Building Whole Applications Using Only Programming-by-Demonstration

Richard G. McDaniel

May 14, 1999
CMU-CS-99-128

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Also appears as CMU-HCII-99-100

Thesis Committee:

Brad A. Myers (co-chair)
David Garlan (co-chair)
Roger B. Dannenberg
David Wolber, University of San Francisco

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center (SPAWARSYSCEN) under Contract No. N66001-94-C-6037 and Contract No. N66001-96-C-8506, and also by the National Science Foundation (NSF) under Grant No. IRI-9319969. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

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

Abstract

Present day tools require a developer to learn complex programming languages to build modern interactive software. However, the effort used to create such software such as games, simulations, and educational software would be better spent not in programming the application's logic, but in providing the engaging background, artwork, and gameplay that keeps users interested. Artists and educators who could produce good material for these applications are often unable to program. Thus, providing a tool that does not require programming skill but still allows a wide range of behavior to be created is desirable.

This thesis has developed techniques that allow a developer to build complete applications without using a written programming language. The foundation of these techniques is programming-by-demonstration in which a developer shows the system what to do by presenting examples of the desired behavior. The techniques include innovations both in interaction and inferencing.

The interaction techniques developed in this research are designed to give developers the ability to express the application's behavior with appropriate detail. The developer can draw *guide objects* in the scene that are hidden to the user. These objects express important relationships and hold the application's data. The developer can also use *cards* and *decks* to represent collections of data as well as to provide randomness. The developer can give the system *hints* by pointing out which objects a behavior relies upon. Also, the techniques include an efficient method for demonstrating examples called *nudges*. Nudges allow the developer to revise behaviors as problems are discovered, and they allow the developer create negative examples as easily as positive examples.

The inferencing techniques allow a broader range of behavior to be generated automatically than prior PBD systems allowed. By using *decision tree learning*, the system can automatically infer conditional expressions where the objects that are referenced by a behavior do not have to be affected by that behavior. Another technique based on *recursive difference* methods allows the system to form and revise arbitrarily long chains of expressions.

These techniques are implemented in a tool called Gamut which has been tested in a usability study to gauge the techniques' effectiveness. The test showed that Gamut can be successfully used by nonprogrammers to build complicated behaviors. Gamut provides a rich medium for expressing a developer's intentions with sufficient inferencing power to create interactive software while requiring minimal programming expertise.

Acknowledgments

I can still remember putting up a slide during the thesis proposal that contained the so-called "schedule." The schedule slide always produces a laugh or two from the audience because all Computer Science thesis schedules at Carnegie Mellon are supposed to total eighteen months. After about 4 years of work on this thesis, I can safely say these were the longest eighteen months I have ever spent. I am thankful that I could spend these long eighteen months with the company of some terrific people without whose help this thesis would not be possible.

Of course, my greatest thanks must go my advisor, Brad Myers. Brad is a long time guru in programming-by-demonstration and was the person who introduced me to the area and inspired this project through the Marquise project which we shared earlier. Brad has been a good friend as well. He has endured my many long arguments over the minutia of this large project and has patiently read my papers to help me improve my writing. He also let me participate in some of his family's life, inviting me to share in their Thanksgivings and other personal moments.

I must also thank my co-advisor, David Garlan, who was kind enough to step in when my advisor situation began to come in doubt. David, though programming-by-demonstration is not in his area of research, nevertheless saw worth in my project and strived to help me improve it considerably. David has an eye for good writing and pressed hard to make my writing meet his standard of approval.

My other committee members, Roger Dannenberg and David Wolber, also deserve my gratitude. They have patiently read through my thesis and offered excellent criticism that I hope to satisfy. Dave Wolber was responsible for the DEMO II project from which many features of this project were derived. Also, Dave took over for David Smith who was originally my outside committee member (but had to leave due to lack of time). My thanks go to Dave Wolber also for taking over for Dave Smith so late in my thesis.

Many others have also helped me maintain my sanity through this arduous task. My officemates Sean Slattery and Howard Gobioff were available each time I finished programming each little feature in the system. Their initial comments influenced many important features of the interface. I also have to thank my other friends who would come in from time to time and help test the system long before it actually worked. I would especially like to thank Michael Duggan who tested the system so many times that he eventually began developing his own style of programming with it. Other friends who have helped were Michael Collins (whose extensive video game collection also helped inspire the work), Bayani Caes, Hiu Ming Lam (Alan), Ben Galehouse, and Rob Miller. I would also like to thank my test participants who took time from their busy schedules to help me finally complete this work.

Finally, I would like to thank my other friends and family who have been there to support me through it all. My family has always provided emotional support and never complained as the thesis dragged on. I am also grateful to the members of Carnegie Mellon's gaming club for their friendship and for letting me satisfy my passion for strategy games. Perhaps someday this work will help people like them create their own games for the computer.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

Computers have become the standard tool for developing almost all forms of human communication. Writers use word processors to capture their thoughts. Artists paint with digital pigments. But computers can be used to produce much more than static media. A computer can provide interactive behavior, where the viewers of the work can become immersed in the developer's creation to form their own experience. The computer can be made to respond to a user's input to achieve worthwhile effects.

With today's tools, programming interactive applications is achieved mostly through written computer programming languages. Programming with these languages has been known to be difficult and requires many years of training for most people. However, nonprogrammers have skills and ideas that would be useful in creating interactive applications. Therefore, it is worthwhile to design and study systems that allow nonprogrammers to build interactive computer software.

This thesis presents techniques with which a nonprogrammer can build interactive software without using a written programming language. These techniques are implemented in a system called Gamut which a developer can use to build game-like applications. The goal is to enable nonprogrammers to create a broader range of interactive behaviors than can be produced using other tools that do not use written languages.

Gamut's techniques are based on programming-by-demonstration (PBD) [21] where the computer is taught new behaviors by having the developer act out examples of the behavior. Gamut extends PBD by giving the developer more channels of communication than simply providing examples of an application's surface behavior. Gamut allows the developer to draw guide objects that show crucial relationships and hold application data. Gamut also supports hints where the developer points out important objects for a behavior that the system would find difficult to find on its own. Gamut uses algorithms from Artificial Intelligence to make its inferences stronger. These techniques make it possible for the system to infer more complex behaviors than was possible in prior PBD tools.

1.1 Gamut's Purpose

Gamut is a testbed for the interaction and inferencing techniques presented in this thesis. The techniques were implemented in a system designed to allow nonprogrammers to create applications. The nonprogrammer in this case is a developer who wants to build an application for someone else to use. Since Gamut is used mostly to make games, this other user will often be called the

"player." The developer uses Gamut's techniques to define the appearance and behavior of the desired application (or game) that the player will later run independently from Gamut. In this thesis, "Gamut" will sometimes be called "the system" to distinguish it from the applications Gamut is used to create.

By implementing Gamut as a real system, its techniques can be judged in two ways. First, the system can be tested with nonprogrammers to show that the techniques are actually understandable. Second, the system's power can be judged by showing that it can create complex behaviors that could not be created by other tools without using a textual language.

First and foremost, Gamut has to be understandable for nonprogrammers. This means that its interaction techniques must be clear and easy to use. Traditional programming constructs that are known to be difficult for nonprogrammers such as for-loops, if-then-else statements, and variable assignment should not be used in Gamut, not even in a graphical or "snap-together" form. Instead, the developer's activities should be concentrated around the application and its behaviors. This makes programming-by-demonstration a strong contender as a programming technique for nonprogrammers. With PBD, the developer uses standard drawing and manipulating commands to teach the system what it should do. Developers should already be familiar with standard direct manipulation editing techniques so learning to use a PBD system does not require much extra effort. Gamut's new graphical techniques and metaphors are also designed to be familiar to people and they use direct manipulation so that they are easy to use.

The second important requirement is to show that the techniques are useful for real-world problems. Though the techniques are designed to be more general, Gamut has been implemented to build software similar to board games. Though some might consider games frivolous, the kind of interactive behavior found in computer games is complex. Game-like behavior can be found in educational software as well as computer simulations. Games are also motivating in their own right. Video and computer games are an eight billion dollar industry [57] as a source of recreation. Though it is not reasonable for a thesis system to be able to compete with a commercial enterprise where a single game costs millions of dollars to produce, it is possible to show that with these techniques a nonprogrammer can create many of the salient behaviors that game applications possess. When one strips away the technical flash and intricate drawings of many computer games, the basic game behaviors can be demonstrated by a system like Gamut.

Creating a real system also imposes constraints on how many features one can program into the system. Gamut has several trade-offs and limitations that would not be present if it could be fully implemented. Thus, Gamut is a prototype system. Its goal is to present enough features to illustrate the power of the interaction and inferencing techniques without becoming too mired in implementation details.

1.2 Implementation and Testing

The most significant cost of developing a system like Gamut is building it. However, in order to make the system usable, it needs to be tested. In addition to building the system, Gamut techniques were tested in two user studies under formal conditions with nonprogrammers. The first test occurred before a single line of code was written. In a paper prototype study (see Section 7.1), Gamut's interface was constructed with a paper and cardboard mock-up. A human experimenter manipulated the paper interface to act out what Gamut would do in response to the test partici-

pant. This was used to work on Gamut's most essential interactions and to refine the *nudges* interaction technique (see Section 4.2.1) as Gamut's method for demonstrating examples. The second and final user study occurred after the system was implemented (see Section 7.2). The final study investigates how well nonprogrammers can use the actual system to demonstrate game-like behaviors.

Of course, the behaviors that a test participant could create in a short experimental session would not be as sophisticated as what an experienced developer could produce. Building games and simulations that test Gamut's capabilities occurred throughout Gamut's implementation. These games tested Gamut's ability to learn various situations that occur in full games such as how to make a board game piece follow a path. Gamut was also used to create several whole applications, including small games such as Tic-Tac-Toe and Hangman. It was also used to create parts of larger games like the monster chasing the player behavior in Pacman. The full set of applications and behaviors created with Gamut so far are listed in Appendix A.

1.3 What Is a Nonprogrammer

A nonprogrammer is usually more than just a person who cannot program a computer. People have talents that go beyond learning programming languages. Some people can draw lifelike pictures and create animated characters. Others can write music. Still others understand the world of finance and how money works. To hone talent, people must practice their skills in the appropriate setting. Learning a computer language also requires effort. In fact, the kind of programming language that a person would typically need to create a game for the computer, like C++, requires several years of training, and even then there are other factors to learn such as graphics libraries and file systems. As a result, talented people remain nonprogrammers for good reason. The effort required to learn to program could easily reduce the time a person would rather spend in other pursuits.

Although nonprogrammers often have little desire to learn to program, their talents are often desired in computer software development. The bulk of the craftsmanship that goes into a typical modern computer game consists mainly of the work of artists and not programmers [81]. Not all people want to build computer software, and it is not clear how many people would want to create computer software if only the task were made easy enough. The nearest analogy would probably be the World-Wide-Web with its explosive growth of home pages and sites devoted to almost any topic [9]. The HTML language [37] which is used to create the contents of the web is a fairly simple scripting language so it is relatively easy to learn. Furthermore, several direct-manipulation editors exist that can create HTML without programming. As a result, many average people are able to present material on the web. Unfortunately, HTML and its likely successors such as XML [8] do not support interactive content. Currently complex, interactive contents are only available on the web when one uses languages such as Java and JavaScript [28] which are much more complicated. These languages have not seen nearly the explosive growth that common HTML has seen with nonprogrammers.

Stories like the growth of the World-Wide-Web suggest that there are people who want to use their talents to produce applications. Other anecdotal evidence suggests that people would also like to produce games and interactive software. Children especially seem to want to create their own games. The essential ingredient toward using a system like Gamut is a spirit of creative inno-

vation. Children would certainly qualify as nonprogrammers if only because they have not had enough time to learn. Another group of people often mentioned as nonprogrammers are educators who might want to use a system like Gamut to create interactive lessons or simulations. Though teachers rarely have the time or resources to pick up a system like Gamut on their own, it is conceivable that some future system might make Gamut's techniques available to these people in a useful way.

