Improving Programming-by-Demonstration With Better Semantic Expression

Thesis Proposal

Richard McDaniel

November 14, 1995

Abstract

The domain of applications that can be created with programming-by-demonstration (PBD) can be extended by improving the developers' ability to communicate with the system. The techniques provided in this thesis will allow nonprogrammers to create a new variety of complete, interactive applications including many board games and educational software using PBD.

A PBD software tool uses inferencing to induce programs by watching the developer demonstrate examples that show how the application should behave. Current systems reduce their scope or resort to having the developer program because they do not provide sufficient ways to express behaviors and the factors that affect them. Therefore, the goal of this thesis is to develop understandable forms of annotated expression and manipulation that help a system infer a broader range of behavior. To test these ideas, this proposal introduces a new system called Gamut that will present the techniques in a unified software tool.

The first technique replaces the macro recorder method for demonstrating behavior used in other PBD systems with a technique called *nudges*. The developer demonstrates by correcting the system at important points during program execution and also using two nudge commands to communicate important situations. First, the *Do Something!* nudge causes the system to reconsider past learned behavior and try to generalize its knowledge to fit the current situation. Using the *Stop That!* nudge will point out improper behavior and generate negative examples.

Second, Gamut will use a new *deck-of-playing-cards* metaphor to express concepts such as randomness, sequencing, and data storage. By constructing an appropriate deck, shuffling, sorting, and playing cards at key moments, developers can incorporate many effects not available without programming in other systems.

Third, Gamut will improve communication about behaviors by making them more manipulable than in previous systems. Behaviors will be represented as small icons near the objects they affect. Using the familiar cut, copy, and paste commands, the developer can transfer behavior between objects. Determining how to make a behavior operate in the new context will be inferred automatically. An object's state from the recent past will be represented as *temporal ghosts* in which objects become dimmed, translucent images. Many sorts of behavior refer to prior states such as a previous position or an old property value. The ghost objects will allow the developer to make explicit connections.

Finally, to reduce the number of options the system must explore, the developer will be able to give hints by highlighting important objects and properties. A new inferencing algorithm will be created that will take advantage of the hints.

By combining these techniques, Gamut will provide a rich medium for expressing developer intentions, fostering greater communication between the PBD system and the developer and enabling the developer to create highly interactive software with minimal programming expertise.

1.0 Introduction

This thesis will increase the expressive power of programming-by-demonstration (PBD) to enable nonprogrammers to build a new class of applications. Specifically, new techniques will be researched to make it possible for nonprogrammers to build more kinds of interactive programs such as educational software, board games, and video games. Among the techniques will be ways that will allow the developer to point out important aspects of behavior to the system that will improve the system's power to infer the developer's meaning. Other innovations will provide better ways to manipulate important concepts and aid the developer in building useful applications quickly and easily.

Making it easier to build applications enables more people to be creative using the computer as their medium. Computers are now a common household item. About 37% of American households own a personal computer today [31] and the percentage is likely to rise. Certainly most computer owners will simply buy much of their needed software, but companies cannot possibly know the needs of every person. Plus, many people get creative impulses and want to build things and not be forced to spend money on a commercial product that does not meet their needs and desires.

It is presumptive to assume that a person wanting to build some piece of software will be motivated enough to learn a programming language adequate for the task. It is well known that programming is a difficult task for people to learn, and most people are not able programmers. In fact, there are studies that show that programming using a symbolic language is a significantly difficult task [39]. Rather than make programming a matter of literacy, the tools for building software should require no programming knowledge at all. There are people with the ability to draw or write music and some of them would like to be able to incorporate their skills into software. The goal is to allow creative people who have an idea for an interactive program to be able to implement their aspirations without ever having to learn a textual programming language.

Software tools are one means for making software development possible for nonprogrammers. A software tool is some program or method that is used to help build other kinds of software. The person who builds an application using a tool is usually called the *programmer*; but since programmers using this thesis' tool are actually nonprogrammers, the term, *author*, is preferred. The audience that uses the author's application are called *users*, or more specifically, *players*, because Gamut's domain is interactive software like games.

The best tools make software development so simple that the author forgets about the tool and just builds things. Excellent examples of these kinds of tools are game builders such as Pinball Construction Set [1]. Game builders present games as an object-oriented set of components that can be assembled directly onto the playing area. The rules for a given class of game are encoded into a set of choices and selected with dialog boxes and menus. Similar kinds of tools have also been made for more sober pursuits like form generators. These kinds of tools can be characterized as very easy to use but also have restricted domains. It would be unlikely that one could make an math tutor using Pinball Construction Set, or for that matter, write an pinball game using a form builder.

Another branch of software tools makes the task of programming easier but does not eliminate it. Tools like Visual Basic [44] and HyperCard [13] provide palettes that allow the author to place interface components directly into windows. Properties like the name of a button or the color of the background can be set directly through menus. Some tools even allow the author to directly specify a few simple behaviors. However, the bulk of any nontrivial application's behavior must still be programmed.

Between these extremes come tools that build fairly general software but incorporate more machine intelligence into the process. Programming-by-demonstration is one field where these kinds of tools are emerging. The tool described in this proposal falls into this category. PBD is a technique where a mechanism is directed to analyze the actions of the author and infer the intended meaning. The system essentially "watches" as the designer performs the application's behavior directly on the objects in the system. The mechanism tries to learn the author's intent from the demonstrations, and builds a program that best correlates to that intent. Since demonstration uses the familiar, direct manipulation interaction that most computer users have already learned, an author should have little trouble learning how to demonstrate a behavior and does not need to write code.

Though the potential benefit that PBD provides for nonprogrammers is great, the work in intelligent tools is recent, and still mostly incomplete. For instance, there are still no widely accepted PBD tools sold commercially. One aspect of a commercial product that comes close to PBD are macro builders such as found in Microsoft's word processor, Word [24]. Currently, the most successful PBD tools in research have been applied to rather limited domains.

Tools to make interactive software like Marquise [30], Demo II [5], and KidSim [2] are just emerging and still have deficiencies that prevent an author from creating complete and useful applications.

The focus of this research will be to invent ways for the author to express behavior for a tool to understand the meaning and context of that behavior better than current tools can. We will build a new tool called Gamut as the means for testing these techniques. Gamut should significantly extend the domain of applications that can be built by nonprogrammers. As discussed more fully in later sections, the kinds of meaning the system will be able to infer include more kinds of conditionality and better descriptions of objects and their relationships to other objects. The means for expressing behavior will be accomplished with several new interaction techniques which emphasize direct manipulation and programming-by-demonstration. These techniques include better ways to present examples to the system, a new metaphor for building common abstractions, and betters ways to manipulate behavior and annotations.

The organization of the following sections details the proposed work of the thesis. First, Gamut's domain is defined and presented. Then the challenges and interesting features of the domain relevant to our research are discussed. This is followed by related work to address how others have tackled similar situations. This is followed by the means for addressing the challenges including our new techniques and ideas. The last few sections include a hypothetical example, the expected contributions of our work, and the schedule for completing the work.

2.0 Domain

An author will be able to produce in Gamut board game-like applications. The computer will be able to act as a medium for human players and serve as referee. Games have been used to simulate real world events like managing a business or replaying an event in history by translating features of these processes into game elements. Other games are more abstract like chess, maze games, and card games like Klondike and Spades. Since the applications are implemented on a computer which can provide automation and animation, the domain also includes video games like Reader Rabbit [37] or Number Munchers [34].

2.1 Why Board Games?

Building whole board games requires a level of sophistication that has not previously been provided in a tool that does not require using a programming language. A demonstrational tool that can decode the meaning of a board game's rules and behavior requires new techniques to be developed. Current PBD systems do not have the expressive power to build even relatively simple games. The required abstractions and engineering can be covered within the scope of a single tool; yet, the principles will be general enough to apply to other domains. Domains that will especially benefit are those requiring interactivity between the user and the application, context sensitivity, and conditionality between the objects in the application.

Board games are also a good domain since there are many useful things that can be accomplished with board games. Many of the applications described in the Computer-Assisted Instruction literature [17] can be easily likened to a board game. There are countless possibilities for building games with educational merit. One problem preventing these ideas from being implemented is the difficulty of programming. With a tool like Gamut these kinds of educational games can be made more easily by people who are not normally considered to be computer users. The game 1830 [42] is a "railroad game" which consists of players buying and selling stocks for several railroad companies; players also build railroad routes to establish the value of their companies. This kind of player interaction can be very educational, especially if it is played in the context of learning how real companies actually trade stock. Other kinds of games can teach lessons in history or mathematics.

2.2 What is a Board Game?

This section presents a definition for board game to help clarify what kind of applications will be possible in Gamut. First, the objects and other artifacts of board games are described. This is followed by a discussion of how these objects interrelate. Then, how a player interacts with a board game is detailed. The last section defines rules and strategy and declares why Gamut will be able to referee a game but not necessarily play as an opponent.

2.2.1 Pieces, Tiles, Cards, and the Board

The constituent parts for board games are fairly standardized. First, there are game pieces which are smaller sized objects that can be moved. Usually there is a board which represents the playing surface, and pieces are placed on the board. The board may be fixed or it may consist of tiles that can be moved, rotated, or otherwise modified. In

Gamut, the board is restricted to two-dimensions. One can simulate more dimensions by stacking board tiles, or by using multiple windows, but Gamut will not support explicit three-dimensional graphics.

One of the more important components to most board games are cards. These are actual playing cards like the Chance deck in Monopoly or an ordinary Poker deck. Cards are data. They can be filled with numbers, pictures, text, instructions, and all manner of other items. Decks of cards provide more operations. Cards can be moved from one deck to another, played out of one deck onto the board, or moved from the board back into a deck. Decks can be shuffled to provide randomness, and decks can be combined or split to provide different kinds of sorting.

What an object is called is likely to have little to do with how it acts. Games routinely use objects in roles that are held by other objects in other games. In Magic Realm [11], the board is composed of tiles that are first shuffled and dealt out to the players just like cards; hence, cards can be thought of as tiles or vice versa. The board may be composed of pieces as well as tiles. In a video game, some of the background objects move or can be manipulated. The hammer object in Donkey Kong [4], may be a piece or it could be considered part of the board. The object's behavior depends on how the object is used in context with the other objects in its surroundings.

