command or dialog box.

The region below the drawing area is Gamut's behavior dialog area. The developer uses this dialog to demonstrate behavior and the system uses the dialog to ask the developer questions. The behavior dialog has two parts. It has a text box that the system uses for messages, and on the right is a set of buttons that the developer uses to initiate demonstration and to answer the system's questions.

4.2. Onscreen Guide Objects

The developer draws the background board using lines, rectangles, and other objects. However, the background alone will not be enough information for the system to determine the path that the pieces follow. Gamut would need sophisticated vision-like heuristics in order to partition the background into a path made of squares. The developer could aid Gamut if each square was drawn as a separate rectangle (instead of drawing one large rectangle and dividing it with lines like a ladder). However, the most indicative method for drawing the path is to use arrow lines. Gamut allows the developer to draw graphics in the window that can be made invisible when the game is played. These objects are called "onscreen guide objects" because of the way they *guide* the actions of behaviors. Gamut borrowed the onscreen guide objects technique from DEMO II [Fisher *et al* 1992] where they were called "guidewires." Metamouse [Maulsby 1988] also used a similar technique. Figure 6 shows how the developer might draw the path.

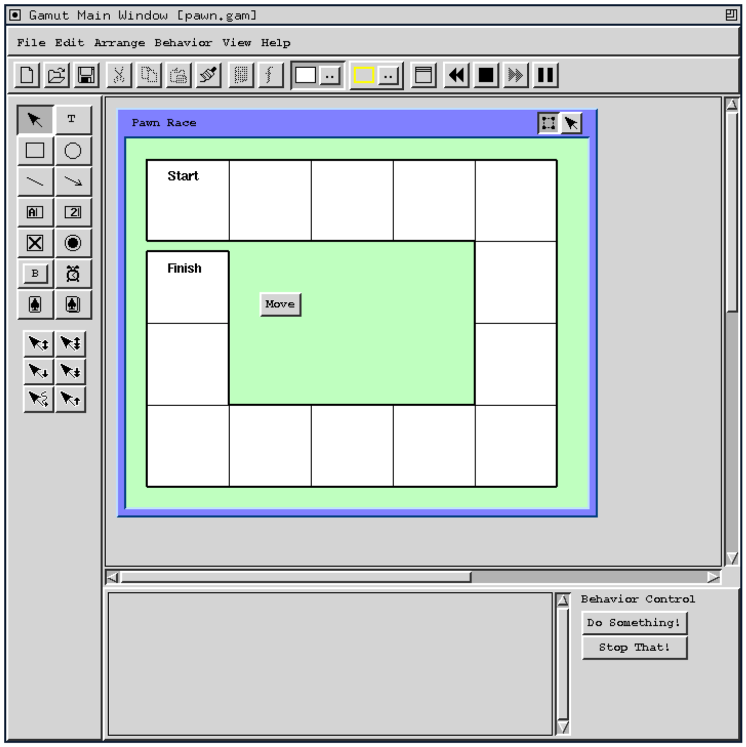


Fig. 5. Gamut's main application window. The initial background for the example game is shown in the drawing area. Surrounding the drawing area are the palettes and menus used to draw and manipulate graphical objects. Below the drawing area is the behavior dialog region.

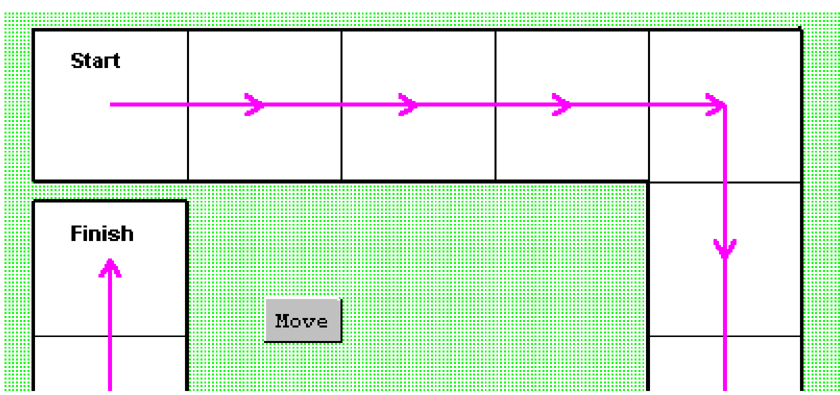
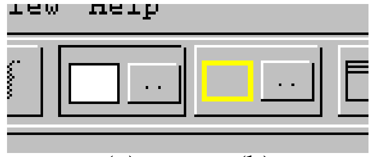


Fig. 6. The developer has drawn arrow lines onto the board to indicate the path that the pieces follow. The arrows are drawn as guide objects so that they can be made invisible when the player sees the game.

The developer specifies onscreen guide objects by drawing them using special colors. Gamut has two color palettes as shown in Figure 7. The palette on the left (a) is used to set normal colors while the one on the right (b) sets the color to be a guide color. When the developer uses guide colors, the objects are drawn in pastel colors with no fill style. The guide objects can be made invisible using a menu command so the player need not see them.



(a) (b)

Fig. 7. Gamut has two color palette indicators in its toolbar. The one on the left is used for standard visible colors and the one of the right is used for guide object colors.

4.3. Offscreen Guide Objects

The developer will also need to create something to represent the current player's turn. The gameboard and the pieces as they currently stand do not show whose turn it is. However, in order for Gamut to move the correct player's piece, it will need to know. To accomplish this, Gamut also allows the developer to create "offscreen guide objects." In Figure 5, one can see that the drawing area contains a smaller window. The objects the developer draws within the window will be visible to the player, but objects outside the window will not and are called offscreen guide objects. These objects are generally used to represent non-graphical information such as counters or modes. In Figure 2, the developer uses a toggle switch to indicate whether a monster should chase or run away from the Pacman. In this example, the developer uses a rectangle as shown in Figure 8. Gamut allows any object to be used as a guide object and not just special-purpose widgets. The developer will use the rectangle's color to indicate whose turn is next. When the rectangle is gray, the gray piece will move and when it is black the black piece will move. Since it is offscreen, developer does not have to use the special guide object color for this rectangle. The developer is using the normal color palette in order to associate the color of the rectangle with the color of the pieces better.

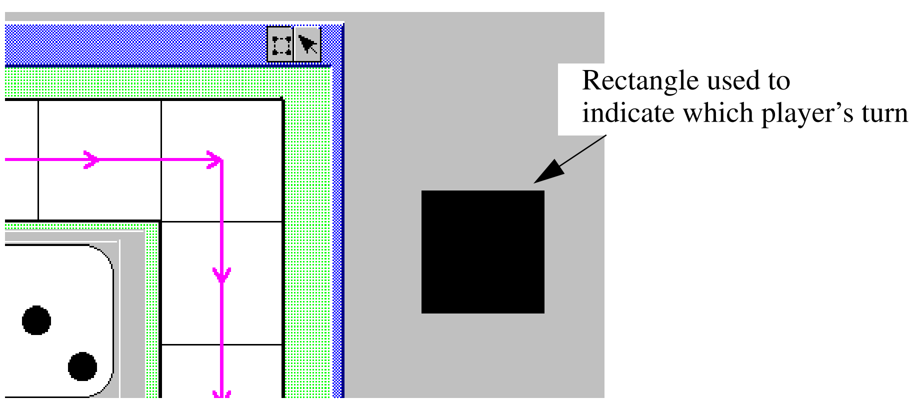


Fig. 8. The developer draws a rectangle in the offscreen region to the right of the board. The developer will use this to indicate whether it is the black or gray piece's turn.

4.4. Representing Hidden Information with Guide Objects

Creating appropriate guide objects for an application is likely the most difficult aspect of creating an application in Gamut. In many ways, creating and manipulating guide objects is like programming. The developer is essentially finding a way a represent an abstract concept using graphical primitives. However, being able to represent this information is key for being able to create full applications.

In Artificial Intelligence terms, guide objects solve the "hidden object problem" which argues that it is virtually impossible to guess what sort of information an algorithm requires without actually having that information present. Some AI algorithms assume the hidden objects take on a specific form. For instance, the algorithm might assume the data is a Min/Max tree which the algorithm then attempts to parameterize. Gamut, however, cannot assume anything about hidden data. The information a behavior needs is highly specific to the application and cannot be guessed easily. Thus, it is imperative that the developer be allowed to make that information explicit. In Gamut, this is accomplished using guide objects.

4.5. Cards and Decks

In order to create the die in the center of the board, the developer will use Gamut's deck and card objects. Gamut defines a "deck of cards" object which is used to hold a set of objects. The deck will only show the one "top" item in the set. This object is ideal for representing the die. The developer creates a "card" object for each face of the die and stores them in a deck which is placed in the center.

Actual cards and decks of cards are common in games and they provide a good abstraction for representing lists of things and random behavior. A deck can be used to represent a set of random effects like the Chance deck in Monopoly. It can also be used to hold unseen information like a set of maps for each level in a game.

4.5.1 Decks. Figure 9 shows an image of the deck that is being used to create the die. Below the top item's image are four buttons that are used to control the deck. The arrow buttons on the right change which item is being shown on top. The "shuffle" button randomizes the deck's contents. The developer will use this feature to "roll" the die. The button on the left is the deck's "lid." When the lid is open, the developer can drag objects in and out of the deck. When the lid is closed, the developer can no longer drag objects in or out manually. This allows the developer to place other objects on top of or behind a deck so that it can be used for layered effects. For instance, if the deck contained a set of boards for a game, the developer would need to be able to put pieces on top of the board without putting them in the deck.

4.5.2 Cards. Though it is not required, the most common objects to put in a deck are cards. The developer can draw objects in a card much like one draws objects in the main window. Figure 10 shows Gamut's card editor. The card editor's palettes and menus are used to create and manipulate objects on the card similarly to the main window. The developer can also drag objects from the main window onto a card and back and can drag objects from one card to another.

Like the main editor, the card has an onscreen and an offscreen area. Objects within the card's viewing region appear on the face of the card object. Objects outside the region cannot be seen. This allows the developer to hide extra information in a card using offscreen guide objects. In Figure 10, the face of the die is drawn in the card's visible area. Below the image, the developer has created a number box widget holding the face's numerical value. By putting the numerical value

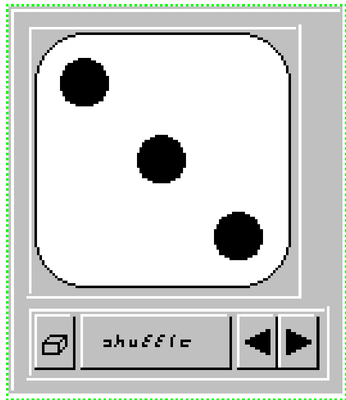


Fig. 9. This deck is being used to create the die object. It contains six items, one for each face of the die. The developer can use the Shuffle button to “roll” the die.