Nonprogrammers are not always oblivious to programming languages. It is possible that many educated people at some point will be exposed to programming. When nonprogrammers are mentioned in this thesis, these people who have some programming exposure are not being excluded. Nonprogrammers in this thesis are people who do not use programming to earn a living. Basically, such people might be able to write a line or two of code in a scripting language but would not be able to implement any software larger than a few dozen lines. Gamut does not rely on people having any programming exposure at all. Its techniques are designed to be used without knowing any programming language, though it is assumed that the developer understands several standard computer idioms such as using a drawing editor. However, in terms of Future Work and providing better feedback (see Section 8.2), factoring in some language design might prove useful.

1.4 Gamut's Domain

Gamut's primary focus is to build games, specifically two-dimesional graphical games like board games and two-dimesional video games like the one in Figure 1.1. The goal of the research on Gamut is to improve the techniques for demonstrating interactive behavior, and games provide a high level of interactivity. Other domains such as business computing tend to produce static products. Though the process for writing a document or building a spreadsheet is interactive, the resulting product is static and could be printed out on paper if desired. On the other hand, a game is a dynamic product. The player has to be able to move the pieces and have control over the game in order to enjoy it. As a result, the developer who uses Gamut will actually have a reason to use programming-by-demonstration. Gamut is not an assistant to help users do something that they could do on their own. Instead, Gamut is a programming system used to build products that act on their own.

Thinking of Gamut's domain as board games helps to clarify what sorts of behaviors the system can be taught most easily. A board game has a background or "board" that stays mostly stationary. The board might be randomly generated or might be a fixed image as in Figure 1.1. The board represents the world in which the players play. On top of the board are various "pieces" that are graphical objects that show the player the state of the game. In a traditional board game, these pieces might be pawns that show the square where each player is located. In a video game, the pieces might be the player's alter ego and other characters which the computer controls. Each of the pieces can usually be moved independently and the graphical location of each piece is often the most salient feature of its state.

Other portions of the game's state can be found in counters and switches. For instance, some games keep counters to track a player's score or how much money the player has earned. Many games also have a large collection of hidden data. In an actual board games, hidden data is often stored in decks of cards. Cards can contain all sorts of data including numbers, text, and images.

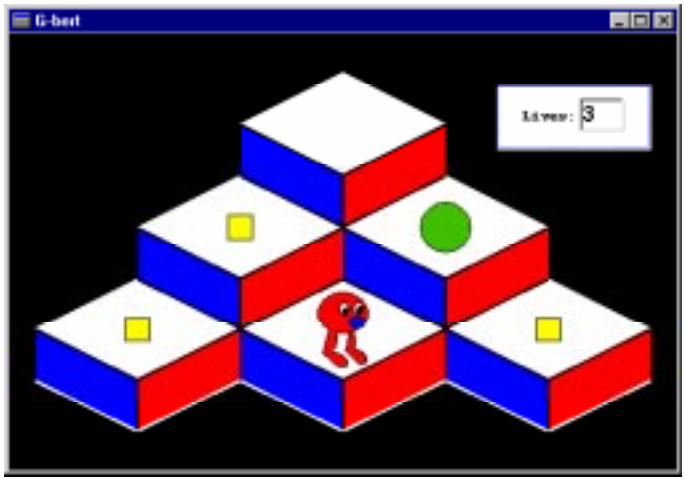


Figure 1.1: An example game created with Gamut. The board consists of a pyramid of cubes (drawn with bitmaps). The player's character jumps from cube to cube collecting markers and trying to avoid the balls falling down from above.

Cards can be played as the game progresses to change the rules of play or to cause various events to happen. Forming a set of cards into a deck increases their usefulness. Decks can have a fixed order so that several cards can appear in sequence. Decks can also be shuffled to randomize their contents. Games often use shuffling as a way to change the outcome when the game is played multiple times. Though computer games do not use literal decks, some of their behavior can be likened to shuffling and playing cards as well. For instance, to make a monster move randomly, one might imagine a deck of cards that lists all the different directions the monster may move. The computer then only has to shuffle and pick a card to determine which way the monster actually moves. (This example is worked out in detail in Section 3.3 and Section 4.3.2.4.)

Many video games use behaviors that are no more complex than the sorts of rules found in board games. The one difference is that the computer provides the ability for objects to change automatically. For instance in a Pacman game, the computer has to move the monsters so that they chase the player's character. The player's character and the monsters are pieces on the background maze which acts as the board. The player's character moves in response to the player and the monsters move in response to timers or other similar automatic events. Of course, modern video and computer games use intricate graphics and animation in order to be more enjoyable or perhaps to be more realistic looking. Though modern games certainly look better, the interaction behind the graphics is often no more complex than games with simpler graphics.

1.4.1 Why Games Are a Good Domain

Games are a good domain to test Gamut's programming techniques because it is a familiar domain to many people, it lends itself to a visual style of presentation, and it is inherently interactive. First of all, the domain exists in the real world. Computer systems dedicated for the sole use of playing games are a popular home appliance. Games are so prevalent that it seems fitting that people should be given the ability to produce game software themselves. Furthermore, games need not only be used for entertainment. Useful software such as educational titles (so called "edutainment") exist as well. Also, some kinds of simulations such as stock market and environmental models can be built using rules that are similar to those in games.

In order to build an application competently, the developer usually needs to understand the domain. Games are an easy domain to understand and many people have become experts voluntarily. People generally like games which would make them intrinsically able to enjoy the products that they build in Gamut. This can help motivate people to take care in what they are building. Since many people enjoy games, finding volunteers for experimental studies becomes easier. For instance, it is relatively easy to find college students who are nonprogrammers that know games and would enjoy building some.

Games are also a good subject for a programming-by-demonstration tool. Games are inherently visual. In a game, the data is mostly on the board. That is to say, the game's state is held in the various data-holding objects such as decks and number boxes and in how the pieces are arranged with respect to one another. This means that a game is a closed system and it does not have to query outside resources such as databases or other systems in order to determine what it should do next. Furthermore, the game's data is presented in a graphical format. The developer can visually scan the contents of the game to see that it is in an appropriate state. Keeping the game's state visible is a key element in Gamut because otherwise such data would need to be represented in some other form such as a textual language.

Finally, games are inherently interactive. Behavior occurs in the game because the player does something that causes it to happen. Understanding what the player does to cause an event to happen and what the system should do in response helps the developer understand how to program interactive behaviors in general. By including timers as a source of automatic events, the developer can also build autonomous behavior using the same style. In other domains, such as word processing, the product of the system is a static document. When the final product is static, a PBD system acts as an assistant and is not critical for achieving the developer's goals. For instance in word processing, the PBD system might try to watch the user and determine when a task becomes repetitive so that the system might finish it. However, users can always ignore such a system and perform the work on their own. With an interactive product like a game, however, the PBD system is not an assistant but the crucial means for achieving the user's goals. The user must use the PBD system or else be forced to write code in order to make the game operate.

1.4.2 What Makes Games Challenging

Games have many of the complexities of other programming domains and require a degree of sophistication that is not handled by other PBD tools. The class of games which Gamut can produce possesses complex rules which control how the graphical objects in the game react to the player and to one another.

For instance, games use complex logic and conditions. The rules in games can form complex dependencies from many different kinds of data. Games have many different modes such as the current player's turn and what part of the turn is currently being played. These modes affect the kinds of behavior that various parts of the game produce at different times. For instance in Pacman, the kind of dot the Pacman eats determines whether or not to change the monster's color to blue. Furthermore, when the monster's color changes, the monster's behavior changes as well from chasing the Pacman to running away. Modes are often represented by objects that the affected behavior does not control. In other words, objects can affect behaviors while not being affected by the behavior themselves. Inferring this sort of relationship is not handled by most other systems without resorting to written languages.

The objects to which the rules of the game refer can require complex descriptions. For instance, objects can be referred to singly and in sets. Furthermore, different properties of objects can make one object preferred over others. For instance, a player may only be allowed to manipulate pieces that are marked as belonging to him or her. Just the simple act of clearing the game board requires the system to distinguish between the players' pieces and parts of the background that should not be deleted.

The rules in games can sometimes use long description chains where the description for one object feeds into the description for the next. For instance in Monopoly, the space to which a piece moves when the player throws the dice could be described as "the space the dice's number of squares distant from the current location of the current player's piece." This one description incorporates several objects including the dice, the pieces on the board, and the mode that indicates who the current player is. Gamut has to be able to generate this complex chain of descriptions preferably in an incremental and robust manner.

Finally, games require a broader base of data types than other systems have handled. Several systems have provided basic figure and widget drawing but few offer ways to encode lists and sets of data. The system has to be able to handle data in this form as well as make the data available to other behaviors to act as modes and conditions.

1.4.3 Example Games in Gamut's Domain

Gamut can be used to create computer versions of actual board games like Parcheesi and Sorry. Games where one piece moves at a time in a fixed direction are especially easy for Gamut to infer. In theory, it could also be used to create more complicated board games like Chess and Monopoly. The reason that these games are currently out of reach is that Gamut does not have all the required heuristics implemented (see Section 1.4.4.2). Gamut has been used to create complete versions of smaller games such as Tic-Tac-Toe and Hangman, but larger games will require some more implementation.

Gamut can also create video games such as Pacman and Q*bert. These games have simpler graphics than modern video games so they were easier to draw using Gamut's editor. In fact, a Q*bert-like game was used as a task in Gamut's final user study (see Figure 1.1 and Chapter 3). Sometimes only key behaviors would be demonstrated in Gamut. For instance, the key behavior in Pacman is how the monster chases the player's character and so this is what was implemented in Gamut. The rest of the game is similar to other behaviors that were demonstrated for other games, so it is assumed that they could be demonstrated for Pacman as well, provided the system were sufficiently debugged. On the other hand, the Q*bert-like game was constructed in total. Other small game-like behaviors were also created in Gamut such as characters that move through a maze using different algorithms and a Turing machine simulation.

Gamut can be used to demonstrate the behaviors of educational games such as Reader Rabbit [48] and Playroom [38]. These games have a board-game-like appearance and use graphics similar to pawns and flash cards to provide the gameplay. The Safari usability task (see Section 7.2.2.3) uses cards in a similar way that would be used to match phonemes in Reader Rabbit; thus, Gamut can provide the essential logical component for these kinds of games. Gamut can be used to create other things as well; Appendix A lists the full set of games and behaviors that Gamut has been used to build along with pictures of some. The majority of the created games were used to debug

Gamut or show how to infer a specific behavior. Creating full-fledged applications, though, was not a priority so most of the games are small.

1.4.4 What is Outside of the Domain

Of course, Gamut cannot be used for everything, and, for that matter, it cannot be used to construct all games. Gamut has a limited repertoire of graphics and data types. Gamut is also limited by what concepts it can recognize in a demonstration. Forms of description are beyond Gamut's capability when they cannot be represented by Gamut's internal language and when the needed level of inferencing is just too great. Though Gamut's domain is restricted, it is important that it not be restricted to so-called "toy problems." Gamut's domain has not been purposely restricted to make its problems tractable. In fact, Gamut has been used to build several actual games and other behaviors that exist in modern games.

Gamut's restrictions come from several sources. First, there are basic restrictions that are just inherent in the problem. Gamut cannot read the developer's mind and it also cannot know more about the application being created than the developer knows. There are also restrictions due to the limited amount of time and effort that can be realistically given to a thesis. Thus, Gamut only represents a subset of all the concepts it would probably need to cover every possible game. Finally, there are problems that may be solvable, but are simply too difficult to handle in the scope of this thesis.