2.2.2 Relationships Between Objects

The proximity and position of objects are an important consideration for many game rules. Objects that are co-located can have different meaning than an object placed by itself. For instance, in Parcheesi and Backgammon, two pieces in the same location mean the path is blocked. A single piece in the same location can be captured by the opponent. Another example where proximity matters is when the PacMan [35] runs into a monster, then the PacMan dies. Also, the paths between objects can have as much meaning as the objects themselves. In chess, each piece is constrained to follow a limited number of paths. To describe a chess move requires the author to describe as clearly as possible all legal paths.

2.2.3 Interactivity

Interaction between a player and a board game is normally through direct manipulation. An often-used interaction is moving a piece by dragging it with the mouse. Though the act of dragging may seem simple, the kind of dragging found in board games is filled with content. Moving one kind of piece to a position could potentially have different effects than moving another kind. Pieces are often owned by a player and cannot be moved by other players. The final position where a piece is placed can also have different meanings. In Monopoly, moving a piece to the "go to jail" space has significantly different meaning than moving to the "chance" space.

Other kinds of interaction include manipulating widgets and entering text. Board games differ from many standard applications in that very few standard toolkit style widgets, like buttons or scrollbars, are found in the main area of the interface. In many ways, the game is itself a large, custom-built widget. Text entry is not emphasized in most games. When a text field is needed, it will normally require entering only a number or a string name. Most games do not require advanced command language parsing.

2.2.4 Rules Versus Strategy

One goal of Gamut is to make it possible for nonprogrammers to develop a board game including its rules so that others can play the game on a computer. It is not within Gamut's purpose to provide intelligent agents for computer opponents. This difference here is the difference between a game's rules and its strategy. The rules of a game are its components and the conditions that determine whether a player's actions are permissible. The application becomes the referee for the game. Illegal moves can be prevented, turn orders are enforced, and the computer takes over most of the tedious bookkeeping chores, like keeping score. Likewise, the application can have some autonomous actions like timers and animations controlled by the computer.

On the other hand, a strategy is what a player uses to play and try to win a game. There are AI researchers who work on making computers play games more intelligently, but that research is not part of Gamut. It is the strategy component that makes playing a game interesting; so strategy will be reserved for the players. For this reason, it will be relatively easy in Gamut to make a PacMan-like maze game, except that it will be difficult to make the monsters act intelligently. Only those strategies that are easy to encode as rules will be easily expressed in Gamut.

2.3 Example Board Games

Some games have already been mentioned in passing. Board games like Chess and Backgammon have been around through the ages. Their paucity of board elements are more than made up for in complexity of their rules. Card games like Klondike, Spades, and Poker have rules as complex as Chess and Backgammon. Like the board games, they lack graphical embellishment and rely on rules to describe the game. Modern board games like Monopoly rely less on memorized rules and install more of the behavior into the board or the card elements. Both of these styles will be realizable with Gamut. In general, the difficulty of building a game is expected to increase as the complexity of the rules increase.

Gamut will be able to produce many kinds of video games. Games designed for younger children, such as Playroom [8] and Treehouse [9], where the player is presented with a scene of clickable objects will be easy to assemble since the behavior in these games is notably simple. Other video games include puzzle games and two dimensional arcade-like games. A puzzle game like Lemmings [18] is very much like a board game in which the pieces (the Lemmings) are assigned behaviors that change the state of the board. It will also be possible to build maze games such as PacMan and Number Munchers with Gamut. Even monster behaviors like the kind found in PacMan will be expressible.

Building some applications will require true experts. For instance, it should be possible to build an application as complicated as KidSim [2] in Gamut. To do so, one would need to demonstrate how to demonstrate things in the KidSim way. Likewise, representing KidSim-like data in Gamut's framework requires much forethought. No one would expect uninitiated people to be able to build KidSim in Gamut without considerable time and motivation.

Of course, there are countless games that are simply outside of Gamut's domain. Some examples are three dimensional games like Starblade [40], or applications that require an external database such as a large dictionary. Gamut will have a limited ability to infer arithmetic expressions implicitly. Expressions requiring anything beyond linear arithmetic will require the author to type in the expression by hand. Complicated expressions that require special solving techniques like differential equations will not be expressible at all. This makes games that simulate true physics like Smith's ARKkit [38] inexpressible in Gamut. Some of these deficiencies can be circumvented by adding special interfaces for expressing these complicated sorts of interactions.

3.0 Requirements and Challenges

A game author desires to express the design completely and efficiently. The question is, what is the best way to express the rules of a board game? Actually, one might expect authors to prefer to write rules as English sentences as would be found in a rulebook. Unfortunately, that kind of natural language understanding is beyond the scope of current technology. Also, expressing rules in a formal language -- though useful as the system's internal representation -- would violate the main point of making Gamut usable by nonprogrammers. A terse, nonverbal, means of communicating rules to the system is required that can be easily learned and understood by nonprogrammers. The means of expression should provide all the concepts the author needs to use. Since the use of a formal language is not an option, ways to express behavior through direct manipulation and demonstration will be used.

The requirements come from two main sources. The first source is the domain. Building board games has certain requirements regardless whether it is built in a PBD system or something else; although, the focus will be on those requirements that have traditionally been challenging for PBD systems. The second source of requirements comes from programming-by-demonstration.

3.1 Expressing Board Game Elements

The rules of a board game consist of more than graphical objects. Relationships between objects can be based on whose turn it is to play, the moves the player made in the past, as well as the relative position of the pieces. A board game's elements have been categorized into five parts. First, there is player input: the way players can interact with the computer. Second, there is modal behavior which represents internal conditions which make behaviors change. Third, there are graphical pathways and connections which control how objects can move or change. Fourth, there are temporal relationships which connect the current state of the game to the recent past. Finally, there is the manner that games use numbers such as for counting and arithmetic.

3.1.1 Kinds of Player Input

Board games are very interactive; hence, an emphasis must be placed on expressing player interaction. A player has two input devices, the mouse and the keyboard. The mouse can be used in two ways: clicking and dragging. Keyboard behavior can be treated mostly like clicking behavior. Each key on the keyboard acts like a mouse button. The keyboard can also be used to enter whole strings as a unit. Most current demonstrational system are built to handle only single-event kinds of behavior like clicking the mouse; dragging has not yet been handled to the degree required for board games. Dragging consists of a continuous series of interrelated events with behavior that depends as much on the order of prior events as to the event that just transpired. Since dragging involves processing more than just a single event, most of today's PBD systems either require significant preparation to be performed on objects that can be dragged before the drag is demonstrated [6], or severely limit the kinds of dragging permitted.

3.1.2 Conditional and Modal Behavior

In a board game, a single action can have different effects under different circumstances. In Monopoly, moving a piece to different kinds of squares makes the game do different things. This means that the inference algorithm needs to be sensitive to the possible cues the context can provide. Since each path of a conditional behavior must be demonstrated independently, the system must determine how the paths join together and where the branch completes. Preferably, demonstrating conditions should be kept implicit. At no time, should the author be confronted with having to manually "add a condition" to an already created behavior as is required by some present-day systems [2][5].

Many games have distinguished phases or "modes" where different rules apply and the players have different goals. For instance, modes are used to control multi-player interaction through turn taking. A railroad game like 1830 divides a single turn into several phases such as a stock buying phase, a dividend collection phase, and a stock selling phase. When switching between different modes, whole sets of objects are changed at once. The whole screen may change to a different pattern, or the behaviors of objects can be switched to make objects interact differently. The author needs a way to group objects and behavior into modes as well as ways to tell the system how to change between modes.

The benefit of providing a direct way to express modal behavior is mostly a matter of convenience. The alternative to manipulating modes is to distribute a condition to each behavior and object as required in Marquise [30] and KidSim [2]. Distributing modes when there are a large number of affected behaviors is tedious for the author, both when the modes are created and especially later when one wants to make modifications.

3.1.3 Graphical Paths and Connections

Paths are used in board games to determine where objects move and the order of some events. Objects on the board are often connected implicitly by being nearby other objects or can be connected explicitly with lines or arrows. Pieces and tiles connect to the board by being placed on top of it. Sometimes it is not apparent to the system when objects are connected in a path. The author needs to be able to show connections that are hard for the system to infer. For instance, a system would likely have trouble detecting that the tunnel in PacMan connects opposite sides of the board.

There are also more abstract kinds of connections. The legal moves of a chess piece surround the piece in a certain configuration. The author may want to mark destination squares to distinguish them from other squares. These kinds of connections will move as though they were part of the piece. Other kinds of connections also come and go depending on where a piece is moved. A piece will be connected to one set of board objects at one position but it will need to change connections when it moves to another position.

3.1.4 Temporal Relationships

Time is used in games to establish an ordering of events and turns. Time is also used to control animation and other kinds of automatic behaviors. The author needs to be able to look at time in a way that is manipulable and understandable. Connections have to be possible between the state of the objects in the present and their state in the recent past. Some kinds of temporal concepts used by games include: a piece's last position, whether a piece has been moved before, the direction a piece is moving, etc.

Gamut should not be as concerned with the distant past because those effects are few and generally have less direct impact on a game. Effects of the distant past can be recorded using markers and off-screen objects whose state changes as the key events occur. Although the author could use the same techniques to record the recent past, the effects are so frequent that assembling objects to record the actions puts a burden on the author. The author can make connections more easily if the recent past is made accessible.

Prior systems like Marquise [30] have just touched on the benefit of making the recent past accessible. Marquise uses special cursors to indicate the positions where the mouse was clicked down, moved, and lifted. These cursor positions reflect the recent past of the mouse events and the author can use them to show explicitly how the graphics should respond. The success of these special cursor markers suggests that even more benefit can be derived by making more effects of the recent past accessible automatically.

3.1.5 Being Able to Count Things

Finally, the system must be able to count objects. The number of certain kinds of pieces or tiles can determine important factors in a game. In Civilization [41], the number of city pieces owned by a player determines whether the player's game can proceed to the next phase. Often, the player's final score is determined by counting the pieces the player owns and applying the number to a simple formula. There are two kinds of counting: absolute and relative. Example absolute counts include how many pieces are on the board or how many rounds have the players played. Relative counting includes questions like: has this one piece moved after another piece has moved, or how many squares separate two pieces.

3.2 Programming-by-Demonstration

Gamut's other requirements concern the demonstrational paradigm. Demonstrational programming consists of the author showing actions which the system converts into working code. The conversion process has two areas of concern. First, there is the editor in which objects are drawn and new behavior is demonstrated. The editor is the main interface to the system; it is used to record and annotate demonstrated actions. Second, the demonstrations must be encoded into the rules of the game. This encoding process has three parts: a language component which stores a game's rules, a set of algorithms used to learn the rules, and a means to observe and correct the inferred rules.