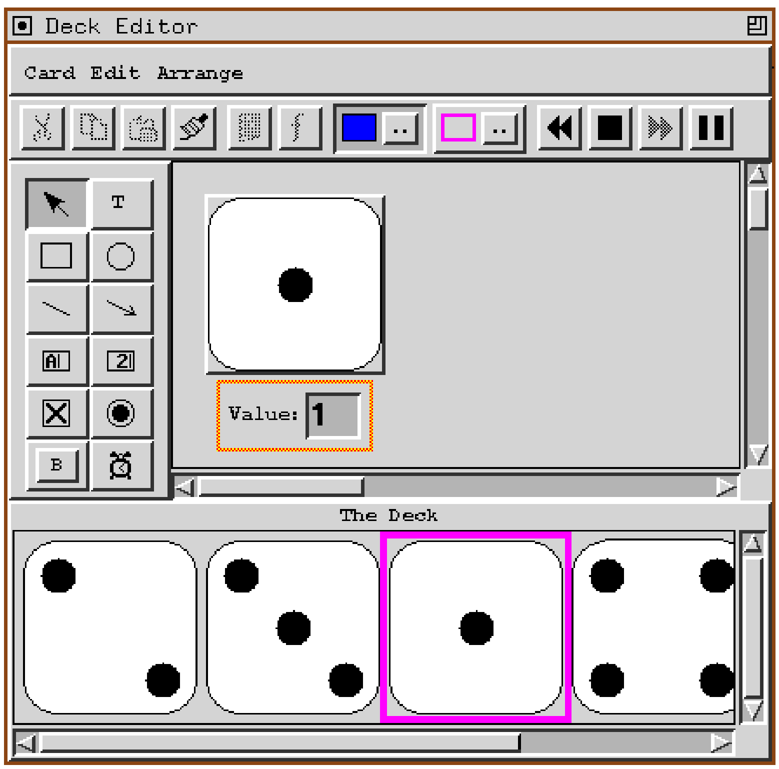


Fig. 10. The card editor is being used to show one of the die’s faces. The other contents of the deck can be seen in the deck editor along the bottom. The image of the die is shown in the card’s onscreen region. The developer has also placed a number box offscreen to indicate the card’s numerical value.

in the card, the developer helped simplify Gamut’s inferencing task. Gamut can find the value of the card directly instead of having to learn to count the number of dots.

The area along the bottom of the card editor is the deck editor which is used to view and select the contents of a deck. In Figure 10, the faces of the die are visible. The deck editor allows the developer to see all the items in the deck. The developer can also click on an item to see and edit its contents. If the item is a card, the card editor will show the card's contents and allow the developer to modify it.

Gamut's card abstraction is not the same as the card abstraction in HyperCard [Apple 1993]. Gamut's cards are based on playing cards and are individual objects that the developer creates and manipulates in the window. A card can be moved from the window into a deck and out again. On the other hand, a card in HyperCard is more like a window or a screenshot of the application. The HyperCard application contains a set of these screenshots which it calls a "stack." The developer creates clickable areas within each screenshot that allow the user to change from one screenshot to another. Gamut's cards are more like graphical objects. The developer may draw whatever is desired on a card, and there is no direct relationship between a card and the application window.

After the developer creates the die object, the graphical objects in this application will be complete. At this point, the developer will have to add behaviors to the application to make it actually work.

5. DEMONSTRATING BEHAVIOR

Gamut allows the developer to create behaviors incrementally. The developer need not demonstrate the entire behavior all at the same time, but can build up a behavior in pieces. Continuing with the example, the developer will demonstrate how the "Move" button causes the pieces to move in the game. The developer will start by demonstrating how to roll the die and move just the black piece. Then, the developer will incorporate the turn indicator and the gray piece in order to complete the behavior.

5.1. Nudges

In order to demonstrate behavior, the developer must provide the system with examples of how the game works. Gamut's method for demonstrating examples is called "nudges." With nudges, the developer always tests the game's behavior. If the behavior performs the wrong actions, the developer corrects the system by showing what it should have done, instead. The developer can correct the system by either telling it to "Do Something" or "Stop That." Do Something is used to demonstrate new behavior whereas Stop That is used to prevent incorrect behavior. Do Something and Stop That nudges are produced by pushing their corresponding buttons in the behavior dialog (see Figure 5).

In the example game, the developer wants to add behavior to the "Move" button. So, the developer begins by pushing the Move button. The system has not yet seen any examples so it does not know to move the game piece. So, the developer pushes the Do Something button. Pushing Do Something or Stop That puts Gamut into *Response mode* where it treats the developers edits as an example. The developer will demonstrate the behavior by performing the actions the system is meant to do. When the example is complete, the developer will leave Response mode by pressing the "Done" button.

5.2. Temporal Ghosts

Demonstrating the example, the developer pushes the shuffle button on the die's deck object. The die produces the number two. The developer then moves a piece two spaces along the board. When the developer performs these actions, Gamut leaves behind *temporal ghosts* of the manipulated objects. Temporal ghosts are seen as dimmed transparent copies of objects in their previous state as shown in Figure 11. For example, when the developer moves the piece, Gamut marks the

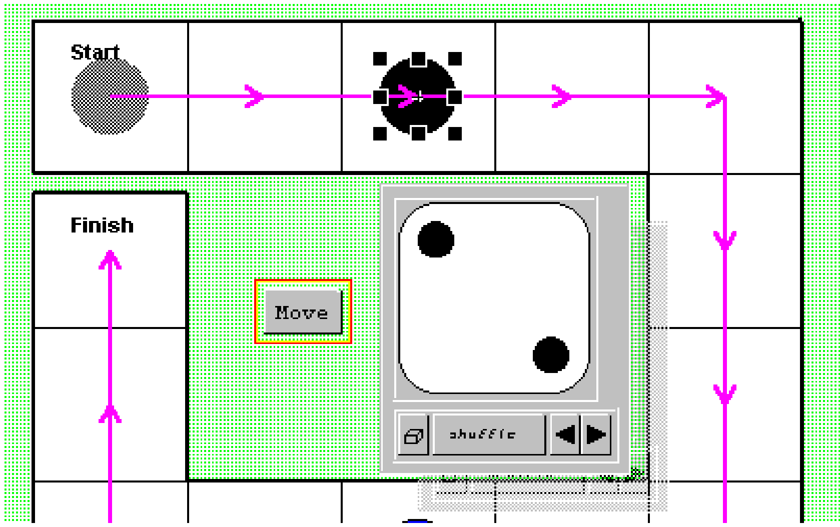


Fig. 11. The developer has shuffled the deck and moved the black piece two spaces. The ghost of the piece shows its original location and the ghost of the deck shows that it has been shuffled. (The gray piece has been moved out of the way.)

piece's original position with a ghost object. For objects that do not move (such as the deck), Gamut will put the ghost in an offset position behind the original object.

Temporal ghost objects are useful for showing which objects were modified in a behavior. The developer can see which objects have changed by noting which ones have ghost objects on the screen. Ghost objects are also used for referring to an object in its previous state. For instance, if the piece was supposed to move the number of spaces shown on the die *before* the deck is shuffled, the developer could refer to the ghost of the die instead of the die itself. In this case, the behavior uses the shuffled value so the ghost of the die's deck is not needed.

5.3. Hint Highlighting

Gamut allows the developer to point out important objects that are used in a behavior by highlighting them. The developer highlights an object by clicking on it with the right mouse button. (Normal selection is handled with the left mouse button.) The developer is allowed to highlight objects at any time during response mode and is expected to highlight objects when the system asks a question. Gamut will try to use any highlighted objects when it describes the behavior presented in the example.

In the board game, the developer decides to highlight the path that the piece followed in order to show that it is important as shown in Figure 12. Highlighted objects appear outlined with a yellow-green rectangle. At this point, the developer has completed the example so the developer exits Response mode by pushing the "Done" button (see Figure 13). The highlighted path will tell Gamut that it is important, and Gamut will create a behavior where the pieces follows the path as its first attempt. If the developer chose not to highlight the path, the system would assume that the piece is moved to a constant location.

Highlighting during an example is always optional and Gamut can ignore highlighted objects when it cannot determine any way to use them. Though the system may not determine the correct

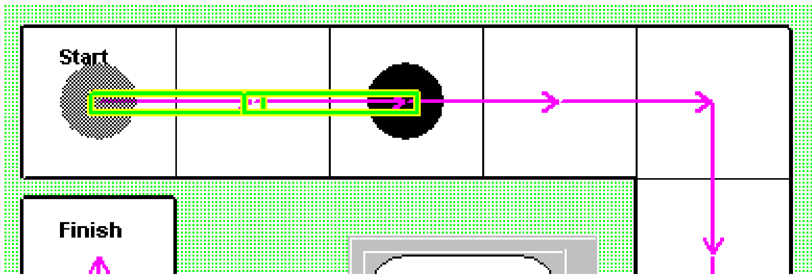


Fig. 12. The developer highlights the arrow lines that the piece followed. Gamut will incorporate these objects into the behavior it infers.

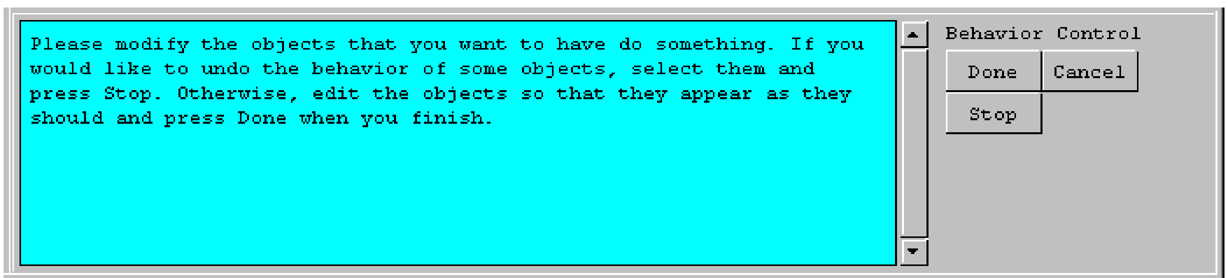


Fig. 13. During Response mode, Gamut asks the developer to complete the example using text in the behavior dialog area. The developer completes the example by pushing the Done button.

behavior immediately, later examples will show when a behavior depends on more than is currently known.

Gamut's highlighting technique is kept separate from normal selection because sometimes the developer will want to manipulate objects after some are highlighted. Also, since Gamut will try to use the objects in the behaviors it creates, it is important that the developer only highlight the objects that actually matter. With normal selection, the developer can "lasso" multiple objects in order to select them all at once. Though the developer can highlight as many objects as desired, only one object can be highlighted at a time. This helps prevent the developer from highlighting the entire game board which would significantly slow and otherwise impede the inferencing algorithms.

Hint highlighting is also used to answer questions that the system asks. When Gamut finds a contradiction or a behavior it cannot explain, it asks the developer a question. Usually this question asks why something has changed and what that behavior depends on. The developer is asked to highlight the relevant object that shows when the different behaviors occur.

Though it is important that the developer eventually highlight the correct objects, Gamut does not fail if the wrong object is highlighted initially. Gamut will continue accepting examples until it determines that the highlighted objects cannot be used to explain the behavior in the application. At that point it will ask another question, which will hopefully elicit a better response. For instance, in the game example, the developer has highlighted the path but has neglected to highlight the die. When the developer presents a new example, Gamut will have to ask a question to determine the missing information.