1.4.4.1 Rules Versus Strategy

Gamut is a tool that a developer can use to *build* a game. It is not a tool that learns how to *play* a game. One cannot have the system watch two people play a game of Chess and have the system learn to play like a grandmaster. This distinction is the difference between a game's rules and a game's strategy. The rules of the game dictate which moves are legal and illegal and describe what the system's responses should be to whatever the player does. However, the strategy of a game is a complex evaluation of all the various moves and combinations of moves that each player will make in order to pick the best one at the present time. A strategy implies modeling one's opponents and considering what they will do in response to your own moves.

Gamut is a programming system. A developer uses Gamut to create behaviors which when combined produce an interactive game. Gamut does not have a will of its own and does not have the capacity to reason. Essentially, Gamut can only learn what the developer shows it. All data, all state, and all behaviors must be spelled out by the developer by creating objects that represent the entire state and demonstrating all the various behaviors. Gamut will not generate state or behavior for the game that the developer does not specify.

Specifying a strategy can require a great deal of state and algorithms with which to manage that state and build conclusions from it. If the strategy is simple enough, the developer may be able to demonstrate it. For instance, a monster in Pacman basically moves toward Pacman taking care to move around walls. If the developer creates state to show which direction the monster is moving and properly points out the walls and Pacman to the system, then Gamut will be able to infer how the monster moves. However without such hints, if the developer simply moves the monster about the screen, the moves will seem disconnected and the system will not be able to infer anything. Of course, more complicated strategies require the developer to create more complicated representations for the state and to demonstrate more behaviors. A strategy for Chess would likely be too

large for a system like Gamut to handle. However, demonstrating sufficient rules so that two humans could play would be much more easily accomplished.

1.4.4.2 Omitted Graphics, Media, and Concepts

Gamut has a relatively small palette of graphical primitives that consist mostly of rectangles, circles, and common widgets like buttons. Gamut can also load in images and bitmaps from outside sources. This makes Gamut limited to two-dimensional graphics. To support three-dimensions, Gamut would need to be outfitted with new editors and primitives. Gamut also does not support some kinds of graphical transformations such as rotating, scaling, and twisting objects in various ways. Since the goal of the project was to concentrate on the demonstrational aspects of the system, there was no time to implement advanced graphical inferencing.

Gamut uses a relatively simple graphical inferencing model. Basically, it can detect exact connections between the points of two objects. For instance, it can tell if an arrow is pointing to the center of a rectangle. More advanced graphical inferencing is possible using computer vision models, but these are not implemented in Gamut. As a result, Gamut is limited in how well it can detect graphical constraints between objects. Also, Gamut does not infer velocity or acceleration or other kinds of physical phenomena. Gamut can detect if two objects are overlapping but it cannot infer that one was bounced off the other. Furthermore, Gamut has little support for smooth motion. Instead, objects are taught to jump from one state to the next as discrete events.

Gamut does not support other kinds of media besides graphics. Gamut does not have a sound library, and its ability to display animated images is limited. Essentially, developers must demonstrate complicated visual interactions manually. Though Gamut can load and display bitmapped images, it cannot display other graphical formats such as movies or presentations. Gamut's graphical and other limitations stem mostly from limitations in the base toolkit in which Gamut was developed. The Amulet system [74] which is discussed in Section 2.2.1 provided a convenient framework in which to build a large system like Gamut but it has all the limitations listed above.

1.4.4.3 Complex Behaviors and Rules

Some forms of expressions are simply very difficult to infer. For instance, mathematical expressions beyond simple arithmetic are for the most part intractable. This means that physical simulations that model complex motions such as particles, planets, and trajectories are very difficult to infer from demonstration alone. Gamut does not provide any support for mathematical expressions so they are not currently available to be incorporated into games. Gamut currently only supports a limited form of addition and subtraction. Gamut can add a positive or negative constant to a numeric widget, but that is all. This was the only numerical relationship that was needed for Gamut's experimental tasks and the other games created so far.

Other rules that Gamut cannot infer involve sets of objects. Describing singleton objects was more crucial toward making Gamut successful. As a result, sets are not as well supported. This is especially true for sorting sets of objects. For instance, Gamut cannot "pick the top three numbers" from a deck of cards though it can pick the single highest and lowest number from a set. Gamut also cannot create objects as a parameter to some other description. For instance, Gamut cannot learn to "draw a circle around all the blue rectangles." This deficiency and a proposed fix is described in Section 8.3.4.

1.5 Programming-by-Demonstration

As previously mentioned, Gamut uses programming-by-demonstration (PBD) as its method for creating programs. In PBD, the developer shows the system what to do by demonstrating examples of the desired behavior. The system watches as the developer gives examples and generalizes the examples to convert them into the application's behavior. With PBD, developers use the same interface to demonstrate behavior that they use to draw their application's interface and other components. Thus, they need not learn a secondary system or notation in order to produce behavior. It should be noted that there are other kinds of programming that use examples as a basis but do not infer behavior. These other systems are listed in the Related Work chapter in Section 2.1.

The task of combining examples into a general concept is called inductive learning [95]. The idea is that the system can detect the primary constituents of each example so it knows how each is similar and different. The goal is to interpret the examples as closely as possible to the way the developer intends the system to behave without requiring so many examples that the developer would give up. The system must therefore try to reduce the number of possible interpretations as quickly as it can while still trying to pick an interpretation that would be appropriate. In other words, the system must seem to converge quickly on the behavior that the developer intends so that the developer will not become frustrated.

There are several known hard problems in PBD for which this thesis had to find acceptable solutions. One of these problems is called the "object description problem." Consider the textual language of a command-prompt system like Unix or DOS. In order to specify an object like a file, one must type in directions that tell the computer how to find that object. For instance, the file might be in the directory two levels up and in the subdirectory called "temp." To specify a group of objects, the user uses a pattern matching language such as including the character "*" to represent any combination of characters. In a direct manipulation interface, the user does not use language to point to an object. Instead, the user picks the object with the mouse, thereby hiding the mental process that he or she used to find the appropriate object on the screen. Since PBD systems use a direct manipulation interface, it becomes the system's job to reconstruct the description of the objects the developer modifies in an example. The difficulty is that the objects the developer changes may represent a general class of objects that the behavior should affect when the programs executes. It is the system's job to describe the general class of objects. Gamut's inferencing algorithm for creating object descriptions is one of the most advanced, and though it is not a general solution to the object description problem, it is sufficient to create descriptions in its domain.

Another PBD problem that Gamut tackles is the general method the developer uses to produce examples. Past PBD systems have used awkward interfaces that forced the developer to demonstrate in a less than desirable way. Some systems required all examples for a behavior to be demonstrated at once. Others only allowed the developer to provide a single example, and the developer would have to modify the system's generated code to correct any mistakes. Gamut provides a straightforward technique called *nudges* for the developer to demonstrate examples. The nudges technique allows the developer to demonstrate a behavior incrementally allowing the developer to add new features to a behavior over time as the need arises. It resolves various conflicts and problems as new information is provided and never forces the developer to examine any code. The nudges technique is described in Section 4.2.1.

1.6 Written Programming Languages

In order to clarify what it means to program without using a written programming language, one must define the properties of these languages. When taken to the logical extreme, any form of communication can be defined as a language. In that sense, Gamut is a language because it is used by the developer to communicate application behaviors to the system. However, it is not a written language; instead, it is based on the language of direct manipulation [90]. Languages use symbols to represent the concepts being expressed. In a written language, the symbols themselves have no inherent connection to the concepts and have to be learned. Consider how the word “blue” is distinct from the actual color. Without a color chart or previous experience, one cannot guess the color of blue from the word. However, the symbols used in direct manipulation are analogous to things that exist in the physical world and are based on metaphor. The color blue in direct manipulation might be represented by some object that is actually colored blue. Similarly, in PBD to move an object, the developer actually uses the mouse and moves the object and does not write text saying “move the object.” Thus, direct manipulation interfaces and PBD reduce the semantic distinction between the symbols of the interface and the ideas that the developer wants to express.

The most obvious written programming languages in modern use are compiled languages such as Pascal and C++. In these languages, the programmer writes out textual statements, then uses a compiler to reduce the statements into machine code which the computer can execute. These languages have been designed primarily so that the compiler can produce efficient machine code in a short time and very little effort has been made to make these languages easy for programmers to write. This emphasis on the machine’s needs and not on the programmer in language design has made programming languages noticeably cryptic and difficult to learn. Unfortunately, the expressions in compiled languages have become so typical that when new languages are produced, they contain similar cryptic forms of expression. For instance, the interpreted Visual Basic language combines elements of Pascal and BASIC (which was partially derived from FORTRAN). A more recent language, Java, includes the bizarre **for**(expr; expr; expr) statement borrowed from C and C++ even though it is known to cause beginners trouble.

This sort of cryptic language design is common in the feedback that PBD systems present as well. The language that the system presents is often based on standard notations such as assignment, if-then, and for-loop statements. If developers are required to annotate or revise the code the system generates, then they are required to learn the same sorts of syntax found in compiled languages. A notable exception to this rule are languages that use icons or pictures of the actions the computer performs. These languages can be easier to understand because the developer can relate the actions seen in the pictures to the actions performed in the interface. Though these languages may be easier to understand, the developer may still have difficulty assembling new constructs in that language or adding annotations. Though the developer may understand the purpose of constructs that the system prepared, creating one’s own code may require using constructs for which the system has not provided any examples. In fact, the reason the system may need the developer to annotate the code is because it cannot generate those kinds of constructs itself. Thus, the developer would still need to learn how to use the language without aid from the system.

The syntax with which one assembles written languages is often complex and is prone to errors and misinterpretations. A compiler will usually complain about any mistakes and requires precision that a typical developer cannot provide in one attempt. A common solution is to provide syn-

tax checking editors that prevent the developer from making these sorts of errors. The more advanced syntax-checking languages provide graphical cues that tell the developer important information such as the types of parameters or how the code is nested. A good example of this sort of language is AgentSheets [84]. I call these languages "snap-together" because writing code in them has a similar feeling to assembling an object using a child's building blocks.

Though snap-together languages are easier to use than typical written languages, it is still the developer's responsibility to know what symbols the code must contain. The language cannot tell the developer what statements the code must use to generate the desired behavior. This is probably the most essential feature of a written programming language. At some point, the developer has to make a conscious decision about how the behavior of the application is represented. The goal in Gamut is to reduce the need for this kind of decision making so that the developers can concentrate entirely on what their applications do and not have to worry about how this is being represented internally.

1.7 Gamut Philosophy

When developing a large system, one tends to devise maxims and guiding principles that dictate how the various pieces of the system will work as a whole. Many prior systems have followed guidelines as well, but Gamut, in many ways, breaks away from the conventions that have guided other systems. As a result, the set of principles upon which Gamut is based need to be made clear and written down. This section discusses some of the principles that were used to design and develop Gamut. Though some of these points are controversial, they all helped shape how Gamut operates. The majority of these points concerns the developer and how the developer interacts with the system. It is hoped that future system designers will be able to use this set of criteria when they build their own systems.

1.7.1 Developers Make Mistakes

Developers are people, too, and mistakes will happen. A programming system cannot assume that the developer will always be exactly right. This is especially true in systems designed for nonprogrammers. A nonprogrammer does not always know the proper way to demonstrate and will have to experiment. When the developer programs by trial and error, the system has to be able to forgive the errors. In practice, this means that new examples ought to be treated skeptically. When the example shows a divergence from what was demonstrated previously, the developer should be made aware.

Since developers make mistakes, it is also true that past demonstrations cannot be trusted. A complicated application cannot usually be built in a single sitting. That means that the first time a behavior is demonstrated, it will not likely have the same context as when the behavior is demonstrated later. Thus, the system cannot assume that if some example were applied in the current situation that it would do exactly the same thing that occurred back when the example was first demonstrated. As a result, old examples need not be recorded since the system will never replay them because the context under which they were demonstrated may not be valid anymore.

1.7.2 Programs Should Be Built and Revised Incrementally

In general, the developer does not initially know how to build the desired application. The first few steps in building a program are typically tentative and experimental. In order to facilitate this

early state, a programming system should allow developers to test their creations as soon as possible. This helps give developers confidence that their application will eventually work correctly and it builds a framework that the developer can later augment to build the full application. Similarly, when the developer wants to refine or modify the application, it should be possible to add new changes without having to redo any of the work that was performed previously.