3.2.1 The Editor

Every demonstrational tool has an editor at its base. The editor is what the author uses to assemble the components and demonstrate behavior. For Gamut, the core of the editor is a graphical interface builder, essentially a drawing tool for applications. The interface builder is augmented with the ability to watch the author manipulate the interface in order to demonstrate behavior. How the author tells the system what to watch is an issue in PBD research because the author will not find the system usable if it requires too much work to make a demonstration. There are two independent realms of input used in this kind of demonstration. The first distinguishes author input from player input, and the second distinguishes the different phases of behavior.

An author must be able to distinguish the interaction that the player would generate from the interaction the author has with the system. When the author draws a chess board, the author is not restricted on where or how squares are drawn; however, in the game, the player is not allowed to move the squares. The traditional solution is to separate these two kinds of interaction into separate modes. This is the Run mode versus Build mode switch found in many application builders. Another solution restricts the kind of input the player can make so that it can be kept separate from author input. For instance, the player might use keyboard input to play the game, but the mouse is used by the author to build the game.

The other set of distinctions the author must switch between are demonstrating the cause for a behavior, demonstrating the effects of a behavior, and just manipulating the interface to set up the next demonstration. Only two of these modes have been traditionally given names in the literature [47], specifically Stimulus mode for showing causes and Response mode for showing effects. These modes arise from the nature of behavior. A behavior not only has actions, but in order to be interactive, must have a set of events that cause those actions to occur.

Many early PBD systems did not have a Run/Build mode dichotomy. Instead, the inferencing was used to aid the author in performing some task such as editing a paper or manipulating files [33][23][10][26]. For these domains, there is no secondary user who sees the results of the system. Other systems did not distinguish the different parts of behavior, and could only produce "procedures" or "macros" that were not interactive [3][10]. Systems that do infer behavior normally use an extended macro recording metaphor for demonstration [47][6]. A macro recorder is designed to be like a tape recorder: pressing the record switch tells the system to start watching the author. The extended macro recorder has an added "record event" phase which tells the system to watch for an event instead of actions. But because behaviors normally rely on more conditions than just the event, extra buttons need to be added. Added switches are used to distinguish positive and negative examples, or whether conditions should be checked before or after an action is invoked. In short, the macro recording metaphor is not well suited to distinguishing the parts of behavior.

3.2.2 Inferencing

A PBD system's inferencing power determines how much it can understand, and what sorts of programs can be generated. A system with greater power can generate more kinds of programs with fewer explicit actions required from the author. In general, the author should be aware of what the system is guessing, and the system should be able to make good guesses with as few examples as possible.

Inferencing has three components. The first two of these parts come from the artificial intelligence community; these are knowledge representation and the inferencing algorithm. The third component relates to human-computer interaction in that the system's knowledge must be revealed to the author in an understandable manner so that it can be reviewed and modified. The inferencing algorithm is what converts the demonstrations performed in the editor into the knowledge representation. There are two kinds of knowledge stored in Gamut, the first are concepts used to describe games in general. These are a set of heuristics used by the inferencing algorithm to recognize the salient parts of a game. The other knowledge is the encoding of the game rules currently being learned.

3.2.2.1 Knowledge Representation

All systems need an internal representation. The representation is a language for storing the inferences and data that are generated by the system. In Gamut's case, the knowledge can also be translated into a working application so the language must be interpretable.

The representation is divided into two parts. The first part represents the objects that are drawn using the editor. This is where state and property information is stored. The main task of designing the object system is to divide up the objects and properties in ways that the author will understand. The board game domain does not seem to need special consideration for the object system. Pieces, cards, and tiles divide nicely into objects and their properties should be straightforward to formulate.

The second part of the representation is the language that tells how the objects behave. This is one of the major challenges for Gamut as the board game domain heightens the needs for more sophisticated kinds of language constructs than needed in any previous PBD system. The language can be divided into two layers, as on Halbert's SmallStar system [10]. The top level, called the structural layer, contains constructs that relate the conditions, loops, and actions of the behavior. If the author moves an object, then a move action will become part of the structure. The challenges involved with designing the structural portion of the language deal with how it can be manipulated. The language should be searchable so that new demonstrations can be compared with already inferred rules. Also, the language needs to be easily transformed. New instructions can be added to behaviors and conditions and loops can be added around existing code.

The second layer is the structure's parameterization. The purpose of parameterization is to provide a "data description" for every object used in a statement. The description must be as close as possible to the author's actual reason for manipulating that object so that when the behavior is performed again, the objects chosen will match the author's concept of the rule. What makes data description difficult is that when the author picks an object through direct manipulation, the reason that object is picked is not apparent. The system must find a way to describe the object by examining how the object is used and by noting its surrounding context.

The descriptive power needed for board game objects is especially difficult. For instance, the final destination for a Monopoly piece might be described as "the square that is the dice roll's number of spaces away from the square where the current player's piece started." This description is complicated because there is more than one factor involved. There is the layout of the board, the starting point of the piece, the current player's turn, as well as the number rolled on the dice. Moreover, these factors combine so that one object's description depends upon the description of another object. In this example, the final destination depends upon a description of the piece's starting point which depends on the description of the piece itself. Multi-step data descriptions have normally been considered too difficult for a PBD system to infer, so previous systems have forced the author to build data descriptions directly through dialog boxes [10], or have simplified the data descriptions so as only to include a single step.

3.2.2.2 Inferencing Algorithm

The inferencing algorithm is the intelligent part of a PBD system. The goal of the algorithm is to make guesses about the author's demonstrations and turn them into better guesses as the author continues. This process is, in essence, machine learning. The difference between this domain and other machine learning domains is that the subject being taught must also be applied by the system in the form of working code.

Normally, the internal language provides a great deal more flexibility than the inferencing algorithm has the ability to generate. In order to generate a greater degree of flexibility, the inferencing algorithm will need two things. First, there needs to be better ways to augment data descriptions. As more examples are received, the data descriptions of existing structures need to be transformed to better describe the author's concept. Second, it must be possible for the algorithm to infer new structures. This involves adding new conditions and actions in response to new demonstrations, as well as generating new behaviors.

When there are a large number of rules and behaviors defined, it becomes more important that the system try to make best use of what it has already inferred instead of guessing each new demonstration from scratch. To look up and generalize previous inferences requires that the inferences be easy to catalogue and search. Different information that may guide a search are player input, conditions or structures, connections with other rules, and object references.

3.2.3 Feedback and Editing

The set of rules and behaviors stored internally must be presented and made modifiable for the author. Most modification will be performed by demonstration, but sometimes the author will need to view the rules in an explicit format to make changes directly. Gamut should not present the language as code to be manipulated with a text editor. That would not only would violate the nonprogramming requirement, but would also make possible syntax errors and other kinds of problems.

Most previous systems presented the internal representation in code form [7][3][19][6][47]. Mudugno has shown in her work that users understand picture representations best [26]. Her system as well as Kurlander's Chimera [16] use a "comic book" metaphor for displaying the system's actions. Other systems like Marquise [30] compromise between picture and text and present manipulable English-like sentences.

4.0 Related Work

This research draws inspiration from at least three fields: graphical software tools, programming-by-demonstration, and machine learning. PBD is, of course, the strongest influence. Graphical software tools are normally direct manipulation editors used for assembling programs or, in many cases, a graphical interface. Gamut uses basic application builder principles in its editor, menus, and object system. Machine learning and especially the sub-field of inductive learning is the field upon which PBD inferencing is based.

4.1 Graphical Software Tools

A software tool is used to reduce the amount of programming required to make a specific kind or part of an application. Tools are essentially special purpose languages and editors that describe programs in their domain more concisely. In the graphical interface domain, these tools naturally gravitate toward graphical representations that are edited with direct manipulation. Three kinds of tools are discussed in this section. First, user interface builders which are tailored to build just the human-computer interface to a program. Next, programming environments for higher-level languages commonly have user interface building properties. These sometimes take the form of prototyping tools which are used to quickly program an application for interface testing purposes or "rapid prototyping." Last are application builders which are tools tailored to a specific domain.

4.1.1 User Interface Builders

A user interface builder is essentially a direct manipulation drawing tool that uses interface widgets like menus and scrollbars instead of graphics like rectangles and circles. UI builders like NeXTStep [32] and the one included in Visual C++ [45] allow the interface designer to arrange widgets on a prototype window, choose colors and styles, and experiment with different configurations. When the designer finds the interface acceptable, the layout is converted into code. The code will contain all the information necessary to draw the interface, but parts that perform the application's behavior are left blank. A programmer then fills out stub procedures or writes callbacks to connect the interface to the actual application code. UI builders give virtually no need to the designer for specifying behavior. Part of the purpose of tools like Gamut is to use similar direct manipulation techniques but also permit the designer to show what the interface does as well as show what the interface looks like.

4.1.2 Higher-Level Programming Environments and Prototyping Tools

When a user interface builder is tightly coupled with a higher-level-language, the result is often an environment that is easier to program that a UI builder alone. Languages like SmallTalk [14] and Visual Basic [44] have graphical primitives built-in to make interface specification easier. Though easier to program, these tools still lack the ability to specify behavior without programming. By making widgets more powerful, one can use them to specify some behavior as was found in Gould and Finzer's Rehearsal World [7]. Rehearsal World based its widgets on an acting metaphor: widgets were called "actors" and events and behavior were called "cues." By connecting cues to actors' attributes, one can generate programs. Though connecting widgets is not necessarily programming, the fact that among Rehearsal World's widgets were a "pointer actor," a "true-false actor," and a "debugging actor" is telling.

Prototyping tools are a form of programming environment where a few behaviors can be specified through direct manipulation. Prototyping tools are purposely kept simple to enable a designer to implement many iterations quickly to test different ideas. When the intended behavior becomes too complex to be specified directly in the prototyping system, the developer is forced to program in a scripting language. The use of scripting languages makes prototyping tools much like a higher-level programming environments. HyperCard [13] uses an "index card" metaphor where cards represent windows and transitions between system states is handled by having HyperCard "go to" the right card. HyperCard's index card metaphor is not the same as Gamut's playing-card metaphor. A deck of playing cards is an entity separate from any given window. The graphics on a given card need not even be shown to the player. Furthermore, one does not "go to" a playing card; playing cards are shuffled, dealt, and played. Macromedia Director [21] uses a score sheet abstraction to lay out events. This allows the designer to build a movie-like simulation of an interface. Though the behaviors provided in prototyping tools can be useful, they are too limited to specify behaviors used in most interactive software.