5.4. Using Stop That

For the second example, the developer once again pushes the “Move” button to see what it will do. Since the developer highlighted the path during the last example, Gamut will make the piece follow the path. However, this time the die rolls three, but Gamut will move the piece two spaces as the developer demonstrated in the last example. To correct this, the developer selects the piece and pushes the Stop That button. Using Stop That will automatically undo any actions the behavior performed on the selected objects before the system transfers to Response mode. Gamut moves the piece back to the position where it started. The developer then moves the piece manually to the correct position as shown in Figure 14 and pushes the Done button.

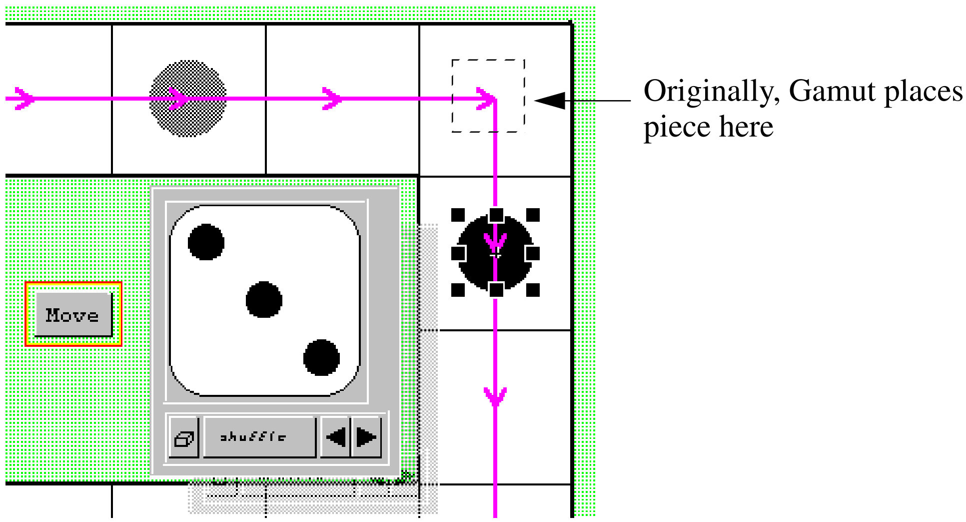


Fig. 14. At first, Gamut incorrectly moves the piece two spaces instead of three. The developer selects the piece and presses Stop That which causes Gamut to move the piece back to its original position (where the ghost is). The developer then moves the piece to the correct position and presses the Done button.

5.5. Stop That and Negative Examples

Using Stop That is an understandable method for generating negative examples. Examples used for inductive learning can be positive or negative. A positive example is one that fits the concept to be learned whereas a negative example is similar to the concept but is not a member. A system must use negative examples in order to learn concepts with multiple disjunctive clauses (in other words, negative examples are needed to learn “Or”-like concepts). Roughly, Do Something and Stop That correspond to generating positive and negative examples. Do Something tells the system what it ought to do and Stop That tells it what not to do.

The nudges technique is an improvement over demonstration techniques that use a separate stimulus recording phase such as an extended macro recorder used in PBD systems such as Frank’s Grizzly Bear [Frank 1995]. An extended macro recorder is similar to a typical macro recorder except that it has an added stimulus recording phase. When the developer initiates demonstration, the recorder asks for the developer to demonstrate the stimulus event. For instance, the developer might initiate recording and then click the mouse to demonstrate the stimulus. Then, the recorder switches phases to record the response of the behavior. This phase acts like a standard macro recorder and records the actions the developer makes as a procedure. In order to demonstrate a negative example using this technique, the recorder interface will often have special switches to tell it to record a negative example. When the developer selects negative example mode, he or she would demonstrate the stimulus normally but does not record a response (because there would not be one).

Since the developer would have to select a switch in order show a negative example with an extended macro recorder, the developer would need to know beforehand when a negative example is required. Using Stop That, on the other hand, does not require the developer to know when a negative example is required or even what a negative example is. Negative examples arise through the system misinterpreting the behavior and having the developer correct its mistakes. As will be shown below, during Gamut's usability study, participant's were able to use Stop That in order to generate negative examples, whereas studies of macro recorders [Frank 1995] showed significant difficulties with participants understanding negative examples.

5.6. Asking a Question

Returning to the current demonstration, the developer has just shown a new example that contradicts what the system currently knows about the behavior. When the system sees that a behavior has changed but does not know why, it asks the developer a question by putting text in the behavior dialog area.

Gamut's questions always ask about the state on which a behavior depends. Gamut searches for the most specific value that it can change in order to correct the behavior. Through its matching algorithm, Gamut is able to determine that the length of the path the piece is following has changed from following two path links to following three. Gamut's matching algorithm is discussed below in the inferencing section. It presents the question shown in Figure 15 that asks on what the length of the path depends.

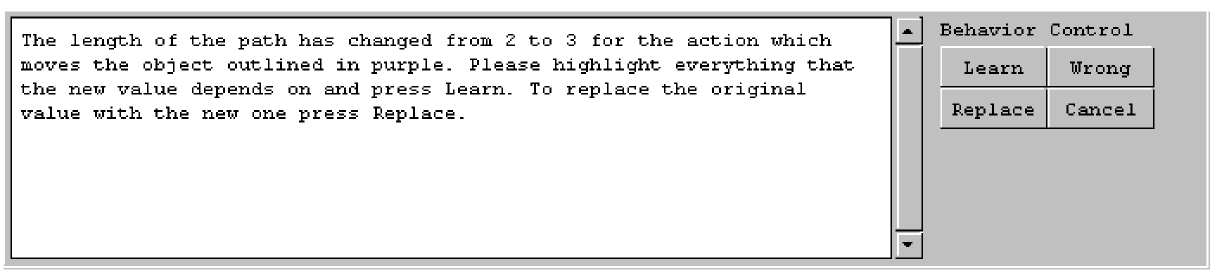


Fig. 15. Gamut's question asks why the length of the path has changed from two to three.

Beside the question text, Gamut replaces the Do Something and Stop That buttons with new buttons that developer uses to answer the question. The developer can choose to answer in three ways using either the Learn, Replace, or Wrong buttons. (The developer can also use "Cancel" to exit Response mode without creating a behavior.)

5.6.1 Learn. The typical way to answer Gamut's question is to highlight the object that the system needs to know and press the Learn button. In the example game, the developer wants the system to incorporate the number on the die, so the developer highlights the die object and presses Learn.

5.6.2 Replace. The Replace button causes Gamut to replace the old value in the question directly with the new value without generating a new expression. The developer typically uses

Replace to correct mistakes in previous examples. For instance, in the example game, the developer would push Replace if he or she always wanted the piece to move three spaces instead of two. The developer can also use Replace to exchange temporary objects. For instance, the developer might substitute a rectangle for a monster object to train the system how the monster should move. Later when the developer draws the actual monster object, he or she can use Replace to make the system move the proper object.

5.6.3 Wrong. Gamut's matching algorithm is based partly on heuristics that can sometimes be wrong. This can cause Gamut to generate a spurious question that does not correlate to how the developer understands a behavior. The developer uses the Wrong button to make Gamut skip odd questions and ask something else that will hopefully make more sense.

5.7. Making the Pieces Alternate

At this point, the developer has trained the black piece to move the number of spaces shown on the die. Now, the developer wants to make both pieces move. The developer wants the black piece to move when the player turn rectangle is black and the gray piece to move when it is gray. Since the developer has not yet demonstrated an example that used the offscreen rectangle, Gamut has not incorporated it into the behavior. This allows the developer to create a behavior incrementally incorporating new objects and actions into a behavior as the developer wants.

The rectangle is initially black, and the developer has been using the black piece in the previous examples. When the developer pushes the Move button, the system moves the black piece as is expected, but the developer also wants the system to change the color of the turn rectangle to gray. The developer pushes the Do Something button, changes the rectangle's color to gray and presses Done. Gamut will add a new action to the behavior that changes the rectangle's color to gray.

For the next example, the developer once again pushes the Move button. Not knowing any better, Gamut still moves the black piece. Gamut also changes the rectangle to gray, but since it is already gray the only indicator of the change is a second gray ghost object behind the rectangle as shown in Figure 16a. The developer selects the black piece and pushes Stop That which causes the system to return the black piece back to its original position. The developer then moves the gray piece to the correct position and changes the color of the turn rectangle back to black as shown in Figure 16b. The example is complete, so the developer pushes the Done button.

Gamut sees that this example causes the behavior to change in two ways. Gamut will ask two separate questions about these changes. In the first question, the system asks why the gray piece moved instead of the black one. The system was able to match the position of the gray piece with the description that was originally used on the black piece thus the system does not have to learn a new method for moving gray pieces. The answer to why the gray piece is moved is because the turn rectangle was gray at the beginning of the turn. The best object to highlight is the ghost of the turn rectangle which still shows the gray color. The second question asks why the color of the rectangle changed from gray to black. It turns out that the ghost of the rectangle is also a good way to determine this information as well. For both questions, the developer highlights the ghost of the offscreen rectangle and presses the Learn button.

5.8. Learning Modal Behaviors

The ability to make the pieces alternate requires Gamut to learn a modal behavior. The key to building modal behaviors is the ability to create guide objects that store crucial state information

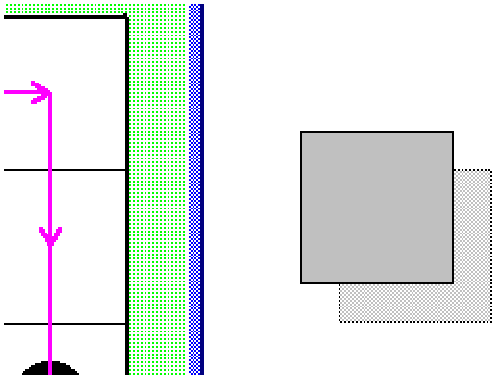


Fig. 16a. Gamut sets the color of the rectangle to gray, but since the rectangle is already gray, only the presence of the ghost indicates a change.

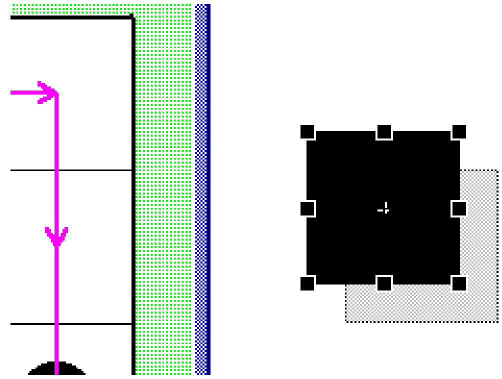


Fig. 16b. The developer sets the color of the rectangle back to black. The ghost remains gray.

and the ability to give hints that point out these objects at appropriate moments. Other systems that do not have guide objects or hints have not been able to learn complex, if-then-like statements through demonstration.

6. OTHER INTERACTION TECHNIQUES

Though they are not used in this example game, Gamut provides some other interaction techniques that are useful for creating game-like applications. These include *player mouse icons* for demonstrating mouse input and *timer and history controls* for controlling timers and moving back and forth in the game's history. Also, some of Gamut's techniques can be used for other purposes. For example, the window frame can be used for creating moving backgrounds, and the Replace button in Gamut's behavior dialog which can be used to defer questions.