In order that the developer might test changes immediately, the system should always keep a functional version of the developer's work available. This means that each time the developer makes a change or otherwise modifies the application, the system should be able incorporate those changes into a working version of the application quickly. Also, when the system inserts new code, it must not ruin the code that was defined previously.

Gamut technique for demonstrating new examples is based on having the developer make small incremental revisions to the existing behaviors. The system allows the developer to revise a behavior at any time and immediately makes those changes available for testing. Gamut uses the same code that it executes as its internal representation for what it has learned about the developer's application. Thus, any changes that the developer demonstrates are immediately converted into working code. Likewise, Gamut's inferencing algorithms attempt to modify as little of the code as possible when the developer makes a change. This helps preserve the developer's previous work.

1.7.3 Do Not Ask the Developer About Code

A system designed for nonprogrammers should not be asking the developer to make decisions about the code the system generates. By definition, a nonprogrammer does not understand code; thus, it is unlikely that the person could make an informed decision regarding code. For instance, some systems present the developer with a list of possible interpretations for the last demonstrated example. The developer is asked to choose which interpretation is correct. Of course, the developer may not think that any of the interpretations are correct or may choose one randomly if the question is confusing. Though the list of interpretations may not be code on their own, the repercussions of the developer's decision will directly impact what code the system chooses. Systems that put too much faith in the developer's selection may not be able to recover when the developer makes a mistake. It seems better for the system to capture the different interpretations internally, pick one, and let later examples show which interpretation was best.

This same principle holds for systems that require the developer to augment generated code with conditional statements. The developer is not likely to know that the generated code must be modified in order to perform correctly. Furthermore, the developer would have to know which part of the generated code must be modified and what sorts of modifications are legal. Also, if the developer picks the wrong condition or modifies the wrong code, the implications on the behavior are unpredictable and possibly severe.

This is not saying that the developer should be kept oblivious of the generated code. For instance, a motivated developer may choose to learn Gamut's internal language in order to look for problems or to build an application more efficiently. A competent developer can often modify a behavior more quickly by editing the code rather than by demonstrating a new series of examples. The point is that to a nonprogrammer, code cannot be read, and the system should not rely on the developer to make decisions about code.

1.7.4 Lots of Examples

The developer uses examples to demonstrate behavior; therefore, it is necessary that examples should be quick and easy to make. The system should not require elaborate set-up procedures or unwieldy dialogs for the developer to make an example. This implies that the developer should be allowed to make an example at virtually any time and in any situation. It also implies that examples can be provided incrementally for any behavior whenever the developer decides that an example is appropriate.

If examples are made easy enough to demonstrate, then the system can assume that the developer will provide plenty of examples. As a result, the importance of any one example is diminished. This agrees with the statement in Section 1.7.1 that examples should not be recorded and replayed, but should simply be used to make immediate modifications and then forgotten.

1.7.5 The System Needs Hints

A system cannot infer more than rudimentary behavior if the developer only demonstrates the surface-level activities of the application. The system needs to have some form of hints in order to infer the connections between the behavior's various components. Machine learning is often characterized as a search process. The system searches the space of possible interpretations of the developer's examples to find an acceptable program. By using hints, the size of the space that the system must search is dramatically reduced.

Asking the developer to provide hints is reasonable because the developer should know how the intended application works. If the developer does not know how the application works, then it is unlikely that the system could make it work on its own and it would be unclear if the intended behavior was even possible to build. The system can ask the developer to provide hints whenever it finds new examples to be in conflict with the generated code. The developer should know why the behavior changes from one time to the next and should be able to tell the system what objects are involved.

1.7.6 Developers Can Draw Objects To Represent Application Semantics

The most important form of hint that the developer provides is the data that shows the entire state of the application. Any programming tool must know about all the objects that affect the operation of a given behavior. Objects that are not drawn explicitly by the developer would have to be inferred by the system. Inferring the existence of new state is very difficult. In AI terms, it is called the *hidden object problem* [93]. If a certain kind of hidden object is presumed to exist, then a system might be able to learn its characteristics. But it is extremely difficult to both determine that a hidden object is needed and discover what type of object it is [93].

Therefore, it is important that the developer be able to make an application's hidden objects explicit. In Gamut, this is performed by creating offscreen and onscreen guide objects. The guide objects show all of the application's state so that Gamut need not infer the state itself. Once again, it is reasonable to ask the developer to create guide objects because the developer should know how the application works. If the developer does not know how to represent an application's data, then it is not likely that the system will be able to do so, either.

1.7.7 No One Likes To Switch Between Run and Build Mode

In Gamut's first user study, a paper prototype model was studied to see how well developers would respond to Gamut's techniques (see Section 7.1). One of the primary discoveries is that no one likes separate Run and Build modes. In an application development system, there is a distinction between the time a developer is creating the application (Build mode) and the time the application is tested or played by the user (Run mode). These modes are common in most user interface development tools. Smith refers to this dilemma as the "Use-Mention" problem [92]. Some PBD systems have even elaborated these two modes into even more modes such as Marquise's Run, Build, Train, and Show modes [76]. These modes appear to be more of an artifice of computer development tools and not how people think about creating applications. After all, if one were creating a game with paper and cardboard, one could certainly create new parts for the game and make up new rules while it is being played.

The lack of distinction between Run and Build modes was made apparent in the paper prototype study. If the developers wanted to push a button, they just pushed it. If they wanted to move the button, they just moved it. Changing a mode switch was irritating and confusing. Thus, it became imperative that various features of Gamut should not rely on a universal Run/Build mode switch. In other words, if the developer wanted to use the mouse, he or she could do so without changing how other parts of the interface behaved. Likewise, a developer should be able to push one button in the application to make it perform its behavior while another button was selected and edited. Though it was not possible to eliminate mode switches from Gamut entirely, the system has been made so only one mode, which tells Gamut to learn an example, is really necessary.

1.7.8 Code Should Be Represented With Unordered Actions

Traditionally, computer code is treated as sequential statements to be executed in order. The effects of one statement trickles into the next creating a combined result. Unfortunately, it is difficult to automatically modify sequential code. If the wrong statement is inserted in the wrong place, it can have undesired consequences. Similarly, portions of code cannot be easily moved or copied from one context to another because the set of actions that affect a given piece of code is difficult to determine. By keeping actions unordered, the amount of state that must be tracked during a demonstration is reduced. With unordered actions, the system is only concerned with the changes that the developer makes in the interface and does not have to deal with interactions caused by the order in which the changes occur.

Dependencies within the code can be created by sharing descriptions in multiple places. For instance, say that two objects are being modified in a particular behavior. One object is being moved while the other is being recolored. If the color of object being recolored depends on the position where the other object is being moved, the description for the moved position will be shared by the other object's description for its color. The action that changes the one object's color does not have to rely on the action that moves the other but only to the description that shows where it will be placed. Thus, the causality of ordered actions is removed by sharing the sources where the actions derive their data.

1.8 Thesis Statement

When reading the prior work in the programming-by-demonstration field, one realizes that people have already claimed that their systems allow nonprogrammers to build games, educational software, and many other kinds of applications. Therefore the goal is not to be the first, but instead to do the job in a better way.

> The interaction techniques and inferencing algorithms in Gamut allow nonprogrammers create a broader range of applications than previously possible without resorting to a written programming language at any level.

With Gamut, nonprogrammers can build whole applications, not just the interface or a few behaviors. Furthermore, they can build these applications entirely using programming-by-demonstration. At no point do the developers need to modify the system's generated code. This puts a much larger burden on the system than has been previously attempted. The way that this is accomplished is that Gamut's interaction techniques allow the developer to provide the system with needed hints and information in order to resolve ambiguities. Also, Gamut's inferencing uses more sophisticated algorithms to infer more complicated behaviors automatically.

1.9 Contributions

This work significantly improves the state of the art in programming-by-demonstration. In achieving this improvement, there have been several contributions:

- The *nudges* interaction technique provides a streamlined interface for demonstrating examples. Programming is reduced to telling the computer to either "Do Something" or "Stop That."

- The "Stop That" portion of nudges makes demonstrating negative examples relatively easy. Nonprogrammers can use Stop That's negative examples without confusion.

- *Hints* are incorporated into the inferencing algorithms in a useful manner. The developer uses hints to point out crucial objects that the system would have difficulty finding on its own. Hints also serve as a means of communication when the system asks the developer to resolve ambiguities.

- The card and deck widgets are introduced to store a game's data. Cards and decks are a metaphor based on playing cards. They can be shuffled and presented sequentially and can be used to control behaviors within the game.

- The *recursive difference* algorithm is developed that allows chains of descriptive statements to be formed efficiently. This algorithm propagates changed parameters back into the generated code in order to revise a behavior.

- A method for automatically generating attributes for an inductive algorithm using hints is developed. This allows Gamut to use *decision tree learning* to automatically infer the effects of modes and conditions.

- A set of criteria or a "philosophy" is provided for creating PBD systems. The guidelines in this philosophy were used to form the basis on which Gamut was implemented and can serve to guide future projects.

- Finally, a system that integrates all of these ideas in a coherent way was implemented. This allowed Gamut to be tested with nonprogrammers in a laboratory setting. These tests demonstrated how well Gamut might perform under more stringent and realistic conditions. It showed that nonprogrammers could use the techniques fairly well and were able to perform tasks using programming-by-demonstration.

1.10 Criteria for Success

In order for this thesis to succeed it had to meet three criteria:

- *Adequacy*: It has to be possible to use the system to build entire applications using only programming-by-demonstration. In other words, it cannot require the developer ever to have to use a written programming language.

- *Novelty*: It has to be possible to create behaviors that other programming-by-demonstration systems cannot build without requiring the developer to manually edit code. This would show that its inferencing is significantly more powerful.

- *Usability*: It must be usable by nonprogrammers. This would show that others can use Gamut to build applications. In other words, Gamut needs to be subjected to a usability study involving nonprogrammers. The tasks in the usability study needed to be difficult enough that they could not be easily replicated by other programming-by-demonstration tools.

1.11 Overview of Thesis

The following chapters show how the interaction techniques and inferencing algorithms presented in this thesis allowed Gamut to meet these criteria. First, the next chapter discusses work performed by other people that inspired this project. It also contrasts others' results and ideas with those presented in this thesis. Then, Chapter 3 presents an example to give the reader a basic impression of how Gamut is used and operates. After that, the bulk of the thesis, Chapters 4, 5, and 6, describes how Gamut works and lists all of its features. Then in Chapter 7, the results of Gamut's user tests are presented and discussed. The final chapters, 8 and 9, discuss how Gamut relates to the rest of the world and gives ideas for future work that could make a system like Gamut better.

Chapter 2: Related Work

There have been many influences that have driven the research in Gamut. This includes research in programming-by-demonstration, machine learning, and application-building software. PBD is, of course, the strongest influence. Gamut is related to application builders in how it is used to create interactive software. Gamut's editors use direct manipulation techniques in the same way as interface and application builders. Gamut uses inferencing techniques found in machine learning to enhance its capabilities. The inferencing algorithms are based on several inductive learning techniques one of which, decision tree learning, is borrowed directly from the literature.

2.1 Programming-by-Demonstration Systems

The past work in programming-by-demonstration spans about 25 years of research. The first efforts such as Smith's Pygmalion [91] involved using examples to aid in writing small programs. The field has since branched out from there. Recent research includes domains such as creating charts and graphs [73], editing text documents [70][79], and creating interactive games like Gamut. The motivation for using PBD has always been to provide end-user programming, that is, PBD systems have been aimed at nonprogrammers who could possibly benefit from having the system automate some of their tasks. This section presents some of the different ways in which PBD has been applied to different problems.