4.1.3 Application Builders

An application builder is tailored toward building a specific kind of application usually without programming. For example, Pinball Construction Set [1] lets the author draw the parts of a pinball game in a window. Components such as ball and levers have behavior built-in and other connections are made using menus. A game built with this tool can be executed immediately and with little effort. Financial programs can be built with similar ease using spreadsheets. A spreadsheet's cells hold the requisite numbers and equations in a mostly free-form style which is easy to create and use. Form generators can be used to build data entry systems. Like a UI builder, the designer can lay out data widgets onto forms. Since most data base entries have no behavior besides being catalogued, the widget behavior is sufficient to build the whole interface. Though application builders can be used to build complete applications without programming, they cannot be used to create new forms of behavior. The domains of these tools are have been selected because they require only a small set of specific behaviors. The techniques presented in this proposal are developed to specify new behavior using demonstration. By allowing the developer to design new behavior, the developer is given a whole new dimension of control over the system.

4.2 Programming-by-Demonstration

Programming-by-demonstration has been used in a number of research applications. The purpose for using PBD has always been to aid end-user programming or customization. Many PBD systems serve to augment an existing interface to help a user perform a task, but this research is aimed at using PBD to build applications. Therefore, the PBD systems Gamut most resembles are those that are used to build interfaces or small programs. Also, since the research is concerned with communication between author and system, we examine other systems that use hints and annotations.

4.2.1 Building Things with PBD

Peridot [27] was the first attempt to apply PBD to the task of building user interfaces. Peridot could be used to program widgets like buttons, scrollbars, and menus. Peridot's inference engine was rule-based. When the author provided an example, the system would run through its list of rules asking the author which ones applied to the situation. The system had built-in knowledge about specific kinds of widget behavior like scrolling and selecting; plus, it had the ability to detect some kinds of iteration. Naturally, since Peridot had no real learning capability, the rules could easily be exhausted when the author attempted to demonstrate something novel. Likewise, users were annoyed by the constant barrage of rule confirmations. Gamut rules will be able to be composed so that novel behaviors will be more likely to be recognized; plus, the means for correcting behavior will not require constant, yes/no confirmations.

Gamut most resembles Marquise [30], a previous system the author built. Just as Gamut, Marquise's task was to develop whole applications, but Marquise focused on the graphical editor domain. Marquise provided a set of built-in tools for distinguishing common graphical editor characteristics. For instance, Marquise made it easy to build a palette or cause a node to be selected. Marquise's heuristics also made it possible for most demonstrations to be understood with only a single example. This rigidity, though, makes Marquise's inferencing ability too low to be applied to the board game domain. Marquise did provide a set of ghost cursors to record where the mouse was used in the recent past; Gamut will extend this idea further.

The Stimulus/Response demonstration style was first used by Wolber in the Demo [47] system and its successor Demo II [5]. These systems are able to infer linear constraints between graphical properties. Interaction in the Demo series is limited to the transformations provided in the graphics editor such as clicking on or resizing an object. Demo does not permit the author to demonstrate whole new interaction styles. Conditions in Demo are not inferred by the system as they will be in Gamut; instead, the author is required to choose conditions from a menu and apply it to a behavior. If the author does not know that a condition is needed, chooses the wrong one, or puts the condition in the wrong place, then the system's behavior can be difficult to understand and fix. Also, Demo's data descriptions were confined to a single, constant objects unlike Gamut's indirect data descriptions. This made it impossible to have one behavior that has effects on different kinds of objects.

4.2.2 Using Hints and Annotations

Part of Gamut's expressiveness relies on the author providing hints and annotations in parts where the system would have trouble making inferences. Most prior PBD systems provided absolutely no forms of annotation ability instead relying on plain unannotated examples, but some system did provide some annotation. Demo II provided "guidewire" objects borrowed from Maulsby's Metamouse [22]. Demo II and Metamouse used geometric graphics such as lines and rectangles to connect and show relationships between other objects. Other systems like NeXTStep [32] used objects to space out widgets and constrain how objects react to window resizing. "Off-screen" objects are put into an application but not within a visible window and serve to represent data like variables that have no graphical representation. Rehearsal World made use of this technique calling these objects "off-stage actors."

Maulsby applied hint-giving ability to his PBD text editing systems Turvy, Moctec, and Cima [23]. Turvy and Moctec were not working systems but instead were "Wizard of Oz" experiments to see how users reacted to PBD including how they express hints. In Cima, he applied the findings of those experiments into a working tool; though, Cima was generally not powerful enough to actually perform much real work. Cima's task was to recognize features of English text but the text was not structured in any way to facilitate that kind of inferencing. Furthermore, the language that Cima generated could not make full use of hints because its data descriptions were simple pattern matching expressions and not structured to incorporate a hint. As such, Maulsby concluded that hint-giving ability was not as useful as it may seem. Gamut's data descriptions and internal structure should be more accommodating towards hint-giving. Gamut should show that hints can be very powerful when they are thoughtfully applied.

Kosbie mentions in his thesis the use of highlighting to point out important objects calling them "focus" events [15]. No mention is made in the thesis, though, about how to implement, categorize, or use these sorts of hints. Since Gamut will use this style of highlighting for its own hint annotation, it will provide an example implementation and a rationale behind its use. Other hint annotations have been used in Artificial Intelligence. For instance, Instructo-SOAR [12] provides a language for the user to instruct the progress of a AI search engine. The techniques used in AI to provide hints have been oriented toward programming languages. They require the user to read the language provided by the system that describes the problem, then compose a response in another language. Because Gamut is intended for nonprogrammers, though, it must use different techniques.

4.3 Machine Learning

The inferencing performed by PBD systems is a form of inductive learning. In inductive learning, a set of examples is presented to the system which generates an internal model to represent the concepts the examples teach. The results of the system can later be used to apply the concepts to some task. There are many variations to inductive learning. Winston's classic system that learned to recognize the properties of arches was one of the first inductive systems [46]. The differences between techniques depend both on the kinds of examples the system can use and the kind of language the system produces to describe the examples. Annotated examples provide clues that explain why the example is important. Positive examples show concepts that the system should recognize, and negative examples teach the concept boundaries.

4.3.1 Classifiers

The most basic form of inductive learning and one that has been more heavily researched has been classifying examples. In this task, the system is given a list of examples in the form of a set of predicates followed by a classification symbol. The goal is to predict what symbol is correct for a given set of predicates. Many approaches have been taken to solve this problem. The algorithm ID3 [36] builds a decision tree using the predicates as choice points. Another solution to the problem was Mitchell's concept space algorithm [25]. A concept space represents the set of all solutions as a graph and each positive and negative example cuts off portions of the graph until one is left with a single concept in the space. Classifier algorithms have tended to use only examples devoid of annotation. The reason for this is that one would prefer the algorithm to learn a task that even a human might not understand how to teach. These kinds of algorithms could be useful in Gamut for building conditional statements. In a conditional, the system must construct a predicate that chooses whether to branch or not. A classification algorithm can combine predicates based on the properties of objects involved in the branch to decide whether the code takes the branch or not. Determining a good list of predicates will be part of the research to be undertaken.

4.3.2 Learning Procedures

In Artificial Intelligence, learning a task is often called "planning." The system is given a set of goals and operators that can be applied to achieve the goal. The set of operators that the system applies to reach a goal state from a given initial state is the "plan." In Gamut's case, the author already has a procedure in mind and the system's task is to discover that procedure by watching examples. This is called "plan recognition." Instead of producing a plan that accomplishes a specified goal, a plan recognition system is given a list of operations and tries to generalize those operations into a higher-level set of operations. In Gamut's domain, the system would be given the list of actions that the author has produced during a demonstration, the output would be a higher-level description of the behavior those actions represent.

Part of work this research will require is to learn about the various algorithms used by plan recognition systems. Plan recognition is a broad area spanning many domains. Commonly, plan recognition is used to model some aspect of human cognition. For instance, speech recognition or dialog modelling use plan recognition principles to understand human speech. These systems tend to be specialized to their domain to such a high degree as to be relatively useless to apply any of their techniques to Gamut. On the other hand, simpler systems such as Van Lehn's SIERRA [43] which models how a student learns how to perform subtraction can be quite useful. Van Lehn argues that a student learns multiplication by learning and using the special annotations that show where work is performed. By crossing out digits and writing carries, a teacher shows the student annotations that make the problem tractable. This same "show work" principle can be applied to Gamut and has a direct correlation with the annotations and techniques with which an author will demonstrate the rules of a board game.

Lutz's Pascal tutoring system, PROUST [20], uses plan recognition to help the student debug a program by examining the incorrect Pascal program and determining what is missing or mistaken. Lutz's system could disassemble a program into a network of control structures whose nodes represent the conditions that determine which path is followed. By analyzing the structure and comparing to known programming idioms, PROUST could determine what student programs are meant to do and find errors. Gamut will need to have a similar ability to structure programs so that code fragments can be easily compared and analyzed. One advantage Gamut has over PROUST is that the system is writing the code so it can be made more amenable to computation from the start.

5.0 Plan of Attack

This section presents a set of ideas that address the problems and challenges set forth in section 3.0. The main point of these ideas is to make it possible for the author to express clearly all needed constructs. This involves making key abstractions concrete and manipulable. The first section introduces the new techniques Gamut uses to improve an author's expressiveness. The hardest things to demonstrate are those that are most difficult to express. It is important to provide the author with the right set of tools that decompose the domain into the proper level of detail so that expressing any part of a board game never seems daunting. The second section concerns how the system may actually be implemented. Ideas about the inferencing algorithm and the data structures of the system will be described.

5.1 Expressiveness

Because our medium is limited to typical means of computer input and output, we will use direct manipulation for most author communication. Using direct manipulation has three parts. First, there are the objects that the author manipulates. To allow the author to manipulate abstract properties such as modes, randomness, sequences, and constraints, Gamut will provide graphical objects that represent these abstractions. These objects include the "deck of cards" metaphor and other forms of "annotation objects." Second, there are the different views that the author uses to see various aspects of the work in the proper context. The main "window" view is augmented in Gamut by a timeline view and a counting sheet view. Finally, there is the manner in which the author manipulates objects. These include common direct manipulation techniques such as selection and dragging, but include some new techniques such as highlighting to indicate hints and "nudges" used to demonstrate and edit behavior.