6.1. Player Mouse Icons

Gamut provides a new technique for demonstrating player input with the mouse. In order to demonstrate a mouse event, the developer selects the event's icon from a palette and drops it on the screen where the event is supposed to occur. Using the mouse icons allows the developer to demonstrate player input without having to switch modes as is necessary in other systems.

The icons representing the various mouse events are shown in Figure 17 and the palette of icons can been seen beneath the object palette in Figure 5. The icons were borrowed from Marquise [Myers *et al* 1993] and consist of an arrow that looks like the mouse cursor with a smaller arrow beside it that shows which event it represents. The click and double-click icons have arrows that point both up and down whereas the down and double-down arrows only point down. There is also an icon for mouse up with an arrow pointing up. The icon that is used for both mouse move and drag has a small wavy arrow.

In most PBD systems, the developer is expected to perform a mouse event by actually using the mouse. Normally, this requires the developer to switch to a special mode for demonstrating the stimulus event. However, some mouse events, like demonstrating a mouse down (but not up), are difficult to show because exiting stimulus mode usually requires the developer to use the mouse. Marquise handled this situation by having the developer use function keys to transfer between modes. Frank's Grizzly Bear [Frank 1995] used a timer that automatically switched modes when

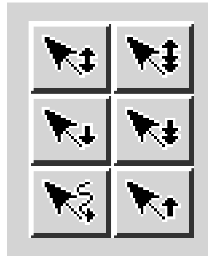


Fig. 17. The player mouse input icons. These are used to generate player mouse events without having to switch out of editing mode.

it expired. Gamut's mouse icons, however, eliminate the need for special stimulus modes. The developer can drop mouse icons while still being able to edit objects normally.

Gamut's mouse icons also help eliminate the need to have an explicit Run/Build mode switch. Though it is possible for the developer to switch modes so that the mouse operates directly in the window, there is no need to switch to this mode while the developer is demonstrating behavior. This reduces the number of modes with which the developer must contend while creating an example.

6.2. Timers

Gamut uses timers to create events for monsters or other kinds of autonomous behavior. A timer acts like a button that automatically presses itself. The developer demonstrates behavior for a timer by using its controls as shown in Figure 18. The leftmost button produces a single tick event. The developer presses Do Something or Stop That after pressing the tick button to train the timer's behavior. The other two buttons are used to stop and start the timer. The developer can also set the timer's duration (not shown). To create multiple repeating behaviors, the developer can create as many timers as needed. Gamut's timer widget is similar to timers found in other toolkits like Visual Basic [Microsoft 1993].

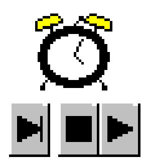


Fig. 18. The timer widget is used to create automatically repeating behaviors.

6.3. Timer and History Controls

Gamut provides a set of buttons that provide a global control for timers and a means for examining the application's history. These buttons are shown in Figure 19. Reading the buttons from left to right, the first and third buttons, which are marked with arrows, control the game's history. The second and fourth buttons, which are marked with stop and pause symbols, control timers.

Gamut's history controls allow the developer to move back and forth in the application's state as it is being tested. By being able to revisit old states, the developer can move back in a game's history. This allows the developer to find the state when a bug first occurred and then create a cor-

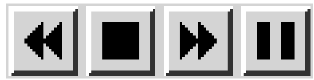


Fig. 19. The history control buttons are located on the toolbar. The buttons marked with arrows are used to move forward and backward in the application history. The stop and pause buttons are used to control timers.

recting example at that point. It also allows the developer to quickly restore the game to a common state when the developer wants to demonstrate a behavior that performs multiple actions. For example, if the developer wanted to create a behavior that performed two behaviors, the developer could train the first behavior then back up to the state before the first behavior occurred and then demonstrate the second behavior.

The global timer controls allow the developer to perform stop and pause operations on all timers at once. This can be helpful when there are many timers or when some are hidden in cards or behind other objects. The stop button will turn off all the timers so that they will not generate further events. In order to reactivate a stopped timer, the developer needs to find the timer and start it again (a demonstrated behavior can also restart a timer). The pause button will also stop timers, but unlike stop it acts as a toggle switch. When pause is on, all timers stop but are not deactivated. When pause is turned off, all the still active timers will resume generating events.

6.4. Window Frame

As previously mentioned, Gamut's application window divides the screen into a visible and offscreen area. Normally the offscreen area remains offscreen, but Gamut allows the developer to move the window if desired. To the player, the window always remains stationary. If the application window moves, it appears to the player as though the contents of the window have shifted. Gamut also allows the visible area of a card to be moved in a similar manner.

For instance, if the developer wants the background to be a large, scrolling map, the map would be drawn full-size in the background of the main window. The application frame would be positioned only over the part of the map that the player can see. When the player's character moves, the application frame follows the character. This would cause the character to appear stationary to the player while the map scrolls behind. As another example, if the developer wanted to scroll a long paragraph of text, the entire text might be written on a card. The card's viewing area would be positioned over the part that the player could see at any point in time and moved down as the player read. The text would appear on the card object which could then be placed in the application window as part of the game.

6.5. Using Replace to Defer Questions

The Replace button has another purpose besides fixing mistakes as discussed previously in Section 5.6.2. Consider demonstrating a behavior that occurs when one piece lands on another in the example application. Suppose that when a piece lands on another, the developer wants the landed on piece to go back to the starting square. In the first example, the black piece lands on the gray piece, so the developer uses Do Something and demonstrates that the gray piece goes back to start. At this point, Gamut only knows about moving the gray piece and does not when to apply that behavior so it applies it all the time. In the second example, the gray piece jumps back to start even though the black piece did not land on it. The developer selects the gray piece and presses

Stop That putting the gray piece back in its correct position. The system will see an ambiguity at this point as ask the developer why the gray piece did not move at which point the developer is in a bind.

Gamut can only learn true conditions. For instance, it can test whether two objects intersect, but if the two objects do *not* intersect at the time the developer highlights them, then Gamut would not know that the objects ever intersect and would not create that test. This is what happens when Gamut asks a question at the wrong time. The reason the gray piece moves in the above example is because it intersected with the black piece, but the system asks the developer to highlight objects at a time when the pieces do not intersect. The system cannot anticipate the correct time to ask questions so this problem cannot be prevented. The developer may choose to highlight the objects anyway in which case the system will not create the needed predicate and it will ask the same question later. Or, the developer may recognize that the objects are not in the proper configuration and defer the question using Replace. With Replace, the system will still have to repeat the question later but it will not create the relationships it would if the developer highlighted something. Though this seems like an advanced technique, two of the four usability study participants used it correctly to demonstrate their applications.

7. INFERENCING

Most previous PBD systems have used simple algorithms to infer behavior, but these are not sufficient to handle the behaviors needed by complete applications. There are a rich variety of algorithms developed by AI researchers and Gamut adapts two of them. Gamut uses *heuristic search* as the backbone of its inferencing. Heuristic search learns a behavior quickly because the rules encode useful features of the domain. Gamut also uses *decision tree learning* [Mitchell 1997] which is described below, but Gamut does not use the decision tree algorithm to produce code directly. Gamut encodes if-then-like expressions that use a boolean test using decision trees.

One of Gamut's key abilities is the ability to revise an inferred behavior with new examples. To better explain how this occurs, this section will show how Gamut infers one example from the board game sample application described above. Suppose, the developer has just finished the first example which created a behavior that moved the black piece two spaces every time. The developer then demonstrates the second example which incorporates the value of the die into the movement. The section will show how this example is interpreted.

7.1. Internal Representation

Gamut uses three kinds of objects to represent application behavior: *events*, *actions*, and *descriptions*. These components are similar to the language invented by Halbert in his SmallStar system [Halbert 1984]. An *event* object is used to represent player input and editing commands that the developer produces. Gamut augments events by adding *actions* demonstrated by the developer, and then events become *behaviors*. Actions represent the parts of a behavior that change the state of the application. Gamut defines seven actions which are listed in Table I.

The parameters of actions are called *descriptions* which are named after Halbert's "data descriptions." A description can be a constant value like a number or a color, or it can be an object which represents a short, composable expression that can be evaluated. Description objects can have parameters that themselves are descriptions. For instance, "the color of the object to which the arrow points" is a Get Property description with one parameter referring to the constant property, "color," and another parameter that contains a Connect description that refers to the arrow.

Table I: List of all actions defined in Gamut.

Create Object: Creates one or more graphical objects from a prototype.

Delete Object: Removes graphical objects from their window.

Move/Grow: Moves or resizes graphical objects.

Change Property: Changes one property of one or more graphical objects.

Reparent: Moves one or more graphical objects from one window or group to another.

Shuffle: Randomizes the contents of a deck.

Change Z-Order: Changes the stacking order of one or more graphical objects (like To Top or To Bottom).

Table II: List of several descriptions implemented in Gamut.

Get Property: Returns the value of a single property from a graphical object.

Align: Returns a graphical location based on connections to other objects and locations.

Select Object: Picks one or more objects from a set of objects according to a boolean expression.

Choice: Picks one value from a set of several values according to a boolean expression. This is Gamut's form for an if-then or case statement.

Equal: Tests if two values are equal.

Number Test: Tests two numerical values for equality, lesser, or greater than.

Connect: Given a location, return the objects that are connected to that location.

Chain: A Connect that is repeated an integer number of times and returns the last set of objects.

Count Objects: Returns the number of objects in a set.

Objects From Group: Given a grouping object, like a card, deck, or window, return the objects within that group.

Top Card: Return the top item in a deck.

Add: Returns the sum of two numerical values.

Get Prototype: Returns the base type of a graphical object.

Default Object: Return the object parameter of the action this description is contained within.

Descriptions have two significant methods. One is a computation method which evaluates the description's expression. The other is the "recursive difference" method which is used for revising the description. Gamut defines 22 descriptions, some of which are listed in Table II. The others are described in the thesis [McDaniel 1999].

7.2. Example Behavior

When the developer finished the first example in the board game, Gamut created the behavior shown in Figure 20. It consists of two actions. The Shuffle action (a) shuffles the contents of the die's deck object and the Move/Grow action (b) moves the black piece. The position of the piece is determined by the descriptions that are attached to the Move/Grow action's value parameter. There are two descriptions. The first is an Align description (1) which shows how to attach the black piece to the end of an arrow line. The second is a Chain description (2) which counts the number of arrows to follow starting from the original location of the piece. The length parameter of the Chain description is currently set to the constant value of two. When Gamut examines the new example, it will find that the length is not constant and will need to be replaced with a new description. The “Default Object” description (3) refers back to the black piece object that the Move/Grow action (b) holds in its object parameter. Actions that modify an object often refer to that object in its descriptions and the Default Object description provides a convenient way to do so.