2.1.1 Dimensions

PBD has been applied to problems in a variety of ways. This section discusses some of the general aspects of PBD systems. For instance, some systems use inferencing to interpret what the user is demonstrating whereas other systems use an example as a template to perform other operations. For systems that do perform inferencing, there is a question of how many examples the user may show the system for a single behavior. In general, systems that can incorporate multiple examples can learn more complicated behaviors. Another aspect is how the system provides feedback to the user and whether the system can ask the user questions. Finally, there is the role the PBD system plays in the user's application. Some systems act as an assistant to automate repetitive tasks while others are more crucial to the work of the application.

2.1.1.1 Inferencing Versus Reference Examples

One of the major distinctions in PBD systems is whether or not they perform inferencing on the user's behalf. Early systems such as Smith's Pygmalion [91] and Halbert's SmallStar [39] did not perform inferencing. In these systems, the developer created an example as an aid for writing

code. For instance in Pygmalion, the developer programmed in a graphical language which the system would run using the developer's example data. As the program reached areas that required new branches and variables, the developer would add the appropriate code. SmallStar thus worked like present day macro recorders. The developer would record a series of operations and then would edit the macro to add conditional expressions and to refine the code using the dialogs shown in Figure 2.1. Halbert introduced the term "data description" to refer to the expressions that refine the program's parameters. Gamut uses a similar code structure for its internal language (see Section 6.1) but, unlike SmallStar, does not expose it to the developer.

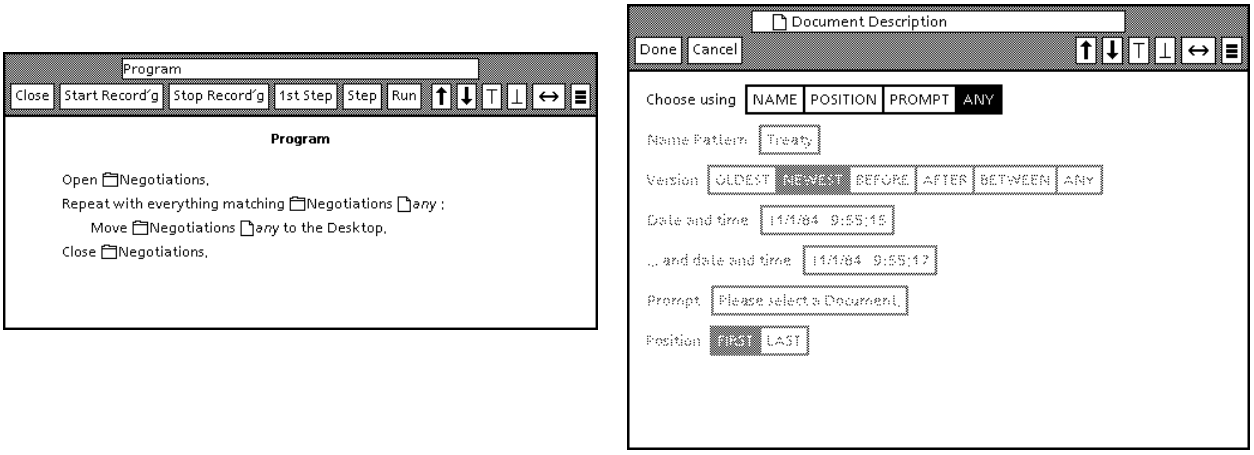


Figure 2.1: The SmallStar dialogs used to edit the developer's example [39]. In the left dialog, the developer selects portions of the code to edit. In the right dialog, the developer modifies an item in the code. (Reprinted with permission.)

Some systems used inferencing for a limited portion of code development and then used editing to further refine the inferences. For instance, Wolber's system, DEMO [96], could infer linear algebraic relationships by demonstration, but the developer had to add conditional expressions to the code manually. The Marquise system [76], which is described in detail below, made a base inference and then the developer used a dialog to revise what the system inferred. Marquise's use of an editing dialog made it similar to SmallStar except that Marquise would make an initial guess for code's data descriptions whereas SmallStar initially assumed that all parameters were constant.

There have been some systems that used inferencing exclusively. For instance, DEMO II [26] could infer graphical constraints using only demonstrated examples. Grizzly Bear [30] was another system that only used examples and it could infer linear algebraic expressions like DEMO along with simple conditional expressions. A distinguishing property of PBD systems that use inferencing exclusively to create behavior is that they all accept multiple examples. For instance, Grizzly Bear uses a "snap-shot" metaphor where developers took "before" and "after" pictures of the desired behavior using a virtual camera. The developer could take as many before and after pictures as was needed.

2.1.1.2 Kinds of Feedback

Much of the research in PBD systems has been involved with the technical details of making the system infer some kind of behavior. The need for intuitive interaction techniques and feedback was often a secondary issue. For instance, Metamouse [54] and DEMO II had very little feedback.

The developer had little way of knowing what the system has inferred. In some systems, such as Marquise and Grizzly Bear, the system showed the developer the inferred code. The purpose of the code display was to allow editing. Some PBD systems have also used feedback to ask the developer for assistance. For instance, the system might ask the developer to select among a set of possible inferences. This section discusses some of these techniques.

Cypher's Eager system [20] highlighted elements in the user interface to provide feedback. Eager was used to infer data manipulation tasks within HyperCard [4] applications. When Eager discovered a pattern in the user's actions, it would highlight the next item in the interface that it determined the user would next click on. If the developer decided that the system was selecting items correctly, he or she could push a button to have Eager finish the task that the user was performing.

Myers' Peridot system [67] used a rather direct approach to ask the developer for assistance. Peridot was the first system 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 which consisted of rules about various kinds graphical layout and widget behavior like scrolling and selecting; plus, it had the ability to detect some kinds of iteration. When the developer provided an example, the system would run through its list of rules picking out which ones applied to the situation. The rules that matched criteria that was evident in the example would be presented as questions that the developer would have to answer. Developers tended to be annoyed by the constant barrage of rule confirmations. Kurlander's Chimera system [47] improved on this technique by presenting the entire list of possible interpretations all at once as shown in Figure 2.2. The developer would check each item on the list that applied to the current situation.

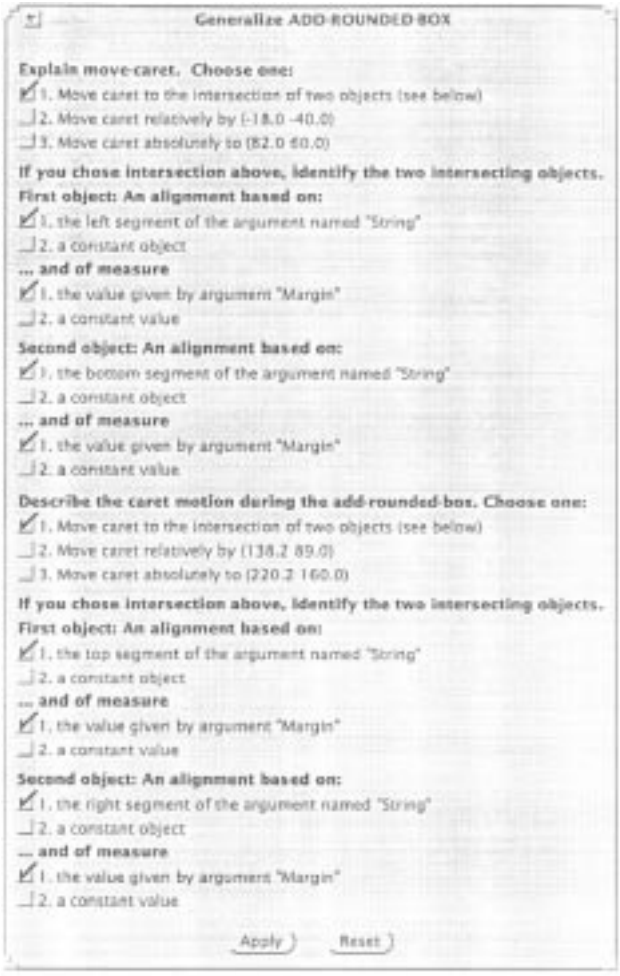


Figure 2.2: Chimera allowed users to select rule interpretations from a list. The rules that applied to the situation would be checked [47]. (Reprinted with permission.)

Chimera also used the "comic strip metaphor" as a way to display the code that the system had inferred. The language consists of before and after pictures of each transformation that the developer has demonstrated. Redundant pictures are removed as new commands are added to the end

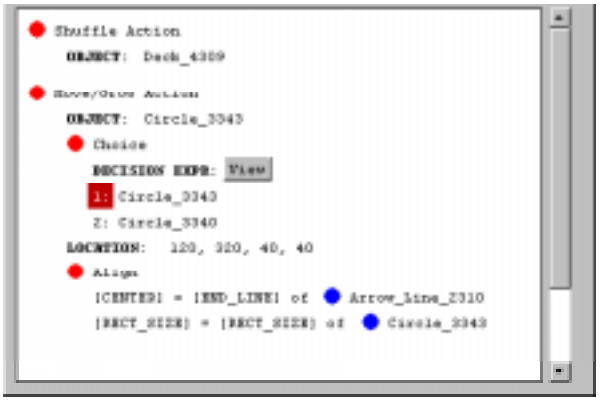


Figure 8.7: A theoretical display for Gamut's code. The display is a hierarchical index based on the behavior's actions and descriptions.

8.2.2 Editing Code and Adding New Code

When the developer views the system's code, the developer might be able to find incorrect description values. When the developer realizes that a parameter is wrong, the code display would provide a direct means for correcting the error. The system could be corrected simply by having the developer type in or select the correct value. This would also be a good way for the developer to select among alternative descriptions that the system considered. When the developer finds a poor description, he or she can select from a list of alternatives and have the system fill in the parameters as usual.

However, the developer should probably be discouraged from adding new code directly into Gamut. When faced with hand generated code, the system might become unable to revise it. The system relies on metadata that it stores in conditional statements to help it select among the various alternative descriptions that it finds (see Section 6.5.6.2 for an example). Without the metadata, the code could become a black box that the system would be unable to parse.

8.2.3 Using Standard Written Languages

Gamut uses its own representation to encode application behaviors. However, there could be some benefit if Gamut used a standard programming language, instead. For instance, if Gamut generated code in Java, it might be possible for an experienced Java programmer to modify the application that Gamut generated in order to add things that Gamut could not infer.

The problem with using a standard language is that once the developer hand modifies code that the system produces, the system is unlikely to be able to read that code again. This would make it impossible to use the system to revise or add new behavior to the code. Thus, this raises an important research issue. Is it possible to make a programming-by-demonstration system that can read and modify a standard written language like Java?

If it is not possible to read a standard language, what would be the qualities of a written language that a programming-by-demonstration system could read? The language for displaying Gamut's internal code might be an example of such a language if it is possible to engineer all the metadata

that Gamut normally uses. However, a written language that Gamut could read might still not be usable for any other programming-by-demonstration system. Currently, the algorithms used by programming-by-demonstration systems can only use data that are specifically engineered for the system. It would be compelling to show that a language intended for people to produce can also be used for programming-by-demonstration.

8.3 Improving Gamut's Inferencing

There are several improvements that could make Gamut inferencing algorithms faster or produce less code so that the application runs faster. Some would also broaden the range of behaviors Gamut could infer. None of these ideas are implemented so the extent of their impact is not known. Of the listed improvements, expanding the heuristics, relinking descriptions, and pairing generated descriptions with ones that match are relatively straightforward extensions to Gamut. On the other hand, interpreting creation as an action on objects, using the example history, and inferring higher-level properties are more involved and would likely require more research to implement.

8.3.1 Expanding the Heuristic Base

Gamut encodes a significant proportion of its heuristics in its description objects (see Section 6.1.3). These objects contain the recursive difference method for revising the description as well as the heuristics for choosing when the description should be applied. Currently, Gamut has a relatively small set of description objects implemented. The system only needed enough descriptions to be able to support the usability experiment's tasks and to build test behaviors. As a result, many types of values are not supported by Gamut.