5.1.1 Annotation Objects

Annotation objects will be used to represent board game abstractions. An annotation object is, in fact, a real object, but it is invisible when the game runs. The player only sees the effects of an annotation object indirectly. The author, however, does see the object when the application is edited and can manipulate it directly. Gamut will use five kinds of annotation objects: playing cards, mode switches, guide objects, behavior icons, and temporal ghosts. The guide objects are borrowed from the "guidewire" objects in Demo II [5] and MetaMouse [22]. The others are new and will be tested in Gamut for the first time. The constraint objects as well as playing cards and mode switches will be objects the author must create and assemble to use in a game. The other abstractions, behavior icons and temporal ghosts, are visual devices the system will create automatically to help the author use the system more efficiently.

5.1.1.1 Playing Cards Metaphor

Most everyone is familiar with an ordinary deck of cards. Many existing board games have custom decks of cards specific to the game. Gamut will also allow the game to use cards automatically to provide events such as moving a monster or causing the board to change. In computer science terms, a deck of playing cards is a memory abstraction. The deck is a list of objects. A deck can be ordered or unordered (shuffled), and cards can be dealt from the top of the deck. Cards can also be placed into a deck from an outside source such as another deck. Typically, the author will create custom cards by specifying graphics and properties using a card building editor. The data stored in a card can be numbers or graphics as well as references to other cards, board tiles, or pieces. A deck of cards can also have a graphical look and may exist as an object distinct from any one card. Decks will be used to maintain aggregate properties of cards such as the number of cards in the deck or total sum of some property of all the cards.

5.1.1.2 Mode Switches

Modes are ubiquitous in board and video games. Gamut will display mode objects as a row of small icons; each icon represents a mode that is observed in the game. The simplest mode switch is a boolean toggle with two panels representing on and off states. A mode object is trained by demonstration. The author switches a mode by clicking on one of the icons then demonstrate how behaviors and objects should change in response. Likewise, to cause a mode to switch interactively, the icon is pressed during a response demonstration. One very common set of modes that Gamut will provide is a multiple player switch. These modes indicate which player is currently playing. This switch can be used to control whose pieces are active and which windows will allow input. Some high-level objects come with a mode switch attached. For instance, a deck of cards object will have a switch for shuffling.

5.1.1.3 Guide Objects

Guide objects represent spacial proximity and location. Rectangles can act as placeholders for the positions where pieces are to be placed on the game board. A set of "tracks" made of lines can be drawn to show the path that a monster will follow. Likewise, the speed and direction of the monster can be shown by drawing a vector line out of its center and demonstrating how it moves along the vector. These constraints are assembled by creating guide objects from a palette and directly assembling them together with objects already in the window.

5.1.1.4 Behavior Icons

To present behavior so that it can be manipulated, Gamut displays icons directly above the objects that they affect. Each kind of behavior is categorized by an event which is represented by a unique icon. This icon is placed above the objects that are affected by the behavior. Behaviors are considered to affect an object if one of the behavior's data descriptions names the object. By selecting the behavior icon, a dialog box describing the behavior can be presented to and edited by the author. The icons themselves will be treated much like other objects; one can select, copy, paste, and delete behavior icons. By copying a behavior icon to another object, the system will try to mold that behavior so that the new object will behave the same way as the original object. When the behavior's original object is too different from the destination object to make a clean match, the system will alert the author and carry on a dialog to determine the author's intentions.

5.1.1.5 Temporal Ghosts

A source of annotations that has been traditionally neglected concern the recent history of changing objects. In games and animation, recently past states are often as important as the current state of an object. For instance, to show the direction of a moving object, Gamut will leave traces of the original position of the object before it moved. This techniques makes it possible to relate the previous position of the object with the new position. Movements in Chess, for instance, depend strongly on where the piece starts. By keeping a "ghost" of the original object around, the author can show connections between previous positions and the current position. We are calling these objects "temporal ghosts" or "ghost objects" because the system leaves dimmed, translucent images of the recent past in the window for the author to manipulate. Temporal ghosts can also be applied to property changes such as color changes. The previous values of a property will be retrievable through the property sheet by clicking on a ghost arrow. The previous value will then be shown in the property sheet in the same ghost shading as objects in the window and can be manipulated the same way as regular properties.

5.1.2 Views

Part of being able to manipulate new abstractions is the availability of the right context in which to view them. Most pseudo-objects will be constructed directly in the game's various windows but some concepts need to be represented in different views. Gamut will have two secondary views. The first is a timeline view where events and action are displayed in sequence. The other display is the "counting sheet" which is like a spreadsheet which automatically knows how to count various properties in the game.

5.1.2.1 Timeline

A timeline feature has been provided in several other tools. Macromedia Director [21] uses a "score sheet" technique where the author lays out actions in time. Cypher's KIDSIM [2] uses a clock metaphor which the author starts and stops to run the simulation and can be used to reverse time. Gamut will use a timeline to provide feedback to the author. A behavior is displayed in the timeline when it is executed. Each part of the behavior including the event and each action are displayed separately to allow the author to manipulate them individually. Since timing is normally not important to board games, the events will not be lined up in time as they would be on a score sheet.

The timeline will mainly be used for four purposes. First, the author can select actions and events in the timeline and flash the affected objects. This can be helpful for debugging. Second, the items can be highlighted to use as hints. Sometimes one action can act as a condition for another behavior, the timeline lets the author specify such relationships. Third, the timeline can be used to backup execution. When behaviors are automatic, the behavior may execute several times before the author can respond to a problem. Using the timeline, the author can rewind the events to the point where a problem occurred to make corrections. Finally, the timeline can be use to control the number of ghost objects. The number of ghost objects displayed at any one time will be based on heuristics. When the author needs to reference ghosts further back in time or wants to make all ghosts disappear, the author adjusts the timeline.

5.1.2.2 Counting Sheet

The counting sheet is a list of boxes arranged in a column that can hold numbers. Each box is given string name like a property sheet. The cells of the counting sheet are much like a spreadsheet in that the author will be able to perform arithmetic by assigning values to cells and forming constraints. The counting sheet's main use will be to count things that are important to the game. For instance, if the number of pieces a player owns is important, then the author selects all the pieces and makes an entry on the counting sheet for that number. The system guesses a data description for the set of selected items and the author can later use that number as the basis for other rules. If the number has parameters or other constraints, the objects that affect the cell can be highlighted. The author may also fill a cell directly by picking the proper numeric property using a dialog box or typing in the constraints by hand. Once a cell is filled with a data description, the system will maintain the number automatically.

5.1.3 Input Techniques

Gamut will use the direct manipulation paradigm for virtually all interactions. Creating, moving, and manipulating objects will be performed by directly clicking on objects with the mouse pointer. In order for the author to demonstrate behavior, Gamut will augment standard direct manipulation with two techniques. The first technique is called “hint highlighting” and allows the author to point out objects, properties, and events that affect a behavior but cannot be reasonably expected for the inferencing system to guess. The second technique, called “nudges,” is actually a set of techniques which the author will use to build and correct behavior.

5.1.3.1 Hint Highlighting

When a behavior is demonstrated, the objects which affect the behavior are not always clear. In particular, when an object should be used in the predicate of a condition or the event of a behavior, the object can not always be discovered by the system. The system cannot consider all objects and all possible conditions because for even small applications there are too many different combinations to be able to generate and test them all. Gamut will provide a special form of selection called “hint highlighting” to allow the author to indicate objects that are important in a demonstration. By allowing the author to highlight important objects, specifically those objects that affect the behavior but are not necessarily affected by the behavior, the system can reduce the amount of inferencing because the highlighting can reduce the depth of the search. Highlighting is used in tandem with nudges. The specific ways that highlighting will be incorporated in demonstrations will be mentioned in the next section.

5.1.3.2 Nudges

“Nudges” are the technique that Gamut will use for demonstrating behavior. The author assumes that Gamut knows what behavior is supposed to occur but makes mistakes from time to time. When the system makes a mistake, the author “nudges” the system to make a correction. The corrections may include demonstrating new behavior, augmenting previously learned behavior, or making behavior stop. The basic means for nudging is to let the game execute and to steer the execution toward not implemented features. When the author sees an opportunity to demonstrate something new, one presses the escape key to bring the system out of run mode. At that point, the author can manipulate objects to force new behavior upon them or use the two special nudge buttons.

The first nudge is “Do Something!” and indicates that the author has just performed an event to which the system was supposed to respond. Sometimes, an inferred behavior can be too specific. For instance, the author may demonstrate a behavior on a single object but intends that the behavior be used on a class of objects. If an object does not respond to an intended behavior, the author selects the object (or objects) and presses the “Do Something!” button. This forces the inference system to do a search, looking for a behavior that the author means to have happen. To narrow the search, the system has the last few events that the author executed. If Gamut finds a suitable behavior, it will apply the behavior to the selected object by generalizing its events or actions to operate over a larger set that at least includes the selected object. When the system cannot find a reasonable response to the author’s “Do Something!” request, the system asks the author for help. The author may choose the behavior out of a list, or manipulate the object as the behavior should have.

The other nudge is the opposite of “Do Something!” and is called “Stop That!” which is used to indicate negative examples. When an inferred behavior is too general, it causes extraneous actions to occur. To remedy this, the author selects the objects that did more than was expected, and also highlights the condition that makes the current situation different from the situation that the behavior was originally demonstrated, then clicks the “Stop That!” button. The one negative example indicated by the “Stop That!” press is combined with the positive examples created when the behavior was first demonstrated and possibly the other successful times that the behavior had been invoked.

The difference between run and build mode and stimulus and response mode has been discussed in previous sections. Gamut will keep these two interactions totally independent. A common mistake is to relate build mode with response mode and run mode with stimulus mode. By separating these modes, we enable showing stimuli based on events that are not input events, and showing responses that could be more easily demonstrated in run mode. In Gamut, run and build mode will be selected by a toggle switch. The different behavior of the system while in the different modes will make it apparent to the author which mode is currently active. Other kinds of cues such as changing the background color of certain windows or changing the cursor to look different can also help distinguish the modes.

Response mode is entered by pressing the escape key. Response mode ends when escape is pressed again. Since this can be confusing, special coloring and other cues will need to be added to keep the author aware that response mode is active. Gamut totally removes the need for a stimulus mode. Stimulus will be inferred by the author making connections such as using ghost cursor arrows and highlighting events in the timeline.

Hint-highlighting objects in response mode can be used by the author to indicate conditions on which the behavior depends. A single demonstration can only show one path through a branch, though. This means that highlighted items must be saved with the behavior until its use can be justified later. Normally, a hint will only take effect when the author contradicts the system using "Stop That!" At that time, the system will examine items highlighted during previous demonstrations and determine how they have changed from their original state. These changes are used to build the predicates that control branching behavior.