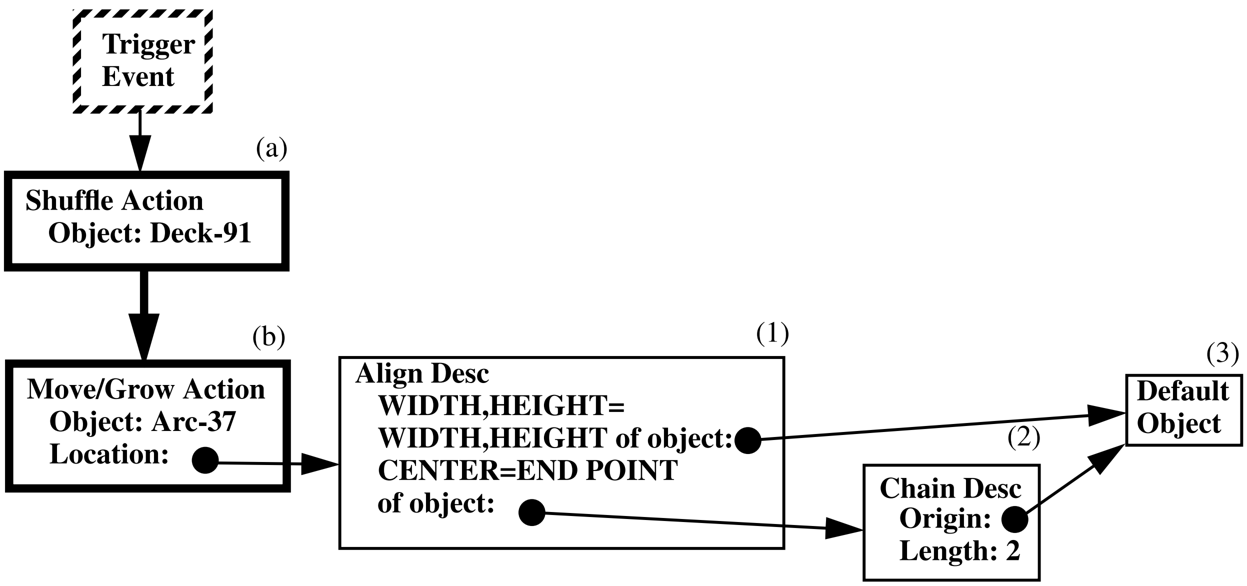


Fig. 20. The Behavior after the developer’s first example in the board game application. The behavior’s event is a Trigger that is controlled by the Move button. The two actions shuffle the die’s deck and move the piece two spaces.

Gamut was able to create this complicated behavior because it found ways to incorporate the objects that the developer highlighted in the first example. Recall that the developer highlighted the two arrow lines that the piece followed. Gamut could recognize that the black piece was connected to the end of an arrow line so it created the Align description, and then it could follow the arrows to determine that it could create the Chain description. The algorithm that Gamut uses to create new descriptions will be discussed below.

7.3. The Difference Between Events and Actions

Both events and actions are derived from the Amulet “command object” [Myers and Kosbie 1996] data type. A command object represents a command such as one might find in the menubar. For instance, the standard cut, copy, and paste editing operations are implemented using command objects. Command objects are stored in widgets and input objects (called “interactors”) and perform the work that the widget or interactor performs. In some systems, a command object would be likened to a “callback procedure” except that when a command object is executed, it stores state information within itself and is then copied to the undo history. The command in the history uses the state to implement Undo and Redo for the operation it performs.

In Gamut, all commands that the developer can use in a demonstration are represented with events. Thus, the commands on widgets and interactors that the developer uses to manipulate objects in an example use event objects. Interactions that the player may cause also must be recognized during a demonstration so they are also represented by events. This includes events for mouse input and an event for recognizing when a button is pushed.

Actions, on the other hand, are commands that Gamut creates dynamically when it learns new behaviors. Gamut can store actions in any event, especially the events that correspond to player input. For instance, Gamut can put a Move/Grow action into a button's event which will cause the button to perform the Move/Grow. In other words, events are permanent fixtures that are part of the objects and widgets in the application whereas actions are dynamically created objects that are stored within events.

In some systems, actions and events are the same entity. Though this is not necessarily a problem, Gamut keeps actions and events separate for two reasons. First, Gamut's set of actions is designed so that each kind of action is mutually independent. Each action affects a different kind of data. This helps Gamut categorize a set of actions and determine which ones affected which objects. Second, actions may be stored in events but not in other actions. This prevents there from being a hierarchy of actions in a behavior which would be more difficult to parse.

7.4. The Stages of Inferencing

The stages in Gamut's inferencing are shown in Figure 21. In the first stage, actions from the original behavior and the developer's manipulations are converted into a new action list. The second stage compares the new actions with the actions in the original behavior and generates a data structure that captures the differences between the two. In the final stage, the differences found in stage two are resolved to create a new behavior.

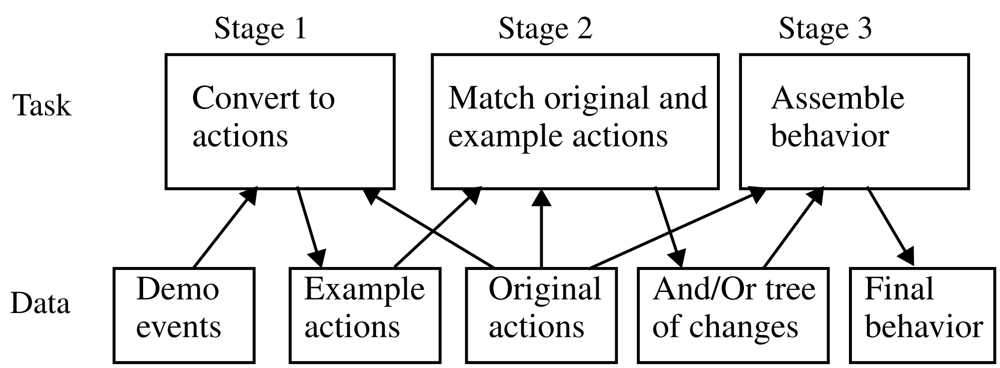


Fig. 21. The stages Gamut uses to create new behavior.

7.5. Stage One

The first step converts the developer's response into a list of actions. Gamut reads the undo history to extract the new example as a list of events. The Do Something and Stop That commands along with the dialog's Done button form the boundaries. If there was an original behavior, its actions are added to the list. Many graphical operations are complicated. For example, grouping objects creates a new group object, changes the selected objects' positions, and reparents the selected objects into the group. Gamut therefore translates the events into more primitive actions to make them easier to manipulate.

In the developer's example, the black piece is being corrected to move three spaces instead of two. The developer first pushed the Move button which caused the system to move the piece two spaces. The developer then selected the piece, pushed the Stop That button, and then moved the piece the correct distance. Finally, the developer pushed the Done button because the example was complete. This series of events was recorded in Gamut's history list as shown in Figure 22. The history list is a queue and is shown listed in chronological order from left to right (which puts the head of the queue on the right). The event for the Move button contains the actions that shuffled the die's deck and initially moved the piece. This event also points back to the original behavior in the Move button that Gamut will have to modify. Since the developer had selected the piece, the Stop That button performed an undo on the piece's Move/Grow action and contains a reference to that action. The developer's correction which moved the piece three spaces is shown as a Move/Grow event which occurs after Stop That.

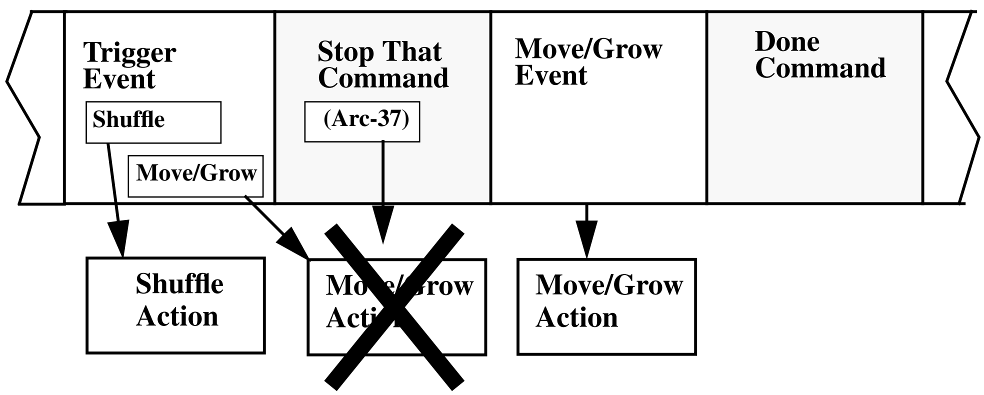


Fig. 22. History list showing the developer's example. The example begins with the trigger event representing the Move buttonpress. Then, the Stop That command appears followed by the developer's correction to the piece. The example ends with the Done button being pushed. Converting the example from the events in the history list into actions involves retrieving actions from the original behavior, converting the developer's edits into actions, and merging them together along with any editing caused by Stop That.

In this case, converting the developer's events into actions is straightforward as shown in Figure 22. The developer's Move/Grow event translates directly into a Move/Grow action. Gamut will also retrieve the actions from the Move event and remove any actions that are eliminated by Stop That. Gamut then combines these actions into a final set, which in this case, is a Shuffle action and a Move/Grow action; however, the final position of the Move/Grow action has changed to a different location.

7.6. Stage Two

In this stage, Gamut compares the newly formed example actions with the actions found in the original stimulus event. This stage performs the bulk of the inferencing work. There are two parts to this stage. The first part matches the actions in the new example to the original actions. This will determine which changes were made to the original actions' parameters, which original actions did not execute, and which example actions are new. Of course, if there are no actions in the original stimulus, stage two is skipped and the system moves to stage three.

The second part of stage two propagates the changes found in the original actions' parameters to the description objects contained within. Each description determines how its parameters change using its *difference* method. This is recursively repeated for the whole tree of descriptions. The final result of this stage is a set of changes one might make to the parameters of the original actions and descriptions in order to make them perform the same operations as the actions in the example. The algorithm Gamut is using is a heuristic search, and the object-oriented design provides a good framework to apply heuristics efficiently and allows extensibility.

7.6.1 Matching Actions. Here Gamut matches actions in the new example to actions in the original behavior. Gamut's actions were intentionally designed to be unordered. Most systems use temporal order to represent constraints between actions. The result of one action feeds into the next to provide a combined result. Instead, Gamut uses explicit descriptions where the result of another action is described fully by the action that actually needs it. Whether the actions execute before or after each other does not matter. Gamut prevents code duplication by sharing common descriptions. By being unordered, actions can be easily matched by looping through all the actions and comparing them. In our experience with Gamut, behaviors contain very few actions (usually two or three), so the time to compare them is insignificant.

Two actions are considered *identical* if all of their parameters match. If all but one parameter matches, the actions are considered *similar*. Identical actions are ignored since they already do what is required. Similar actions are recorded as having changed the one parameter. Unmatched actions from the new example are considered to be new, and unmatched actions from the original behavior are considered not to have been executed.

There are a few complications to consider. The first concerns actions that operate on sets of objects. In this case, example actions that act on single objects must be coalesced with original actions that have that object as a member of its set. Only after all actions are matched can the system determine if all objects in the original action are accounted for and whether the original action is identical to the example.

A second complication concerns the Create Object action. If the developer demonstrates creating graphical objects, those objects will not be the same as objects created by the original Create Object action. Furthermore, objects created by a Create Object action might be deleted by the developer and replaced by identical objects. Instead of matching the parameters of Create Object actions, the system examines the graphical objects created by the developer to see if they "look the same" as the prototype objects from which a Create Object action instantiates new objects. This involves examining the graphical object's properties and type to see if they match. The prototype parameters of Create Object actions match when they produce objects that look alike.