For instance, Gamut does not support strings and textual values. It would be useful if Gamut were able to convert numbers to strings and *vice versa*. Also, providing operations that concatenate strings together or pick out portions of a string to form new values would allow behaviors to create new strings. The reason Gamut does not implement string heuristics is that they can be complicated. Text descriptions require the system to consider ordering, positions where words begin and end, and a host of other challenges that can occur in languages. Whole PBD systems like Cima [55] have been designed around the goal of interpreting and recognizing textual descriptions. However, it should be possible to implement a smaller set of text heuristics that a developer would find useful.

Gamut also needs to have its numerical heuristics expanded. In the present implementation, Gamut can only add a constant value to the value in a widget. It would be useful to have a full set of arithmetic capabilities available including multiplication and division as well as the ability to nest and combine multiple expressions. Note that Gamut would need to use hints to be able to assemble a larger equation. The developer would have to use guide objects to hold partial values so that the system does not have to infer complex expressions (which is known to be intractable [7]). It is not likely that a PBD system will be able to infer complex numeric expressions requiring differentials or higher order mathematics. For these tasks, the system would need a way for the developer to type in mathematical formulas directly. Typing in formulas would not violate Gamut's principle of not showing the developer code. After all, if the developer thinks of the formula as a written mathematical expression, the system should allow the developer to express the formula in the written mathematical domain.

Another area that Gamut should handle concerns sorting objects and determining how objects are ordered. Constraints involving order occur in several games. For instance there is the turn order, or a rule might change if one event occurs before another. Decks have the most need for order-determining descriptions. Currently, Gamut does not provide descriptions of any item from a deck besides the "top" item. Implementing ordering descriptions would also allow Gamut to support z-ordering actions. A z-order action changes the order in which objects are stacked on top of one another in a two-dimensional interface. For instance, with ordering descriptions, the developer might demonstrate "bring the lowest green object to the top of the window." Implementing ordering descriptions is not difficult, but there are a lot of possibilities to consider. Ordering must provide not only before and after tests, but it has to count things (pick the third card after the Jack), and it has to form sets (take the three largest values and delete them). Thus, implementing ordering descriptions, though useful, would not be trivial.

One final area in which Gamut could be expanded is supporting more graphical constraints. For instance, some systems like DEMO II [26] can recognize when two objects intersect and the point of intersection can be used to position other objects. Better graphical constraints would let the developer specify positions such as "somewhere inside the box" or be able to connect to points other than the corners, sides, and center. Several researchers have developed sophisticated graphical constraint algorithms and routines for recognizing them by example [33]. It should not be difficult to incorporate many of these techniques into Gamut.

8.3.2 Recomputing Descriptions to Maintain Sharing

When Gamut uses searching to find a suitable description for a parameter, that description becomes shared by multiple parameters (see Section 6.5.2.1). Allowing shared descriptions is helpful because it reduces the code size and prevents Gamut from having to learn concepts twice. However, it is possible for a description to become shared before it has been completely generalized. If the description is shared before it is completed, it is possible for Gamut to repeat work while trying to revise the shared description.

When the system recursively traces back through the descriptions' parameters using the difference methods (see Section 6.4.4), it treats the code like a tree. Shared descriptions have a different context in different parts of the code, hence it is necessary to build a separate state for each parameter that contains the shared description each time the description's difference method is called. This can cause a shared description to separate into two different descriptions, even if the two continue to return the same value. This problem is partly remedied through the search process. If the two descriptions remain the same, then once one of the descriptions is revised, it can be reused again (through searching) in the second location. However, since the search process was already performed in an previous example, it seems wasteful to have to repeat the same search again. Furthermore, the search might fail to reunite the same description with the original pair of parameters and instead return a different description that just happens to return the same value.

A better technique would be to keep track of descriptions that become shared and see what values are passed to the shared descriptions from their different locations. If the difference method of a shared description gets passed the same value in one context as it did in a prior context, then the prior context can be shared and difference method need not be re-executed. Since a difference method's context can send multiple values to the same parameter, the system would need a way to

track each of these values and see that the description will only become shared again if both contexts choose to use the same value.

This technique would not likely expand Gamut's domain. It would mostly serve as an efficiency improvement. Since more shared descriptions would be maintained, the code size would be smaller, and a smaller code size also improves other factors such as the time it takes the system to search for other shared descriptions. The main drawback is the added complexity to the changes set data structure (see Section 6.4.4.1) and the associated algorithms that use the set. If there are many shared descriptions and each context where the descriptions are used finds many different changes, the size of the changes tree might become too large because of all the markers noting the different values. The system would also need a way to track which new descriptions were derived from previous ones so that all parameters maintain the same description. This would add a whole new database to the system, but the new database is similar to others already present so it would likely not be too troublesome.

8.3.3 Redirecting Matching Descriptions Into Prior Descriptions

This next technique concerns making Gamut be able to revise a description even when its difference method fails to find a suitable parameter change. The heuristics in difference methods limit their search space to conserve time, but limiting the search space also means that Gamut can miss potential changes. For instance, suppose the system has already generated a long series of descriptions that in the end refer to the color of a blue circle in the main window. The developer shows another example, but now the desired object is a red circle that lives inside a card widget. Gamut sees in the example that the color has changed from blue to red. The object description that was used to produce the blue circle is a Get Property description. The Get Property difference method is passed the color red and asked to provide a reasonable substitute object. This particular method limits its search by only looking at objects in the same window as the object it used originally so it only looks in the main window. As a result, the Get Property's difference method fails to find the red circle (since it is in a card that uses a separate window) and the system is forced to make a general query like, "highlight the objects that the red color depends on."

So far, there is no real problem. If the developer highlights the circle in the card, the system will be able to start a new description beginning with that object. However, it would be better if the system could take the new circle and pass it along to the Get Property difference method that failed earlier in the process. Knowing that the developer wanted to use the red circle, the method can pass that along to whatever description resides in the Get Property description's object parameter and perhaps find a better way to revise the code. As it is currently implemented, Gamut would fork off a new description branch that would need to be described anew when it could have just generalized the original object description.

It is not possible to guarantee that the difference methods will always produce all possible revisions in stage two, but it is possible to use new information gained from questions in stage three to enhance the results. To do this, Gamut would look at newly generated descriptions and see whether they match in structure to one or more descriptions that they are intended to replace. If a replaced description could have been revised to create the new description, then it can be assumed that the old description should have been revised and that the difference method simply missed it. The system should use the "one difference" rule (see Section 6.4.4) in its matching criteria so that it would only match descriptions whose parameters are almost identical anyway. Instead of stor-

ing the new description at the level it was generated, the system takes the parameters of the new description and applies its values to the parameters of the old description. The system would not lose information because the new branch that would have been created for the new description is simply transferred to the parameter of the matching one. The system would also run the difference method for descriptions in that parameter to generate new items for the changes set. In the worst case, the system would add the new values as constants and the developer would be asked to create predicates to distinguish between the old and new parameter values.

This technique would make Gamut more forgiving in unusual situations such as moving objects between different owners. It could also make the code smaller since new descriptions can be incorporated into previously learned descriptions and not have to be added to the behavior.

8.3.4 Inferring Creation as an Operation on Other Objects

Gamut can already infer the creation of objects and it can infer creating a variable number of objects. However, it does not have a way to relate the creation of a set of objects with another set that exists in the application. For example, consider the behavior, "draw a box around all the blue circles," where the number of blue circles in the scene is not fixed and each circle can be moved to different locations. This behavior seems similar to a behavior like, "change the color of the blue circles to red," where the system is also affecting a set of blue circles but is performing a different action on them. The key difference is that the action that affects an object's color naturally refers to the object being affected, but the action that creates a rectangle does not refer to any other object. In a sense, the objects "being affected" in the creation situation (the blue circles) are not changed in any way. When a square is drawn around a blue circle, the blue circle remains the same as it was before. The key to solving this problem is not only forming a relationship between the created object and some other object in the application but inferring that such a relationship may exist in the first place.

The way that Gamut would form a relationship between a Create action and objects in the scene would not concern the Create action *per se*. Instead, some other action that modifies a property of the created object would be affected. For instance, in the "draw a rectangle around all the blue circles" example, the affected action is the Move/Grow that moves each created rectangle to enclose each blue circle. Thus, the relationship that Gamut must infer involves a combination of created objects and a property setting action like Move/Grow. The first time the developer provides an example for this behavior, each Move/Grow action in the example trace is separate. This is because the developer only moves one rectangle at a time. Gamut has no criteria that it could use to join the Move/Grow events together besides the fact that they are all Move/Grow events. Neither the object or value parameters of the Move/Grows are the same; thus, they are treated differently. The goal, therefore, is to find the commonality among the set of Move/Grow actions to combine them into a single operation on the set of all created objects.

The first criteria would be that there be only one Create action that creates the entire set of objects. This is not a problem since the Create action has a "count" parameter that says how many objects to create. The expression in the count parameter could be inferred to be the number of blue circles using the usual mechanisms. Once Gamut knows that the created objects belong to a single set, it could see that each of the Move/Grow actions affects a single member of the set and thus they become candidates for combination. It is likely that the value of each Move/Grow action will be set with a description that connects one of the rectangle objects to a particular blue circle in the

scene. But each Move/Grow action would have a different description that links its rectangle to a different blue circle. Pointing to different, constant objects would make their descriptions not match one another, but the descriptions might have a regular structure. In order to combine the Move/Grow actions, the descriptions of each Move/Grow would have to be the same for each, the problem now becomes one of matching the descriptions to see if they are common enough to merge into one.

It turns out that the system would not have to parse each description and compare their structures to perform the combination step. It would be sufficient to take one of the descriptions and generalize it so that it would work for the other actions. The mechanism that Gamut uses to generalize a description is revision through its difference method. Gamut would take the description from one of the Move/Grow actions and apply the location of a different created rectangle to it. The difference method would create a set of changes to indicate how it could change its parameters to comply. One of these changes would swap the constant blue circle that the first Move/Grow was using to be another blue circle used in the other location. If Gamut applies this technique to all the Move/Grows in the set, it could find the entire set of blue circles. Having the set of blue circles now lets Gamut replace the constant blue circle in the first description's parameter with a new description that picks out blue circles from the set. Thus, Gamut can convert the first Move/Grow's description into one that works for all the Move/Grow actions, thus combining them into one.

This process of revising a description to combine it with other descriptions may also be generalized and used for other purposes. For instance, a similar behavior would be "move as many rectangles as needed to overlap the blue circles." In this description, there is no Create action though the needed description process is similar. One major difference is that the number of rectangles available might be more or less than the number of blue circles. Another problem is that without the Create action, there is no obvious common ground with which to combine the various Move/Grow operations as one. This is an area where further research would be needed.

8.3.5 Using the Example History

A good area for future research would be to find ways to use the history of examples more effectively. Currently, Gamut incorporates new examples into the behavior and never processes them again. It might be possible to infer some behaviors with fewer examples if Gamut were able to revisit a previously demonstrated example and search for new data.

For instance, when the developer is demonstrating the second branch of a conditional expression, Gamut might be able to revisit the example where the developer demonstrated the first branch. By contrasting the old and new examples, Gamut might be able to generate attributes for the branch's decision tree expression. This could eliminate the need for the developer to demonstrate a third example if the objects referenced by the attributes are not in a good configuration when the second example is provided.

Searching the history of examples for new data can be dangerous. In general, the system cannot detect whether the developer intends an old example to be valid or not in a new context. The system would have to be able to distinguish information that it finds from a current example from information it finds from old examples in order that it may revoke the old information if it is later found to be invalid. Furthermore, the system would have difficulty asking the developer about

contradictions it finds when it uses old examples. Depending on how distant in the past the old example is, the developer may not even remember demonstrating it. Bringing up old examples in dialogs could be potentially confusing.

8.3.6 Inferring Higher-Level Properties of Games

Though Gamut is designed to create game applications, Gamut does not really contain much information about games. Gamut can be trained to create graphical objects, move them on the screen, and other similar behaviors. By combining these simple behaviors, the more complex behaviors of the application emerge.