5.2 Implementation

Gamut contains a set of algorithms collectively called the inferencing algorithm. These algorithms are roughly divided around the tasks that Gamut performs. For instance, there are algorithms for looking for paths among the objects, and algorithms for determining the predicate of a conditional expression. The presentation of each algorithm is the purpose of the thesis document. This proposal will describe the overall task: converting the author's demonstrations into the rules of a game.

Generating rules requires inductive learning. Given a set of examples, the inferencing algorithm learns how to referee a game. This task has the three components mentioned in section 3.2.2: a knowledge representation for cataloging and executing the rules, the algorithm which learns the knowledge, and an interface for editing the rules.

5.2.1 Knowledge Representation

In Gamut's framework, rules are the code that specify behavior. The code is both a means for storing inferences as well as an executable run-time language. This gives the language two mediums. The first is the static representation which can be analyzed by the system and modified as the system makes inductions. The other, dynamic representation is used by the system to hold the state of a running game.

The basic structure of the static language is comprised of behavior, conditions, and actions. An action changes the state of the application objects. A behavior causes actions to occur in response to events. Conditions cause different actions to occur under different circumstances. In chess, a behavior is clicking and dragging to move a piece, a condition is whether the move is legal, and an action is the changing of the piece's position.

The behavior and conditions can be thought of as the structure of the game's rules. The structure tells the system what will happen, but it does not say what objects will be affected. For this there needs to be parameterization or data description of the objects involved in the behavior. These data descriptions can also be conditional since a behavior can affect different objects. Data descriptions are generated by looking at the circumstances of the demonstrations and attempting to determine what the author means to affect with the rule.

The run-time language uses the static language as code but includes two other features. First, there are the actual objects that constitute the game's components. These objects are derived from basic types that come from the menus and palettes. The other run-time feature are events. Events are the causes for behavior. When the mouse is moved or a key pressed, events are generated. These events are used to cause behaviors to run. Actions also cause events to occur. When a piece is moved or a property is changed, an event signalling the change in state is sent to the rest of the system. This allows behaviors to be based on internal phenomena and not solely on the input devices.

An object-oriented frame system will be used to store information about graphics and system state. These objects will also be used for the executable representation of the game. Frames will also be used to represent language's structures such as behaviors and data descriptions. The idea is to create a set of basic building blocks which can be easily assembled into a game. Yet, the building blocks must be decomposable. Every condition, data description, and every other item can be modified individually. This provides an ability to transform that will allow a greater number of behaviors to be inferred.

Implementing “Do Something!” will require the ability to search for code using a variety of criteria. When “Do Something!” is pressed, the system can use the partial list of events as a key for searching. Likewise, if the search is unsuccessful at first and the author provides more information, the system must search again using the new information. No previous PBD system has placed this great an emphasis on search and code reuse. Other systems have been content to relearn knowledge and treat each example as though it were a brand new concept to be learned.

5.2.2 Inferencing Algorithm

Winston categorized the spectrum of learning environments [46]. The first end is learning by pure examples. In this category, the author makes no attempt to explain and only provides a set of positive and negative examples. This is called pure inductive learning, learning by watching, or programming by example. At the other end of the spectrum, the system is told the concept outright. This is called learning by being told, or simply programming. Our task lies in between. The author is given opportunity to help the system as much as desired, but does not require the author to specify every detail.

We allow the author to give help because machine learning literature shows that pure inductive learning algorithms cannot be taught complicated lessons [43]. Our task requires inferring, among other things, iteration, conditions, and connections between objects. This implies that pure inductive methods are going to fail if applied to our domain. In order for learning to go smoothly, the author must do two things. First, the author must demonstrate only one new concept at a time. This equates to one condition (disjunction) added to a behavior. Hypothetically, this constraint can be reduced somewhat as long as the conditions are independent. The second part of good learning is the “show work” principle. This principle concerns the use of annotation objects to hold partial results and to connect seemingly unrelated events. When the author “shows work,” all of the annotation objects are presented to the system as well as the normal objects and the system must infer the relationships between all the objects.

In Maulsby’s work with Turvy [23], we see that although some authors are quite expressive when they give hints and point out annotation objects, others are quiet and do not give hints unless prompted. When the author refuses to show the work required to accomplish a task, the system will first try to infer the omitted annotation objects using heuristics. Unfortunately, annotation objects have the property that they could be anywhere and do anything. When heuristics fail to generate an acceptable solution, the author will need to be prompted to show more work. This will be a situation dependent dialog that will ask for information based on the kind of connections that are missing. For instance, if the author moves a piece but there is no indication of why that move is legal, then the author will be asked to draw or select something that makes obvious why one position is allowed but another is not.

The point at which the system has to ask the author for help is an impasse. An impasse is a point where the algorithm’s guessing mechanism runs out of reasonable choices. The message sent to the author is based on the type of the impasse and the data the algorithm was trying to infer at that time. Identifying all potential types of impasses and providing a reasonable message for each is part of the project’s research. Should the author be providing all needed hints and annotation objects in the examples, the system should never reach an impasse. Instead of asking for information, the system will record all hints; although, it may not know how to apply them at first and incorporate them later as the author makes corrections. Recency and the number of times a hint is given will be a factor in determining whether a hint should be applied to a given situation.

A critical point for the inferencer is determining conditional execution paths. A condition cannot be shown with a single example because each path much be demonstrated entirely (though with clever use of the timeline the author may save some work). With two paths the inferencer must determine the point where the behaviors rejoin to become one. Being able to compare procedures helps at this point. By determining the differences between the traces, the procedures can be joined. This mechanism, however, can only learn one choice point at a time. Should the author try to demonstrate more than one, the inferencer is more likely to be confused and will likely generate seemingly odd results until the author corrects the matter by providing more examples.

Data descriptions will be generated in two ways. First, in order to learn from as few examples as possible, the system will attempt to apply a set of heuristic rules. These heuristics will be based on the type of the object described, how the object is used in the structure, which objects have been highlighted as a hint, and as much other context as can be determined. Graphical objects will fair the best in this scenario as there have been a large number of systems that have categorized the kinds of descriptions graphical objects seem to use most. If the initial guess is proven wrong, then the system will try again with the contradicting example as a new data point. When the heuristics are depleted, the system will move to a disjunction learning system. The purpose will be to determine a small number of disjunction statements that can account for the description. This process may be helped by having the author select all objects on the screen that would fit the expected description.

One especially important piece of context that must be considered is that there are two forms of data descriptions. One is an inclusive description. The purpose of an inclusive description is to name all objects for which a behavior is possible. When the player clicks the mouse over a rectangle, that rectangle is described by an inclusive description. The other form is an exclusive description that singles out the object or objects on which an action is to be performed. Changing a rectangle's color to red requires an exclusive description for the rectangle. Though similar. these two forms of description have considerably different goals and require separate heuristics to generate them.

Assembling the structure and data description will be kept as simple as possible. The structure has obvious points where descriptions are used and these will be filled accordingly. Though there is influence from the structure on the data descriptions, the influence of data descriptions on the structure will be kept minimal. Generating branches and other new structures based on attempts to learn a data description appears to be infeasible at this time. Structure will only be affected by multiple demonstrations that show a behavior performing different actions.

5.2.3 Feedback and Editing

Buffer commands like cut, copy, paste, and undo have become synonymous with direct manipulation. The commands have become so popular and omnipresent in graphical interfaces that omitting them would be a fatal mistake. Buffer operations should be able to be performed on anything that can be selected by the author. This means that behaviors as well as graphics will have to have a reasonable set of semantics that define how they are copied to different contexts. Being able to copy a behavior from one object to another has never been tried before in a PBD system. Potentially, the author can save a great deal of time by taking a behavior, making a copy of it, and pasting it onto another object.

To manipulate behavior beyond aggregate buffer commands, Gamut will use English-like sentence dialogs similar to the interface found in Marquise [30]. In the future, we would prefer to see a better, more pictographic representation used because studies have shown that users understand diagrammatic representations better than text [26]. Since this is not the thrust of our research, though, the text method should prove sufficient.

5.2.4 Using Amulet

Amulet provides several conveniences that make tools like Gamut easier to build. Amulet is a user interface development system (UIDS) that makes an excellent starting point for basic widget and graphical programming. Amulet provides a retained graphics model so that all drawing is handled by the system.

The executable form for the rules should be straightforward to represent in Amulet. Amulet's object system should be sufficient to use directly as Gamut's object system. Behaviors correspond to interactors [28], and the actions correspond to command objects [29]. Using command objects, a new abstraction specific to Amulet, may also be a simple way to implement Gamut's timeline. Data descriptions do not exist in Amulet so they will have to be designed and implemented. Likewise, the conditional portions of behavior are not well modelled in Amulet requiring another addition. The basic approach will be to design a set of interlocking components made of objects that can be assembled into a program. Certain common behaviors will be built from existing objects in Amulet and fitted with special constraints to make them easier to customize by the inferencing algorithm. One main drawback to Amulet objects is that they are difficult to search and compare. Techniques to summarize the information stored in objects so that one can make comparisons will be needed.

Amulet will require some other extensions in order to build Gamut. First, animation and timing interactors will need to be added. Also, in order to "shadow" objects to make them less visible, a dithering mechanism will be included. A commercial system would not permit the needed extensibility this project requires. A research system like Amulet is much preferable as its source code is available and relatively easy to extend and modify.

6.0 Scenario

As a sample demonstration, we will show how to build a simplified version of the game, Number Munchers [34]. Number Munchers is a commercial educational game that teaches young children basic arithmetic and properties of numbers. In the game, the player controls a PacMan-like creature, but instead of eating dots, the creature eats numbers. The goal for the child is to have “the muncher” move to the correct tiles and eat the number on each. Though the original game puts arithmetic statements into the tiles and can vary the goal, in our simplified version, the tiles will contain single digits and the goal will be to eat the even numbers. Eating an odd number causes the player to lose one “life.” To make things more interesting, there is also a monster who the muncher cannot touch that travels the board randomly. The original game has more features such as “safe spots” and monsters that move in different patterns, but our simplified version has enough features for exemplary purposes.

The phases for building this game will consist of first building the parts of the game, then adding behavior to the parts. Player interaction will be shown first then the monster movement will be shown. Finally, end of round and end of game behaviors will be shown.