In the example, the new pair of actions must be matched with the actions from the original behavior as shown in Figure 23. Gamut will match each example action with both behavior actions. Gamut finds that the Shuffle actions match identically and that the Move/Grow actions match in their object parameter but do not have matching values. Therefore, Gamut judges the two Move/Grow actions to be *similar*, and that the value parameter of the Move/Grow action has changed.

7.6.2 Propagating Changes To Descriptions. Once a potential change to a parameter is determined, it is propagated to the description stored in that parameter. If the old description is constant, Gamut records that the constant may have changed to the new value. Otherwise, the system calls the description's *difference* method.

The difference method takes the new value and tries to find a way to modify the description to generate that value. Each type of description has its own heuristic difference method, but most simply apply the "one change" rule. Using the one change rule, an algorithm only tries to change

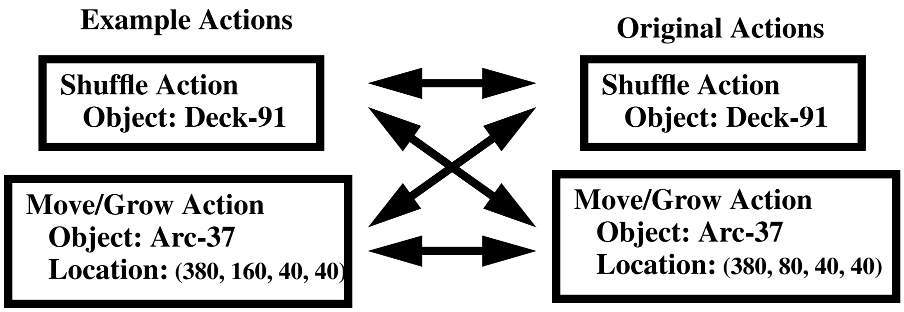


Fig. 23. Gamut matches the two actions from the developer's example with the actions in the original behavior. The Shuffle actions match identically and the Move/Grow actions are similar.

one thing. This is similar to Winston's concept of a "near miss" for training examples [Winston 1984]. The difference method tries changing one of the description's parameters at a time leaving the others the same. For instance, the Get Property description retrieves a property from a graphical object. It has two parameters, the object and the name of the property to get. Using the "one change" rule, the difference method will first search for a different object whose property has the right value. Next, it will look for different properties in the original object to find the value. All matches that it finds are added to the set of changes. The new values are then recursively propagated to the Get Property description's parameters. A few descriptions such as Align (which computes the location of graphical objects) use more complicated heuristics, but they achieve similar results. Propagating new values through the original behavior is common in AI techniques like reinforcement learning and neural networks but has not previously been applied to a PBD tool.

The developer is normally not queried during the search process. Also, most difference methods do not use hinted objects because they are sufficiently constrained by the "one change" rule not to produce too many results. An exception to this is the Count Objects description which returns the number of graphical objects in a set. This description cannot be computed in reverse since knowing that the description is supposed to compute 5 instead of 3 does not tell the description which 5 objects to count. Instead, it asks the developer to highlight the objects and proceeds using that answer.

If the difference methods are poorly designed, there is a potential for a combinatorial explosion. The depth of the search is fixed to be the depth of the description hierarchy, but the number of changes that each description can find could be high. The one change rule and the structure of the application will normally restrict the search space adequately. However, some difference methods must cull the amount of changes they find to only those most likely to be the ones the developer means. Because each description has its own difference method, it can be tailored to meet the specific needs of that expression.

Since the parameters of Gamut's descriptions are not always independent, the result of the difference methods are collected in an And-Or tree. The leaves of the tree are the parameter changes found by the methods. And-nodes indicate when all changes must be made to correct the description and Or-nodes choose among a set of optional changes. Note that this And-Or tree is not related to the decision tree algorithm that is discussed later.

In the board game example, the system has determined that the value of the Move/Grow action has changed from one position to another. It must propagate this new value to the descriptions in that parameter as shown in Figure 24. The Align description (1) takes the new position and tries to find a way to explain it. It finds that if it used a different arrow line (which is marked in Figure 14) instead of the one it originally used, that it can generate the correct position. The Align description then recursively passes the new value for the arrow line to its parameter. The Chain description (2) in that parameter receives the new arrow line object and determines how to use it. It finds that it can change its length parameter from two to three in order to produce the new value.

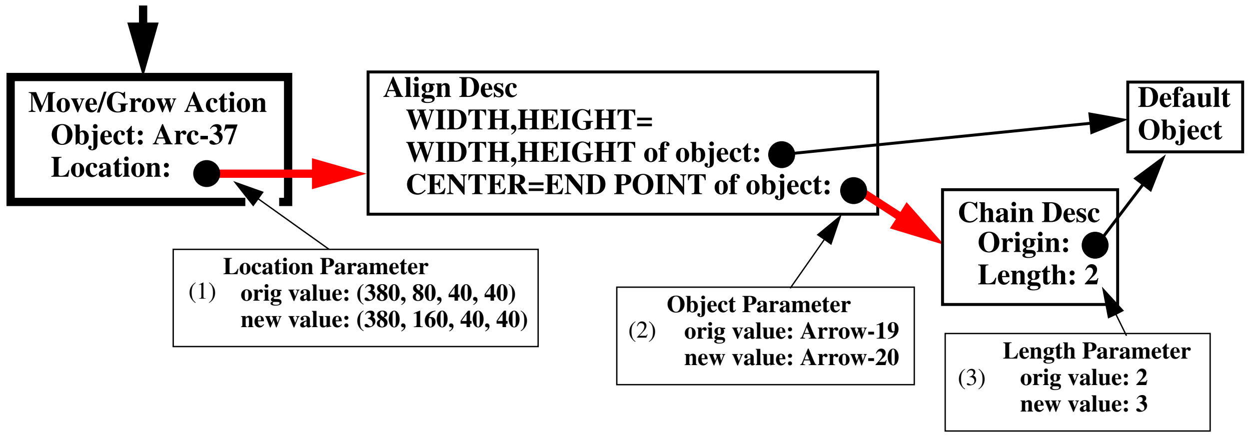


Fig. 24. Gamut propagates the changed values through the description hierarchy. The Align description finds that it can use a different arrow line to generate the new position and the Chain description finds that it can change its length.

Each of the parameter changes discovered by the matching stage and by propagating value through the descriptions is recorded in an And/Or tree. The three changes the system finds in this case are all grouped into a single Or-node as shown in Figure 25. This means that Gamut must resolve any one of these changes in order to satisfy the example.

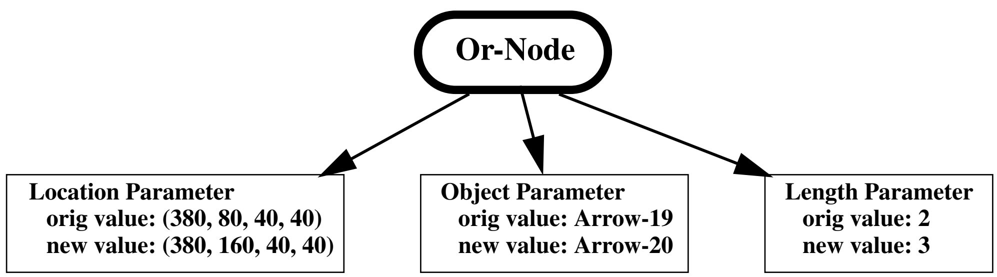


Fig. 25. The various changes are collected into an And/Or tree. In this case, the changes are all independent so they are collected into a single Or-node.

7.7. Stage Three

After completing the analysis stage, the system is ready to assemble the new behavior. Actions that were judged identical or similar to the example are kept. New actions are added in and since order does not matter they can be appended to the end of the list. Unmatched actions are enclosed in a Choice description (see Table II) to indicate that there is a condition. If the behavior already contained Choice descriptions, Gamut uses the matching results to pick the best branch and proceeds to modify that.

In the example, there were no unmatched actions either in the example actions or the original behavior. Thus, Gamut will not have to add new actions to the behavior and it will not have to install a Choice description as an if-then statement.

What remains is to resolve the set of changes found by the difference methods, and to replace the constant parameters of new actions with descriptions if warranted. First, Gamut will search the old behavior for descriptions that already produce the desired result. Descriptions are often shared among actions and searching prevents the system from having to learn descriptions twice. If the search fails, Gamut will try to create a new description. Gamut uses the same algorithm as Marquise [Myers *et al* 1993] to accomplish this task. However, instead of using only heuristics to find matching descriptions, Gamut focuses attention on the highlighted objects.

Creating a new description involves picking one whose type can evaluate to the correct kind of value. Beyond that, the system uses heuristic rules to pick highlighted objects that can be used as parameters. When Gamut replaces the original description, it creates a Choice description and stores both the new and old description together. By not referring to the old value's branch in the Choice's boolean expression, it remains uncomputed, but is available for comparison with new descriptions. Gamut uses these "dead" branches to prevent itself from repeating incorrect inferences.

If Gamut cannot create a new description using the highlighted objects, it asks the developer to highlight different objects. Gamut uses the context of the change including the affected action, the description that is actually modified, and the old and new values, to form an English sentence. After seeing this sentence, the developer can choose to highlight new objects (which are processed as before), or choose to replace the changed value with the new value (without creating a new description), or move on to another question. Any choice the developer makes serves to eliminate items from the set of changes. The system will continue posing questions until the set is empty.

In the example, Gamut has three possible changes it can try to resolve. Since the developer has not highlighted any objects, Gamut does not have any information on which it could construct a new description and searching for descriptions does not find anything useful. So, Gamut will ask the developer a question. Gamut selects a leaf node from the tree that was generated by the deepest description in the behavior's description hierarchy. In this case, Gamut selects the node that changes the length parameter of the Chain description because the Chain description is the deepest in the behavior.

The Chain description contains a method that Gamut will use to generate the question. The method will incorporate the old and new values to create a question that asks about the length of the path (see Figure 15). As mentioned earlier, the developer responds to the question by highlighting the die and pressing learn. This gives the system the information it needs to resolve the change.

When the developer highlights the die, both the deck and the top card are automatically highlighted. Gamut can also detect the offscreen data on the top card and highlight that as well. First, Gamut finds that the desired number, three, is stored in the offscreen number box object in the die's top card. As shown in Figure 26, Gamut creates a Get Property description (1) to retrieve the number from the number box. Gamut then attempts to describe the number box object. It finds that the number box is contained in the highlighted card object and that it is easily distinguished from the other objects in the card (the dots). This causes Gamut to create a Objects From Group description (2) to retrieve the number box from the card and to include with it the criteria to pick the number box from the other objects in the card. Gamut continues by describing the card object parameter in the Objects From Group description. It finds that it is the top item in the highlighted deck object. This finally creates a Top Card description (3) which will retrieve the card from the deck. At this point, Gamut would want to describe the deck object, but it has run out of highlighted objects. So, Gamut leaves the deck object parameter constant.