However, it might be possible to encode more knowledge about games into a system like Gamut. For instance, Gamut might be made to detect when the developer has drawn a background that looks like a Monopoly-like board. The system could then automatically assume that there are pieces that move around the board and the player probably throws dice to determine how far to move. This is an example of a *higher-level inference* which infers higher-level behaviors of the developer's application. Other examples of higher-level behaviors in games are determining that a game has multiple players, determining a player's moves, and detecting that the developer has drawn a maze. Such inferencing could save the developer considerable time because the system would be able to create guide objects, game pieces, and behaviors for the game automatically. The developer would only have to demonstrate (or perhaps select from menus) the details of the application.

It would require significant research to be able to perform higher-level inferencing. For instance, it is not clear what kinds of higher-level behavior should be recognized for a given domain. It seems unlikely that a system could infer all kinds of high-level behavior for any nontrivial domain, so the system developer would have to make trade-offs. It might also be possible for a system to be trained to recognize new high-level behaviors as it is used to create more applications. However, this is entirely conjectural. It is probably possible to recognize some kinds of high-level behavior especially if the developer is allowed to help. Future research will have to determine what sorts of high-level behavior are worth recognizing and how such recognition should be performed.

8.4 Extending the Editor

Gamut only supports two dimensional graphical interfaces in a single window. Real game applications benefit from a variety of media types and modes of use. Animations and sounds make games more interesting to play and multi-player modes allow a game to take on new life as players compete, often from distant locations. The amount of new research required for these extensions is not large. Many conference papers and journal articles such as Bharat and Brown's work on extending the Visual Obliq language [5] show how it can be done.

8.4.1 Adding More Forms of Media

Gamut only supplies rudimentary graphical support. It allows the developer to create line and rectangle-like objects and it can load bitmaps like GIFs. To truly support the artistic process, Gamut would have to integrate the features found in commercial systems like MacroMedia Director [53]. Director gives the developer a sophisticated interface for drawing intricate pictures, and it also lets the developer create animations and add sound and music to the interface.

Inferring graphical constraints under the more robust graphical conditions imposed by a Director-like environment would be difficult. Gamut's representation for constraints is inadequate to handle rotation and scaling. Also, polyline and polygons would have to considered separately from other objects because they can have a variable number of points (each of which might move independently). Gamut's descriptions would have to be extended to refer to parts of graphical objects besides their standard slots. Other systems such as DEMO II [26] and Saund and Moran's PerSketch [88] have shown that more advanced constraint mechanisms are possible. These sorts of algorithms would need to be added to Gamut to support the more advanced graphics of Director.

8.4.2 Supporting Continuous Motion

In video games, objects tend to move smoothly from one point to another and not jump instantaneously. Sometimes the objects will also have an animated "walk cycle" where the objects will alternate between multiple images to appear as though the character was taking steps. However, Gamut currently only supports moving an object to its destination instantly.

Adding continuous motion or walk cycles would not be difficult. For instance, one could add a special "walking" icon that allowed the developer to draw the frames of a character's walk cycle. The icon would have multiple sets of frames so that it could appear to walk in different directions. Similarly, continuous motion could be made a property of objects. If an object were set to "move smoothly" then whenever its location was set, it would move smoothly between its original and new positions. Timing smooth motion with other behaviors is the most significant issue. The developer would need to be able to set the speed of the moving object so that other behaviors could occur in tandem with the motion. Also, it must be possible to trigger other behaviors when the motion ends.

8.4.3 Demonstrating Behaviors with Multiple or Repeated Steps

Sometimes the developer will want to create a behavior that has multiple steps. For instance, in a complex animation, an object may perform several different behaviors one after the other. Once a bullet object hits a spaceship, for example, the behavior for destroying the spaceship might first play a sound, then run an animation of the ship disintegrating, play another sound, and finally add some points to the player's score. Each of these actions must be visually or audibly apparent to the player which means that they do not occur simultaneously. In the present system, the developer must use separate timers to demonstrate each phase of such a behavior, but the developer is more likely to think of these phases as a single entity.

Some behaviors can also be expected to repeat a step multiple times. For instance, in a Monopoly game, the developer might want the player to see a piece jump one space at a time when it moves. Like a behavior with multiple steps, the developer wants each jump that the piece makes to be visible. Yet, the developer is likely to think of the piece's movement as a single behavior and not as a repeated behavior that would be caused by a timer.

It should be possible to extend the nudges technique to permit the demonstration of behaviors with multiple phases. For instance, the developer might be given the option of demonstrating an example and instead of pressing "Done" will press the "Next Step" button. This would be equivalent to pressing the Done button except that after the system asks questions, it immediately returns to Response mode so that the developer can show what happens in the next step. The system would need to be able to detect when an example is repeating all or part of a previously defined

behavior so that it could generate loops. Also, the system would automatically create timers to control how long each phase of the behavior lasted and would create loop counters to control how many times a repeated phase occurs.

8.4.4 The Third Dimension

In addition to scaling and rotating objects, there is a whole new set of challenges to solve in order to apply Gamut to three-dimensional interfaces. Though Gamut's basic nudges interaction would not have to change, the sheer complexity of inferring three dimensional graphical constraints would be difficult.

The main complexity in three dimensional interfaces is that rotation has unusual properties. Rotation has three degrees of freedom in three dimensions as opposed to one degree of freedom in two dimensions. When the developer moves an object from one point to another, it is not clear how the object transforms to reach that point. It could go in any one of several directions and orientations. The developer would need to be able to specify which degrees of freedom are allowed and might have to manually parameterize objects so that it is clear what values the system is allowed to change.

On the other hand, many of Gamut's features translate easily into three dimensions. For instance, the guide object technique is directly applicable. One can just as easily make three-dimensional onscreen guide objects as the two-dimensional versions. Furthermore, Gamut's method for describing objects is also usable. Though the methods for describing locations would need to be extended, Gamut's ability to name and distinguish one object from another would be unchanged.

Gamut's widgets such as cards and decks also can be translated to 3D. A three-dimensional card is just a space where objects can be drawn just like the two-dimensional version. One can imagine a cube to denote where the visible portion of the card is placed. Such a region could be moved in order to generate effects such as cross-sections. In other situation, the properties of the card could be changed so that the view takes on different transformations such as color filtering or exploded views. Similarly, a deck is just a collection of objects. In this case, the deck could hold and iterate among three-dimensional objects. The controls would be a button panel that hangs below the deck similar to the original.

8.4.5 Supporting Multiple Users and Windows

With the rise of networking and the Internet, there has been renewed interest in providing multiple views of the same data or environment. Often these views are presented to multiple users working on different computers in separate locations. This work is usually termed Computer Supported Collaborative Work or CSCW. In Gamut's domain, one might like to use CSCW techniques to make multi-player games where different players can interact with one another using different computers. Similarly, even in a single computer setting, an application may require multiple windows and multiple views of the same data.

Currently, Gamut only supports a single application window. Similarly, Gamut's card widget only supports a single, unmodified view of its contents. Few technical restraints prevent Gamut from supporting multiple windows and views. For instance, in order to build a two player game where both players interact in the same environment, the developer might add a second application window. The second window would be set to appear on another player's computer. If both application

windows share the same background, then both players would be able to interact in the same scene. Figure 8.8 shows an example of this kind of application. However, in the figure, the application is actually using a card for its background and the card has two viewing areas. Behavior would be demonstrated for a second window or a second viewing area in the usual way.

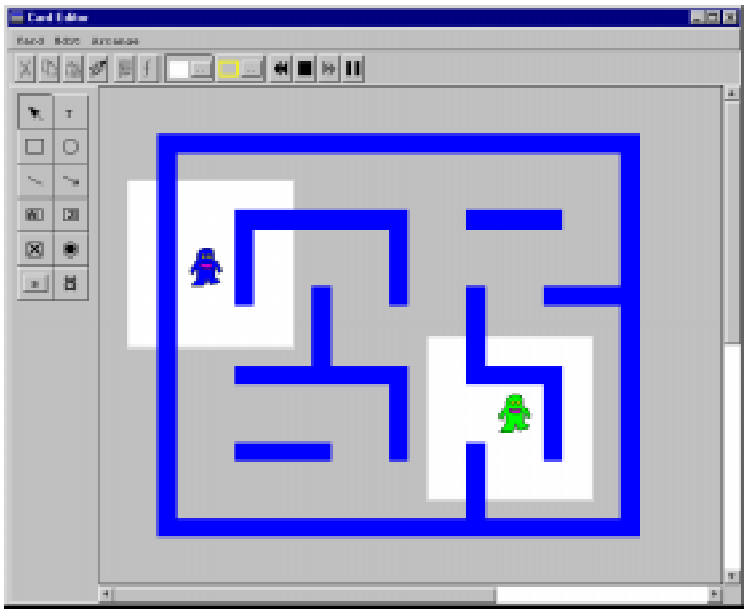


Figure 8.8: The developer uses multiple viewports in a single card to create a multiple-player game.

Of course, the real difficulty for supporting a CSCW application is the underlying network support that connects the various systems. Gamut could use these techniques to support multiple windows on a single computer without difficulty, but making a second window run on a second computer would be more difficult. The scenario envisioned above suggests using a system like X-Windows [89] to relay commands from one computer to the next. In X-Windows, the environment is able to use a network connection to drive a display on a second computer, but all the drawing commands and updates originate from a single computer. A better scenario would allow applications on two computers to synchronize and become connected dynamically. This way, users might dynamically link to applications already running and not have to start a new session each time a new player joins the group. This kind of connection would be more difficult to demonstrate in Gamut and would require further research.

Another issue is that the user may want to have different views of the data. In the scenario above, both users have identical views of the contents of the card. However, consider a board game like Chess where both players might like to see their own pieces start from the bottom of the board. Gamut might be extended to provide simple transformations using its card widget. Thus, the contents of a card might be viewed upside down or with its colors filtered or with various other transformations. Further research might also show ways to provide more powerful transformations.

8.4.6 Incorporating New Modes of Input

An area that would require more significant investigation is adding new forms of input like speech or gesture recognition to a programming-by-demonstration interface. In principle, differ-

ent input modes could provide information that is currently provided through graphics like guide objects and hint highlighting. Though the need for guide objects and hint highlighting would probably not be eliminated, it might be possible to reduce the system's reliance on these techniques.

For instance, instead of answering the system's question by highlighting, the developer might be allowed to answer verbally. The system might ask, "why did the red piece move instead of the blue piece," to which the developer replies, "because it's red's turn." From the developer's utterance, the system can determine that there is some condition in the application called a "turn" and even if the developer has not created a guide object to represent a turn, the system at least knows that it exists. Knowing that a state or condition exists would allow the system to ask the developer to point to the object that represents a player's turn using a gesture or the system might create a guide object itself and tell the developer to use that object to represent a turn.

It seems likely that in order to implement this kind of dialog, the system would need to know a great deal more domain knowledge than any current system has implemented. Furthermore, the state of the art in language understanding is not nearly at this stage yet. Should such speech technology be invented, though, programming-by-demonstration would be a good area to use it.

8.5 Using Gamut's Techniques in Other Domains

Gamut's interaction and inferencing techniques are fairly general and can be applied to other domains. Of course, the techniques emphasize demonstrating behaviors so the domains where the user wants to show a system how to do something would be the easiest to support. All of these conversions would likely require at least some research in order to properly tune Gamut's heuristics to work in the new domains.

8.5.1 Nudges in a Macro Recorder

The nudges technique can be applied to activities where the user is creating a macro. For instance, many Microsoft products use a macro recorder technique to create Visual Basic scripts. Currently, the recorder technique can only record the user's actions and cannot produce conditional logic. To revise the code, the user must single step to the desired point in the code and begin recording a new sequence. (The user may also hand edit the code in the Visual Basic editor.) By using both the Do Something and Stop That nudges, the scripts could be made to contain conditional logic automatically. The user would run the script and use Stop That to undo actions as in Gamut. The stopped actions can then be enclosed within conditional statements. Furthermore, using Do Something would allow the user to add new actions to a macro without needing to single step.