6.1 Assembling the Parts

The process begins with building the board and characters of the game. The board is a grid of tiles each containing a random number. To fill the tiles, the author uses the deck-editor and assembles a deck of cards. The author specifies ten cards numbered zero through nine each number drawn in a large font in the center of the card. Thinking ahead, the author also installs a boolean property called “even” on each card which marks all the even cards true and the others false. For the actual tiles, the author draws a grid of squares on the background window. Moving to the characters, the author draws a picture of the muncher and the monster using the bitmap editor. Finally, to make the “extra men” objects, the author makes copies of the muncher and puts them where desired.

The background is not quite complete. To assign numbers to the tiles, cards must be dealt by the system, one to each tile, at the beginning of the round as shown in Figure 1. Switching the system to run mode at first produces no

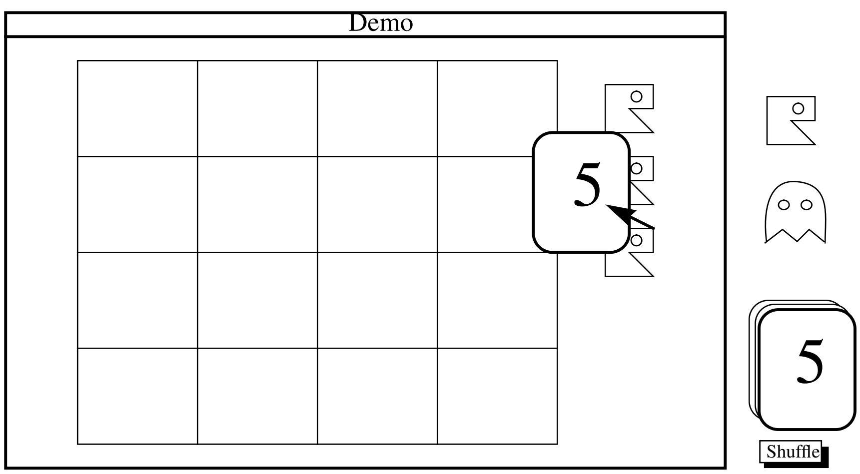

Figure 1: Initial Set-up showing author demonstrating how to assign a card to a tile.

response from the system since no behavior has yet been demonstrated, so the author presses the escape key to nudge the system into action. The author shuffles the deck, selects a card, makes a copy of it and places the copy onto the first tile. Then the author presses escape once more to exit response mode and finish the demonstration. One tile is now filled, but the system has not filled any other tile as it should. To correct this, the author presses escape again, selects the empty squares and presses the “Do Something!” button. Since the system only has one behavior defined at this time, the only difficulty is finding a match between the selected objects and the behavior. It will find that the type of the destination for the card matches the type of the selected objects. The data description for the position where the card is placed is thus changed to include all board tiles. So, the system then copies the card to all squares, but the author notices that all the numbers are the same. Apparently, the system is not shuffling the deck between card placements (the author can confirm this by examining the behavior description). The author highlights the shuffle button on the deck and presses “Do Something!” again. The highlighted shuffle button matches against the shuffle the deck command in the behavior. Because the author has selected multiple objects and the shuffle command originally only executes once in the behavior, the inferencing algorithm assumes the shuffle should occur once per object. Without the hint, the inferencing algorithm would probably not be able to find a proper response as the situation would look identical to the prior one.

6.2 Showing Player Interaction

The next demonstrated behavior provides player interaction. To move the muncher, the player clicks on a square toward which the muncher then travels. If the player clicks on the square at which the muncher is already located, the muncher eats the number in that square. First, the author will demonstrate how to eat a number by clicking on the square where the muncher is located. Nothing happens at first, so the author presses escape and deletes the number inside the square. If an odd number is eaten, the player must lose a life, so the player moves the muncher in build mode to a square with an odd number and demonstrates eating. The author augments the behavior by highlighting the “even” property of the number and deletes an extra muncher life. The number had actually been deleted but the author could still highlight one of its properties because a manipulable ghost object was left in the number’s place. The author finishes the demonstration by moving the muncher to an even number and clicking again. The system will delete the player’s extra man because it cannot distinguish the situation, yet. By selecting the deleted extra muncher and pressing “Stop That!,” the author corrects the system. The author had already highlighted the “even” property in a previous demonstration so it need not be selected again.

To make the muncher move during game play, the author clicks on a destination square. The system does not know what to do so nothing happens. The author first demonstrates the path the muncher will follow by drawing a pair of perpendicular arrow lines that connect the muncher with the destination as shown in Figure 2. The author can

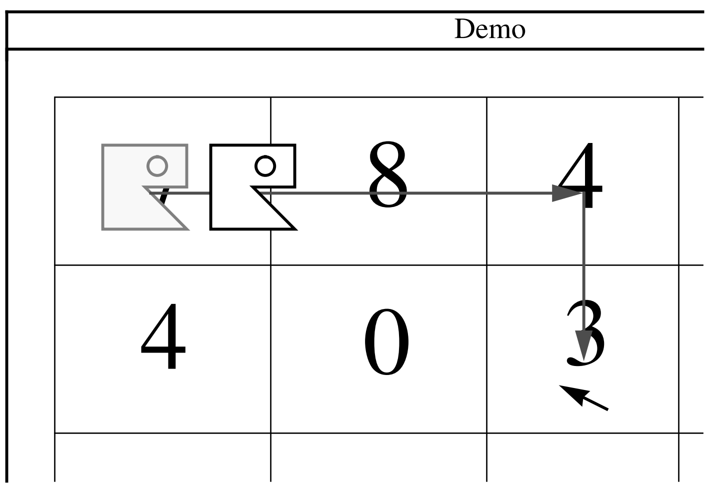

Figure 2: Demonstrating how the muncher moves.

test whether the system understands how the lines are placed by moving the muncher by hand and clicking on a square in a different direction. To make the muncher take a step, the author highlights the direction line and drags the muncher forward along its path. The system still does not know that it must control the muncher autonomously so the muncher remains stationary at first, one step along the path. To correct the system, the author selects the muncher and presses the “Do Something!” button. The connection between the two arrow lines can be detected automatically so the muncher will turn the corner without incident.When the muncher reaches the end of the second arrow, the author demonstrates deleting the arrows.

6.3 Showing Monster Interaction

Monster movement can be demonstrated the same way as the muncher. The main difference is that the direction must be chosen by the system. To do this the author builds a second card deck with four cards. On each card the author draws a arrow line for each of the four cardinal directions. To move the monster, the cards are shuffled and one is drawn. The arrow on the card is copied to the monster’s location and the monster is made to take a step along the arrow's direction. When the monster reaches the end, the arrow is deleted and another is drawn from the deck. To keep the monster from walking off the side of the board, the author moves the monster to the side of the board, but instead of letting the automated behavior run, the author sets the monster movement deck to have the author choose a card whenever it is supposed to shuffle. This way the author can "stack the deck" and force important events to happen. The author picks the direction pointing off the board. When the monster starts to follow it, the author highlights the arrow and the background behind the squares and presses the "Stop That!" button. This operation is repeated at least once on a different side of the board to make sure that the system understands.

The monster will destroy the muncher if the two make contact. This is shown by colliding the monster and the muncher using the same forced card approach that demonstrated walking off the board. The first moment the two touch, the author stops the action, highlights the muncher and the monster, and demonstrates the muncher losing an extra man.

6.4 Finishing It Up

The game is almost finished. The final step is to indicate how each round ends and how the game ends. When a round ends, the muncher and the monster are moved to their starting positions, and the board resets. When the game ends, the same actions occurs but also the extra men are created in full. The game ends when the muncher runs out of lives. To count the number of lives the author selects the extra men objects and makes an entry on the counting sheet as shown in Figure 3. The sheet builds a data description for the selected objects and keeps track of the

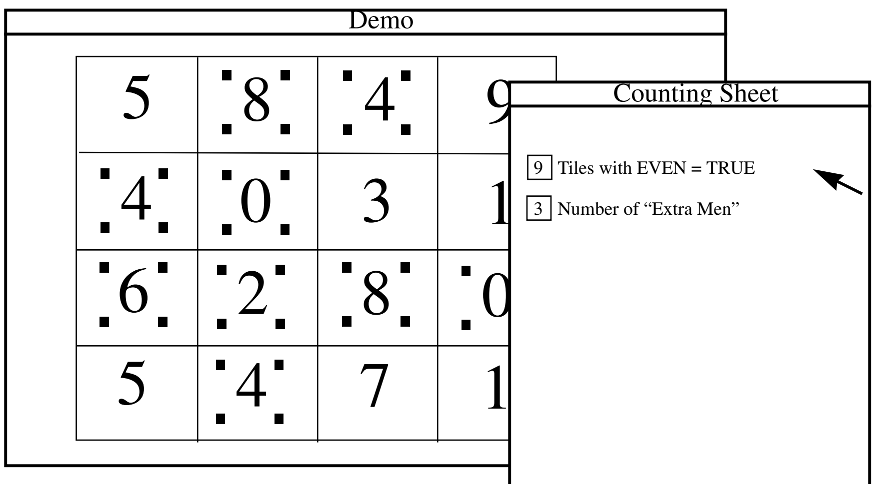

Figure 3: Counting the number of even-numbered tiles.

number continuously. The author does the same to count the even numbered tiles which triggers the end of the round. To distinguish the selection from the odd numbered tiles, the author also highlights the "even" property. When the number of even numbered tiles reaches zero, the author demonstrates the next round must be activated by highlighting the number on the counting sheet and performing the end of round actions. Instead of repeating these actions for the end of game condition, the author makes a copy of the behavior icon that was created near the even numbers counter and pasting it on the extra muncher counter. Restoring the extra munchers is appended to the behavior by switching into response mode and redrawing the munchers back.

7.0 Contributions

The main thrust of Gamut's research and thus the main source of contributions is in the techniques for authors to better communicate their ideas. As a research project, Gamut has much in common with engineering and design. Contributions having the most impact will be ones that people will use to design and build other tools for non-programmers. These are the contributions I expect from this research:

- Using nudges as a more implicit manner of gathering demonstrations. Past systems have required a rather lock-step style where the author would have to spell out exactly what was being demonstrated with every input. The nudges technique seems to be more natural and to require less forethought for the author. Using nudges, examples can still be decomposed into stimulus-response style examples and negative examples are available, but the multitude of modes that have to be used in other systems have been eliminated.