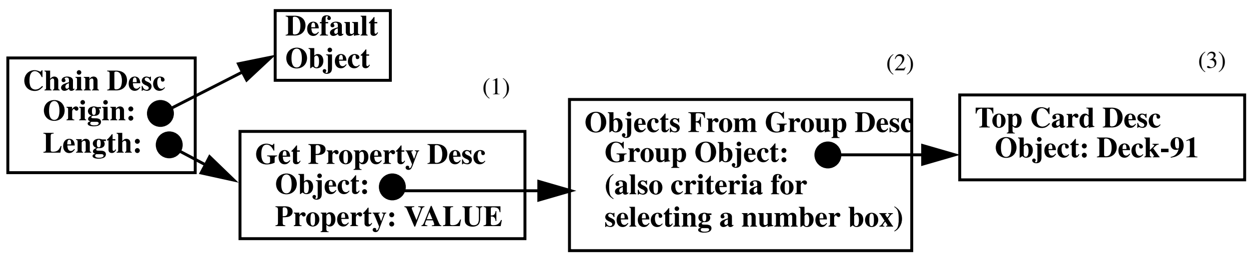


Fig. 26. Gamut creates a series of descriptions in order to describe the number on the die. The descriptions start from the deck object and reach inside to find the value in the number box.

7.8. Decision Tree Learning

In Gamut, decision trees are used to represent boolean expressions in two kinds of descriptions: the Choice description and the Select Object description. Each of these descriptions keeps data to support its own local decision tree.

The decision tree algorithm was invented by Quinlan [Quinlan 1986] as a form of inductive learning (Gamut uses the generic ID3 version of the algorithm). The data from which a decision tree learns has a database-like structure. Each row of the database provides an example of the concept to be learned, and each column represents an attribute of the concept. For instance, the set of concepts might be the different ways a monster can move in Pacman. The monster can either chase the Pacman, run away, or return to the center of the maze (when it gets eaten). Attributes that the monster would use include whether the monster is blue and whether the monster has been eaten. The diagram in Figure 27 shows a possible result of converting a database containing these data into a decision tree.

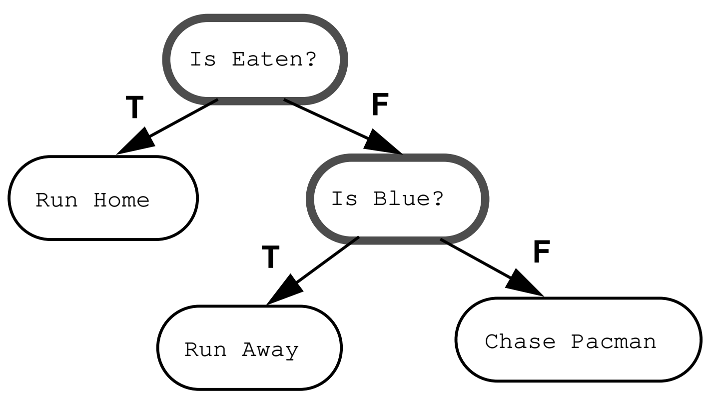


Fig. 27. A decision tree for deciding how to move a Pacman monster.

The decision tree algorithm uses a statistical measure called "entropy" that relates how well an attribute corresponds with a result. The algorithm chooses the attribute with the best entropy value to be the root of the tree, eliminates that attribute from the database, and recurses for each branch of the tree. In Gamut, decision trees learn which objects the Select Object description and which branch the Choice description should pick. Gamut's decision trees never grow very big because most of its boolean expressions only combine a few attributes.

When a Select Object or Choice description is first created, Gamut creates attributes for its database. The attributes are just descriptions and are constructed from highlighted objects as described above. Gamut scans the highlighted objects to create predicates from their properties and positions. Gamut adds new predicates whenever it finds that two examples in the database contradict each other.

For example, when Gamut creates the description that describes how to alternate between the black and gray pieces, it uses a Choice description. The code is shown in Figure 28 and consists of a Choice description that holds two object values. Gamut must then create a decision tree expression to select which piece the description returns. When Gamut asked what should be used to select between the black and gray piece, the developer selected the ghost of the offscreen rectangle. This causes Gamut to generate a set of attributes based on the rectangle. In this case, Gamut will create attributes that test the rectangle's various properties such as its color and line style as shown in Figure 29. If the developer were to highlight more than one object at a time, Gamut could test for relationships between the objects such as whether they had the same color or had a common geometric relationship.

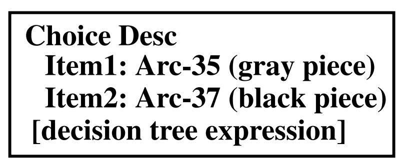


Fig. 28. Choice description that selects between the black and gray piece. The pieces are referenced as constants and the Choice description returns one or the other depending on its decision tree expression.

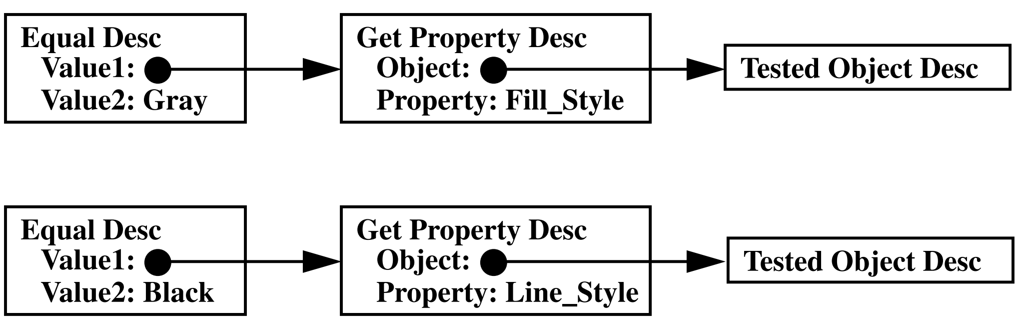


Fig. 29. Gamut creates two attributes for describing the turn rectangle. One examines the rectangle's fill color, the other its linestyle. It creates attributes using the same description objects it uses to describe the parameters of actions.

Each time a Select Object or Choice description is revised, the system will first try to add a new example row to the decision tree database. New examples are formed by evaluating the set of predicates and pairing that with the new result indicated by the revision. If the new example is different from the other examples, a new decision tree can be generated that will distinguish the new case. If the new example contradicts another, though, more attributes will need to be created.

Gamut's use of decision trees is considerably different from the standard practice. Normally, decision trees use hand-crafted attributes and hundreds of examples to learn a large, complicated expression. Gamut uses automatically generated attributes and a few examples to learn smaller expressions. The entropy measure serves as a good way to pick out the one or two attributes that matter out of a set of dozens.

8. USER TESTING

Gamut was tested to see how well its techniques would be understood by nonprogrammers. The goal of the study was to determine how well the participants would learn to use Gamut's various techniques. The sessions lasted three-hours during which 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 qualified participants were able to complete the tasks and the one person who attempted the third task completed it as well. The complete details of the usability study can be found in the thesis [McDaniel 1999].

8.1. Pilot Testing

Prior to the usability study, we performed informal tests to make the system ready for the actual test. These tests involved friends and co-workers who would use the system for short periods and give comments. We used these sessions to fix many of Gamut's bugs and to revise the tutorial and other written material we used for the study. We also addressed some common problems that the pilot testers had with the system which is discussed below.

8.2. Participants

A total of six people participated in Gamut's usability study. 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 also required to be familiar with typical computer interface metaphors as well as drawing editors.

The study's participants were contacted through electronic bulletin boards and email at Carnegie Mellon University. The bboards and email were used in order to help guarantee that the participants would at least have a minimum level of computer experience. As it turned out however, one volunteer had never before used a graphical editor. The person had considerable difficulty performing basic tasks such as creating, selecting, or otherwise manipulating objects. Since the person was completely unable to use graphical editors, it was decided that this person should not be included in the study and was disqualified.

The other person removed from the study was an adolescent who was recruited by his mother. Unfortunately, the child had no real desire to actually participate in the study. The study was not designed for children and was apparently too boring to generate interest. The person spent most of the session looking at his watch and did not try to learn how to use the system.

The remaining four participants were adults with an average age of 28. Some of these people were CMU students and the others were CMU staff members. Two participants were male and two were female.

8.3. Tasks

The experimental sessions lasted three hours during which the participants had to complete an hour-long tutorial and two tasks. Participants used a "think-aloud" protocol [Gomoll 1990] 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. 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?" Figure 30 shows an image of the task's main screen. The goal of the task was to put guide objects into the decks that would tell Gamut the correct answers and to demonstrate the behavior of a pair of buttons labelled "Yes" and "No."

The second task, called Pawn Race and shown in Figure 4, was similar to the board game task that was used as an example in this article. 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 and something to represent the turn indicator. The participants had to demonstrate making the pieces move and alternate turns. They also had to demonstrate that when one piece landed on another, the landed on piece moved back to the start square. They also had to show how the game ended when a piece reached the last square.

The third task which only one participant attempted was based on the video game Q*bert which was call G-bert and is shown in Figure 1. It had a character that jumped on top of a pyramid of cubes collecting objects. The task also had the participant create a monster behavior which was a ball that fell down from the top of the pyramid and moved in random directions. When the ball struck the player's character, the player lost a life, and if the player collected all the markers, the game was won. The third task was fairly long and required its own session. During this session, the participant was given a shortened tutorial that described the new interaction techniques required in the third task.

8.4. Results

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.

Each participant chose to learn a different set of Gamut's techniques in order to complete the tasks as shown in Table III. For instance, one participant preferred to use Do Something exclusively, whereas another preferred to use Stop That. Only one qualified participant was unable to learn enough of Gamut's techniques to build the applications.

8.4.1 Do Something. Though all participants used Do Something to create a new behavior, participant three used it exclusively to demonstrate all behavior changes. Instead of using Stop That, the person would always edit misbehaving objects manually.

8.4.2 Stop That. Though participant three did not care to use Stop That, the other two successful participants did use it regularly. Participant two used Stop That exclusively to revise behavior. In effect, when the person wanted an object to do something, she would select it and press Stop That as if to say "stop doing nothing."

8.4.3 Highlights Ghosts. As discussed below, some participants preferred not to highlight ghost objects. However, this was more apparent during the pilot study.

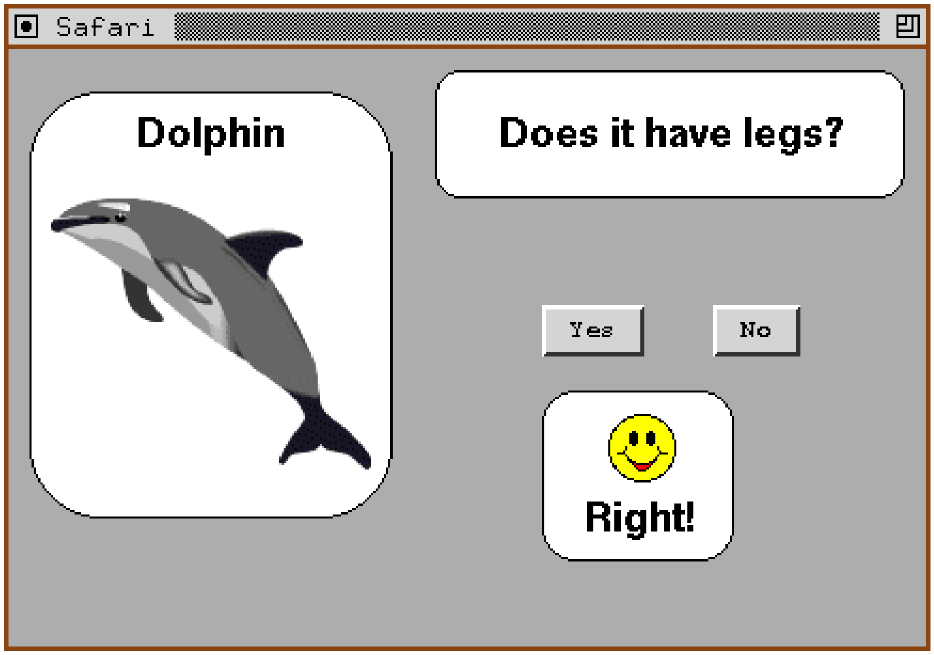


Fig. 30. The first usability task, called Safari, was a quiz game based on a similar principle as the educational game, Reader Rabbit.