- Interpreting the focus hint. Gamut will be the first tool to use highlighting to allow the author to give hints. How this ability is incorporated into the system and how well authors are able to understand their use is a major research area.
- The deck of cards metaphor. Previous PBD work could infer actions based on lists of objects, but the deck of cards is more powerful. The deck provides not only data collection and sequencing duties, but also it provides for randomization and a convenient way to move data from one place to another. Most everyone is already familiar with what a deck of cards is and the actions that can be done with it.
- Mode switches. A mode switch is an easy way to control large scale changes in behavior and graphics. The player mode switch is used to control turn order and which pieces can be manipulated. The player switch also can be used to distinguish moves so that automatic analysis can be taken to reveal cases such as when a player has no legal moves available or which move will maximize a given function.
- Two new views. The first is a timeline for displaying events. This timeline is unlike other system's timelines in that the events can be highlighted for hint giving purposes and serves as a way to control ghost objects. The other kind of view is the counting sheet which is somewhat like the mode window in Marquise, but supports more general computations.
- Ghost objects. Without ghosts or similar annotations, past positions and properties of objects have to be inferred by the system without help from the author. With the new ghost objects, it will be possible for the author to demonstrate explicit, more easily inferred connections.
- Behavior icons. The icons turn behavior into concrete objects that can be manipulated with the familiar cut, copy, and paste operations of direct manipulation. These will let the author make corrections without having to see the behavior directly. Also, by copying the icon to other objects, they let the author make new behaviors without having to re-demonstrate.
- End user programming. As an actual tool, Gamut will be able to be used by nonprogrammers to make working applications. If only viewed as a prototyping tool, the degree of programming performable in Gamut without code should surpass many available tools, and cost a great deal less.
- Gamut's presence as a real tool should be useful as a workbench for future research. Because Gamut's interface will be more or less complete, other researchers need not reimplement their own interface and can either modify or extend the existing one in Gamut. Furthermore, non-interface issues such as inferencing can be explored without having to build a whole new system.

8.0 Schedule

This thesis will involve a great deal of engineering. Most of the work revolves around the main project, Gamut, which is a software tool. Though the most intellectual effort goes into designing and understanding how such systems are used, there will still be considerable time spent programming the system. The phases of work are as follows:

- Pre-testing - 1 month. This consists of finding subjects to help with paper prototype testing. This will be an informal study to work out unforeseen bugs before coding starts in earnest. The testers will be given an informal demonstration of the system and be allowed to ask questions at first. As the design becomes more refined, the testers will be given a more formal demonstration and be asked to answer questions on paper.
- Programming - 8 months. This involves building the actual system. Coding will begin with the interface and basic interaction techniques. When the interface is complete, I may perform some "Wizard of Oz" style tests to further improve the interface. This will be followed by adding the inferencing algorithm and heuristics and testing at various milestones. To test Gamut's abilities, the following sample applications will be built entirely by demonstration:
  - Chess
  - Number Munchers
  - A larger game, perhaps of the fantasy genre

  The purpose of each application is to test Gamut's range. Chess is a strategy game that relies heavily on annotation objects to explain the rules. Numbers Munchers is a fun but simple commercial educational game that teaches arithmetic. The larger game is meant to test Gamut's ability to facilitate a large project. Building a larger program requires Gamut to be able to load and save as well as provide decent editing and navigation through the project. Also, the larger project will demonstrate whether the new techniques become unwieldy on large tasks. These applications will also test Gamut's robustness and to gauge how well it can function outside the research environment.

- Formal User Testing - 3 months. Here we find nonprogrammers willing to test the system. At this phase, we will also announce the tool to interested parties at CMU to see if anyone is willing to use the system. The formal testing will consist of asking subjects to build several styles of games in order to see how well our new techniques hold in practice.The user testing phase will be the ultimate test of Gamut's success. Should the user testing go poorly, more iterations will be scheduled to handle the problems.

  The simplified Number Munchers game described in section 6.0 will likely be the test application. First, subjects will be taught how to use the system and be lead through some example exercises to help them understand what things are possible in Gamut. The teaching phase should last one or two hours and will end when the subject feels confident to move on. The subjects will then be asked to build Simple Number Munchers in four to five hours. Though some subjects will likely not finish the entire program, we hope that most will and those that do not finish will at least be able to build some of the required behaviors. Of necessity, most of the subjects will be nonprogrammers to make sure that Gamut reaches its target audience.

- Writing the Thesis - 6 months

Bibliography

[1] Bill Budge. *Pinball Construction Set*. Exidy Software.

[2] Allen Cypher and David Canfield Smith. *KIDSIM: End User Programming of Simulations*. Proceeding CHI'95: Human Factors in Computing Systems, May 7-11, 1995, pp 27-34.

[3] Allen Cypher. *Eager: Programming Repetitive Tasks by Example*. Proceeding CHI'91: Human Factors in Computing Systems, New Orleans, 1991, pp 33-39.

[4] *Donkey Kong*, Nintendo, 1981.

[5] Gene L. Fisher, Dale E. Busse, David A. Wolber. *Adding Rule-Based Reasoning to a Demonstrational Interface Builder*. Proceedings of UIST'92, pp 89-97.

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

[7] Laura Gould and William Finzer. *Programming by Rehearsal*. Palo Alto Research Center, Xerox Corporation, 1984.

[8] Leslie Grimm, Dennis Caswell, and Lynn Kirkpatrick. *Playroom*. Broderbund Software, 500 Redwood Blvd., Novato, CA 94948-6121, 1992.

[9] Leslie Grimm and Lynn Kirkpatrick. *Treehouse*. Broderbund Software, 500 Redwood Blvd., Novato, CA 94948-6121, 1993.

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

[11] Richard Hamblen, *Magic Realm*. The Avalon Hill Game Company, Baltimore, MD 21214, 1986.

[12] Scott Huffman. *Instructible Autonomous Agents*. Ph.D. thesis. Computer Science and Engineering Division, University of Michigan. 1994.

[13] *HyperCard*. Apple Computer Inc., Cupertino, CA, 1993.

[14] Daniel H.H. Ingalls, "The Smalltalk-76 Programming System: Design and Implementation," *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, January, 1978.

[15] David Kosbie. KATIE: Architectural Support for Programming by Demonstration. Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, not completed.

[16] David Kurlander and Steven Feiner. Editable Graphical Histories. IEEE *Workshop on Visual Languages*, pages 127-134, Pittsburgh, PA, October 1988.

[17] Jill H. Larkin, Ruth W. Chabay, eds. *Computer-Assisted Instruction and Intelligent Tutoring Systems: Shared Goals and Complementary Approaches*. Lawrence Erlebaum Associates, Publishers, Hillsdale, New Jersey, 1992.

[18] *Lemmings*. Psychosis, Ltd., South Harrington Building, Softon Street, Liverpool 134BQ, UK, 1992.

[19] Henry Lieberman. *Mondrian: A Teachable Graphical Editor.* Visible Language Workshop, MIT Media Laboratory, November 1991.

[20] Rudi K. Lutz. Towards an Intelligent Debugging System for Pascal Programs: On the Theory and Algorithms of Plan Recognition in Rich's Plan Calculus. Ph.D. Thesis, The Open University, Milton Keynes, England, 1993.

[21] Macromedia Director, Version 4.0.

[22] David L. Maulsby, Ian H. Witten, and Kenneth A. Kittlitz. *Metamouse: Specifying Graphical Procedures by Example*. ACM Computer Graphics, Volume 23, Number 3, July 1989, pp 127-136.

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

[24] *Microsoft Word 6.0a*. Microsoft Corp., Redmond, WA, 1994.

[25] Tom M. Mitchell. *Generalization as Search.* Artificial Intelligence, Vol. 18, 1982, pp 203-226.

[26] Francesmary Mudugno, T.R.G. Green, Brad A. Myers. *Visual Programming in a Visual Domain: A Case Study of Cognitive Dimension*. Proceedings Human-Computer Interaction'94, People and Computers, Glasgow, August, 1994.

[27] Brad A. Myers. *Creating User Interfaces by Demonstration*. Academic Press, San Diego, 1988.

[28] Brad A. Myers, Dario Giuse, et al. *Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces*. IEEE Computer, Vol. 23, No. 11, November, 1990.

[29] Brad A. Myers and David Kosbie. *Reusable Hierarchical Command objects*. Submitted to CHI'96, 1995.

[30] Brad A. Myers, Richard G. McDaniel, and David S. Kosbie. *Marquise: Creating Complete User Interfaces by Demonstration*. Proceeding of INTERCHI'93: Human Factors in Computing Systems, 1993, pp 293-300.

[31] Newstrack. Communications of the ACM, May 1995, Volume. 38, No. 5, p 9.

[32] *NeXTStep and the NeXT Interface Builder.* NeXT, Inc., Redwood City, CA, 1991.

[33] Robert P. Nix. Editing by Example. ACM Transactions on Programming Languages and Systems, Volume 7, Number 4, October 1985, pp 600-621.

[34] *Number Munchers*. Mecc, 6160 Summit Drive North, Minneapolis, MN 55430-4033, 1990.

[35] *PacMan*, Namco, distributed by Midway, 1980.

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

[37] *Reader Rabbit*. The Learning Company, 1987.

[38] Randall B. Smith. Experiences with the Alternate Reality Kit: An example of the Tension Between Literalism and Magic. *IEEE Computer Graphics and Applications*. Volume 7, Number 9, September 1987, pp. 42-50.

[39] Elliot Soloway and James C. Spohrere eds. Learning Programming Concepts. *Studying the Novice Programmer*, section III, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1989, pp 125-180.

[40] *Starblade*, Namco, 1991

[41] F. G. Tresham and Harland Trefoil Ltd. *Civilization*. The Avalon Hill Game Company, Baltimore, MD 21214, 1981.

[42] Francis Tresham. *1830*. The Avalon Hill Game Company, Baltimore, MD 21214, 1986.

[43] Kurt VanLehn. *Learning One Subprocedure per Lesson*. Artificial Intelligence, Vol. 31, 1987, pp 1-40.

[44] *Visual Basic*. Microsoft Corp., Redmond, WA, 1993.

[45] *Visual C/C++*. Microsoft Corp., Redmond, WA, 1993.

[46] Patrick Henry Winston. *Learning Class Descriptions From Samples*. Artificial Intelligence, Chapter 11, Addison-Wesley Publishing Company, Reading, MA, 1984, pp 385-408

[47] David Wolber. *Developing User Interfaces By Stimulus Response Demonstration*. Ph.D. Thesis, Computer Science Department, University of California, Davis, 1992.