8.4.4 Used Replace. When the system asks a question that cannot be answered immediately, the best way to defer the question in Gamut is to use Replace (instead of highlighting an object and pressing Learn). Though this seemed to be too advanced for new users to learn immediately, some participants did use this technique.

8.4.5 Guide Objects. Though all the successful participants needed to create guide objects, participant four had the most difficulty. Though she apparently understood what was needed, she always asked the experimenter for advice with each object she created. Upon reviewing the video tape, it was not clear whether the experimenter had given away too much information or whether the participant designed the objects herself.

8.4.6 Cards and Decks. Both the Safari and Pawn Race tasks contained decks objects which all the participants were able to use. However, two participants created their own decks to demonstrate behavior. Participant two needed to use a deck to select directions for the ball in the G-bert task. Participant three used a deck to represent the player's turn in Pawn Race.

8.4.7 Player Mouse Icons. Participant two successfully used the mouse icons to demonstrate the G-bert task.

8.5. Observations

One of the problems concerned highlighting ghost objects. Several participants particularly during the pilot study were reluctant 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 to system to learn. Therefore, for some cases, Gamut was aug-

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

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

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

mented so that it could recognize a relationship regardless whether the proper ghost was highlighted.

One of the unexpected positive surprises was that some participants were able to understand how to use Replace to defer questions. When they were confronted with a situation where the objects they wanted to highlight were in the wrong configuration, they read the behavior dialog more closely. The dialog lists using Replace in this fashion as one of the options and two of the participants used it correctly.

The most significant problem found during testing concerned guide objects. The participants were always reluctant to create guide objects to use in their demonstrations. Originally, during the pilot study period, the task descriptions only hinted that guide objects would be needed. The participants would ignore those hints and proceed to start demonstrating behaviors. Even when the instructions were made more explicit, the developers would not create guide objects. The tasks eventually had to include specific instructions, separated from instructions on what to demonstrate, to tell participants to draw guide objects. Though participants never seemed to want to create guide objects, once they were pressed, they were usually able to create objects that were suitable for the task they needed to demonstrate.

Another problem occurred when participants would highlight inappropriate objects as hints when Gamut asked a question. There seemed to be situations where certain objects were considered too obvious to highlight. Ghost objects seemed to be one instance. Some participants did not seem to realize that ghost objects could be highlighted. Visual paths and lines connected to objects also seemed too obvious to highlight.

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 that had a less obvious connection to the behavior. Gamut was designed to be resilient to badly highlighted objects so the participant usually had multiple opportunities to answer the same question. Eventually, most would highlight the appropriate object and the system could proceed.

9. 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 [Budge 1984] which makes pinball simulations. A more recent product is Click & Create [Corel 1996] in which the developer first draws the game objects and classifies each as background, characters, or other objects. Then the developer 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 the built-in methods.

Gamut most resembles our earlier system, Marquise [Myers *et al* 1993]. 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 [Wolber 1996] and Frank's Grizzly Bear [Frank 1995] have shown that simple heuristics can be used to infer many forms of graphical constraints and 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. Grizzly Bear can infer some kinds of modal behavior but requires the developer to add new properties to the objects involved. Neither Pavlov or Grizzly Bear can infer how to refer to an object indirectly such as when an object follows a path.

Gamut's inferencing ability is similar to Maulsby's Cima system [Maulsby 1994]. Cima also has the ability to learn from hints and can learn concepts incrementally. However, 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 can only recognize strings of text.

10. STATUS AND FUTURE WORK

Gamut is implemented using Amulet [Myers *et al* 1997] and runs on Unix, Windows, and the Macintosh. Gamut is a prototype system implemented as the 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.

Gamut has been used to demonstrate behaviors that other systems cannot produce. For instance, it has been used to create a Turing machine emulation, complete Tic-Tac-Toe and Hangman games, and various video game behaviors such as a PacMan-like monster, and a Q*bert-like game. Behaviors from educational games such as matching words as in Reader Rabbit have also been created. 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. For instance, the system should be able to note graphical constraints when they are used in a behavior.

One potential method for feedback would be to display Gamut's internal language in a readable form. Currently, everything that Gamut learns is hidden from the developer. Though it is important that the developer be able to create the entire application using demonstration, it might still be useful if one could view the code that is generated. Seeing the code might allow the developer to see errors that would otherwise go unnoticed and might allow the developer to make small changes directly to the code without having to demonstrate a new example.

It would also be useful to improve Gamut's inferencing to infer more concepts. Gamut is currently missing several useful types such as integer arithmetic and strings which would be relatively easy to add. A more ambitious improvement would be the ability to infer higher level concepts. Presently, Gamut learns to relate actions to the events in the developer's application. However, it might be useful to recognize broader concepts such as a player's move, or the type of game that is being created. Inferring higher-level concepts would allow the system to provide more features of a game automatically. For instance, if it could recognize all the moves a player could make, it would be able to list them for the developer and perhaps help the developer design a computer opponent.

It would also be useful to improve Gamut's editor and how it interacts with external environments. Gamut's editor could be expanded to include more kinds of objects such as three-dimensional graphics. It might also allow the developer to create a game with windows on multiple computers for multi-user applications. The developer might also want to use applications and data that are external to the Gamut environment. Currently, Gamut can only manage the data the developer creates within Gamut's windows. It would be useful if Gamut could interface with external applications and data types so that the developer could manipulate them using Gamut's demonstration techniques.

Though Gamut was designed for use in the board game domain, its techniques could be used to build almost any graphical application. Gamut contains heuristics that judge connections and modal states between objects. It does not contain specific heuristics that are only useful for games. However there are many concepts and data types that Gamut cannot infer even for board games. In general, PBD research should not be as concerned with the domain of its tools as it is with the new concepts that the system allows the developer to create. Many past systems have been overly concerned with applying PBD to new domains but only applied known techniques which tended not to advance the field. More work should be given to broader concerns such as reducing the need for guide objects, providing novel input and feedback techniques, or connecting PBD systems to external applications. Such research would be useful in PBD applications in every domain.

11. 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 take advantage of these techniques. The *nudges* interaction simplifies example recording and provides a simple technique 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* and *card* widgets allow complicated behavior to be specified that can involve sets of data and randomness. *Guide objects* permit demonstration of objects and relationships that the system could not guess by itself. Finally, *temporal ghosts* allow the author to directly form relationships with the recent past.

The inferencing algorithm uses a straightforward "one change" heuristic to find the differences between a new example and the behavior it is meant to revise. The difference methods can find specific changes deep inside the behavior which will make it produce the right results. The algorithm also incorporates decision tree learning to handle concepts that its heuristic search portions cannot understand. This algorithm makes it possible to infer conditions with objects that are not directly affected by the actions that depend on them.

These techniques were tested in a usability study where it was 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.

ACKNOWLEDGEMENTS

We would to thank the many people who have helped and inspired this work. In particular, we thank the members of the other members of the thesis committee: David Garlan, Roger Dannenberg, and Dave Wolber for their dedication. Our thanks also goes to the people who helped test and refine Gamut during the pilot and usability studies. Finally, we thank [****] for helping edit this article.

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

APPLE COMPUTER INC. 1993. *HyperCard*. Cupertino, CA.

BUDGE, B. 1984 *Pinball Construction Set*. Exidy Software.

COREL. 1996. *Corel Click & Create*. Corel Corporation and Europress Software Ltd.

FERGUSON, P. AND BRENNAN, D. 1993. *Motif Reference Manual*, Volume 6B. O'Reilly & Associates.

FISHER, G. L., BUSSE, D. E., AND WOLBER, D. 1992. A. Adding Rule-Based Reasoning to a Demonstrational Interface Builder. In *Proceedings of UIST'92*, pp. 89-97.

FRANK, M. 1995. *Model-Based User Interface Design by Demonstration and by Interview*. Ph.D. Thesis. Graphics Visualization and Usability Center, Georgia Institute of Technology, Atlanta, Georgia.

GOMOLL, K. 1990. Some Techniques for Observing Users. In *The Art of Human-Computer Interface Design*, B. Laural, ed. Addison-Wesley, Reading, MA. pp. 85-90.

HALBERT, D. C. 1984. *Programming by Example*. Ph.D. thesis, Computer Science Division, EECS Department, University of California, Berkeley, CA.

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

MAULSBY, D. 1988. *Inducing Procedures Interactively: Adventures with Metamouse*. M.Sc. Thesis. Department of Computer Science, University of Calgary.

MAULSBY, D. 1994. *Instructible Agents*. Ph.D. thesis. Department of Computer Science, University of Calgary, Calgary, Alberta.

MCDANIEL, R. 1999. *Building Whole Applications Using Only Programming-by-Demonstration*. Ph.D. thesis. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, CMU-CS-99-128, also appears as CMU-HCII-99-100.

MCDANIEL, R., AND MYERS, B. 1998. Building Applications Using Only Demonstration. In *Proceedings IUI'98*, San Francisco, CA, pp. 109-116.

McDaniel, R., and Myers, B. 1999. Getting More Out of Programming-by-Demonstration. To appear in *Proceedings CHI'99*, Pittsburgh, PA.

Microsoft Corporation. 1993. *Visual Basic*. Redmond, Washington.

Microsoft Corporation. 1997. *Microsoft Visual C++: Microsoft Foundation Class Library.* Microsoft Press, Redmond, Washington.

Mitchell, T. M. 1997. Decision Tree Learning. *Machine Learning*. McGraw-Hill. pp. 52-80.

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

Myers, B. A., McDaniel, R. G., and Kosbie, D. S. 1993. Marquise: Creating Complete User Interfaces by Demonstration. In *Proceeding of INTERCHI'93, Human Factors in Computing Systems*. pp. 293-300.

Myers, B. A., McDaniel, R. G., Miller, R. C., Ferrency, A. S., Faulring, A., Kyle, B. D., Mickish, A., Klimovitski, A., and Doane, P. 1997. The Amulet Environment: New Models for Effective User Interface Software Development. In *IEEE Transactions on Software Engineering*, Vol. 23, No. 6. pp. 347-365.

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

Winston, P. H. 1984. Learning Class Descriptions from Samples. In *Artificial Intelligence*, Chapter 11, Addison-Wesley Publishing Company, Reading, MA. pp. 385-408.

Wolber, D. 1996. Pavlov: Programming by Stimulus-Response Demonstration. In *Human Factors in Computing Systems, Proceedings SIGCHI'96*, Denver, CO. pp. 252-259.