8.5.2 Descriptions Applied to Other Domains

Gamut use of nested descriptions to form complex expressions is quite powerful and could be used in other contexts. For instance, object descriptions could be used to describe the data in database transformations. By substituting some of Gamut's descriptions with ones suitable for the database's domain, Gamut could be used to describe the various elements in a database. Similarly, Gamut's descriptions could be used to describe pages on the World-Wide-Web, or values in a spreadsheet. By creating new descriptions that handle appropriate types of values, Gamut's inferencing algorithms could be applied to many different domains.

It also might be possible to use Gamut's description facilities for other Artificial Intelligence applications. For instance, tasks that use plan recognition, such as natural language understanding, might be able use Gamut's style for encoding data. Instead of structuring schemas as large units that have to recognized as a whole, the data might be structured in terms of action and description objects that can be recognized separately and assembled. This might make language understanding algorithms less brittle since they would be less dependent on the programmers having to anticipate all the different schemas that a user might produce.

8.5.3 Hints in Other Domains

Gamut's technique for providing hints could also be used in other situations. Anytime there is a dialog where the computer needs to know to which objects the user is referring, the user can use hints to mark the intended objects. For instance, in a help dialog where the user is confused about what operations can be applied to a given set of objects, the user can highlight the objects in order to ask the computer specific questions concerning them.

Highlighting is also useful as a second form of selection. For instance, the user may want to straighten out or align a set of objects. The user might highlight the object that is meant to guide the alignment of the other objects. Similarly, in a graphical editor that provides snap-dragging, the user might highlight an object to cause other objects to snap to it and permit more complicated forms of alignment. For instance, if two objects were highlighted, the system might try to form alignments with both objects at once. This kind of operation would typically require two or more steps to perform in standard editors if it is even possible.

8.6 Supporting Other Features

Besides extending Gamut's editors and media capability, one could extend Gamut's internal structures to make it more compatible with other technologies. This would allow Gamut's code to be applied in more areas. As it currently stands, Gamut is a self-enclosed environment. People can make games but cannot use Gamut's behaviors anywhere outside of the system. Also, Gamut only provides basic support for creating behaviors and could provide support for other areas of an application such as automated help. Whether these ideas involve new research depends on how much effort one is willing to spend. Compiling Gamut's code could require significant research to make it efficient, on the other hand interfacing Gamut to the internet would not be as complex.

8.6.1 Compiling Gamut's Code

Gamut's code is not especially efficient. In designing Gamut's internal language, most effort was spent making it easy to revise automatically. As a result, performance tends to suffer. The problem could be remedied by adding compilation. Gamut's code is just a computer language and as such it should be possible to compile it. There are, however, a number of complications. First, the language has many dynamic features. This makes the code similar to Self [14] or Lisp in which code is rarely static but tends to contain myriads of pointers to various kinds of objects and data. Compiling dynamic languages is more difficult than static languages such as C++ or Pascal because computer architectures are not as well suited to handling pointers. For instance, dynamic procedure calls can take more time to execute than static procedure calls. Compilers have to perform detailed analysis of the structure of a dynamic program in order to translate it into a form more similar to a static program.

Gamut's description language is not easily compiled into a more efficient form. In order to specify an object, a description often has to search through all objects in a window to find the one it needs. Commonly, hand-written code uses data structures as a shortcut so that an object can be found without search. Converting a description into an analogous data structure without affecting its operation will likely be quite difficult. However, if languages like Gamut become popular, then developing this kind of compiler technology might become more common.

Until such technology is available, though, there are still some transformations that would improve Gamut's code. For instance, it is relatively easy to remove dead code and strip out descriptions that are not being used. This would simplify the code and eliminate the conditional structures that hold the dead code. This would make the code smaller and make it run faster. On the other hand, stripping a behavior's code would eliminate all the alternative descriptions that Gamut keeps for revision purposes. Gamut may have to rebuild a large number of mistaken inferences if the developer ever decides to edit a stripped behavior later on. As a result, a behavior should not be stripped until the developer is confident that it is complete.

8.6.2 Automated Features

Since Gamut's code is automatically generated, it is more amenable to automated techniques. For instance, it is easy to search a behavior for all objects to which it refers. It is also possible to add features to the language that record various statistics. For instance, Gamut might track how often each branch of a piece of code is executed. It could then tell the user when parts of the code have not been sufficiently tested and might even be able to show how to create an example to test it. The statistics could also be used to perform usability analysis such as a GOMS model [44]. At the very least, Gamut would be able to automatically record mouse and keystroke information and be able to associate it with the behavior that is executed. Analyzing this kind of data may help a usability expert diagnose and correct user interface problems.

Systems like Interactive Systems Workbench [80] by Pangoli and Paterno incorporate techniques that generate automated help systems. The developer can ask the system to show how to perform various tasks and the system will act out the procedure onscreen. This technique might be useful in Gamut to explain behaviors and it could be used to generate help for the developer's application.

Similarly, the system might be able to automatically perform regression testing by creating a suite of test programs that is guaranteed to cover all of the currently demonstrated code. First, the system could act out a test example for a given behavior. Then, the developer would be allowed to modify the test and save it for later. After the developer has added more features to the system and fixed bugs, the test suite could be loaded to see if it still performs as it did originally.

8.6.3 Creating Widgets and Small Behaviors

At present, Gamut can only make monolithic applications. In other words, it is not possible to take a behavior demonstrated in Gamut for one application and transfer it to another application. It would be useful to be able to save selected parts of an application and be able to load them into another application. For example, perhaps the developer demonstrates a complicated behavior that involves a timer widget, some guide objects, and a monster character. The developer might select this set of objects and tell the system to save just the selected objects. Then, when the devel-

oper loads this file, the system will bring in the saved widgets but not overwrite the application that is currently being created.

Other techniques would make it possible for a demonstrated behavior to act more like a built-in widget. For instance, if the developer were to create objects and behaviors that implement a pair of dice, the developer should be allowed to install the objects as a widget in the tool palette. Furthermore, the internal guide objects that make the widget function should be kept separate from the guide objects in the application where the widget is used. A complex widget will likely have many guide objects associated with it. To make them visible each time a new instance of the widget is created could be difficult to understand and might clutter the screen.

8.6.4 Interfacing With the World

A number of programming interfaces are now available that allow applications to share data with one another and to affect each other's operations. For instance, the Web has made sharing files across the Internet simple. Also, standards such as ActiveX (once called OLE2.0) [10] and CORBA [16] allow applications running on the same machine to share data and widgets.

Implementing standards like these into Gamut would allow both player and developer alike to benefit. For instance, a number of tool developers provide libraries of widgets in ActiveX. If Gamut were to support this protocol, it could use those widgets as well and not require its own implementation. Likewise, a widget created by Gamut could be made to support the ActiveX protocol and it could be used by other applications that supports it as well.

AgentSheets [84] is a programming tool that allows a developer to store widgets and behaviors on the Web. The tool is able to convert its internal language into Java [27] so that whole applications may be stored on a website to be downloaded and played immediately using standard web browser technology. It would not be difficult to provide the same capabilities with Gamut. Gamut's code can be stored in a textual form that would be easy to store on a website. One could also write an interpreter in Java so that Gamut's original files could be invoked in standard web browsers. It should also be possible to translate Gamut directly into Java, though many of the complications mentioned in Section 8.6.1 would apply.

8.7 The Next Programming-by-Demonstration System

It is likely that other researchers will build new programming-by-demonstration systems in the future. It is important that new research does not repeat work that past systems have already performed. Most of the ideas presented in this section have been mentioned separately in the preceding sections. This section brings these ideas together to show how the research in programming-by-demonstration might proceed.

8.7.1 Domain Is Not an Issue

One of the bottlenecks to presenting new research in PBD is the notion that all new systems must have their own unique domain. The assumption is that past systems have covered their domain so completely that new results are not possible. As a result, new systems tend to use the simplest techniques, many of which have been used before, in order to cover some aspect of a domain that no system has yet tried.

The domain of a PBD system should not be considered a significant issue. Instead, new research should focus on results that are domain-independent and are more powerful than techniques used in past systems. For instance, heuristics that allow the developer to be less specific and provide fewer guide objects and hints could benefit any system. Inferencing that recognizes a broader range of types such as written text, sorted values, and arithmetic has not been handled well by any current system including Gamut, yet these types would be very useful.

8.7.2 Reduce Need for Guide Objects

The largest barrier to demonstrating programs in Gamut seems to be constructing appropriate guide objects. In both the paper prototype and final usability study, participants refrained from drawing objects unless they were forced through some measure. However, the state information that guide objects provide is essential to allow behaviors to be inferred and run once they exist. Many of the techniques described in the preceding sections such as using the example history and showing behavior icons have the side benefit that they can be used in place of some guide objects.

In future research, it may be possible to reduce the need for explicit guide objects even further. Of course, the data that guide objects produce will still be necessary, but it might be possible to generate that data through other methods. There also might be some way to make drawing guide objects easier. If it can be made more clear to developers when guide objects are needed, they might create appropriate guide objects on their own. For instance, if there were visual cues that showed what graphical constraints a behavior will use, the developer might be able to see when a guide object should be added when a constraint is missing.

8.7.3 Improve Interaction With the Developer

In general, there is still much more work needed on improving the interaction between a developer and a programming-by-demonstration system. Some of the tasks that Gamut does not support include inferring behavior with multiple and repeated phases, and providing feedback that tells the developer what the system has inferred. Similarly, adding new modes of input and output such as language understanding or three-dimensional graphics could produce new and interesting results.

Other areas of interaction that have not been explored include the distinction between creating new behaviors for an application versus creating a new application. For instance, it is relatively easy to demonstrate behaviors for a distinct application. The behaviors that the developer creates in Gamut are part of the game. However, the developer might also want to extend or modify Gamut itself. For instance, performing a long demonstration might be tedious and repetitive. Is there some way for the developer to demonstrate a behavior using only a couple of objects and then tell the system to do the same for all the other objects in the application? This would blur the distinction between a system and the product that it is used to produce. It is not clear how to produce such a system with a well-defined and understandable interface.

8.7.4 Improve Integration with Existing Systems

There has been some effort to use standard integration technology within programming-by-demonstration systems. For instance, Cypher's Eager system [20] communicated using the Macintosh's Apple Events protocol. However, attempts to integrate PBD using standard techniques has not yet been successful [49]. In general, integration techniques have not allowed external systems sufficient access to existing applications to make a general PBD system possible. A number of

new integration technologies have been introduced since the Eager system including OLE, CORBA, and the World-Wide-Web. If these technologies are equally inadequate to support PBD then there needs to be further research on more appropriate methods of integration.

Similarly, the languages and data structures that a PBD system uses to produce behavior should probably use existing technology if possible. Currently, it does not seem to be possible to use a standard written language like Java to encode behavior for a PBD system. It might be found through future research that standard languages can be used for PBD which would open up the field for producing new kinds of editors and tools for building programs in these languages.

8.8 Summary

There are numerous ways that Gamut could be extended. This chapter has only mentioned some of the more useful ones. The most pressing concern involves feedback. In order to be better able to debug Gamut applications, the developer needs to have more and better feedback than Gamut currently supplies. This might include providing a display of Gamut's internally generated code as well as interaction techniques such as snap-dragging to help the developer realize when graphical constraints are being applied.

Gamut's inferencing could also be improved. Besides adding more heuristics that would allow Gamut to infer more types of data, the inferencing could also be improved if Gamut were to track some of the links between shared parts of the code. Also, Gamut could infer more application behavior such as noticing when created objects act as a modification to other objects with the proper algorithms.

There is also a world of other media and types of applications with which Gamut might be made to use. Connecting Gamut to the World Wide Web would make a developer's work available worldwide and usable for other developers. Similarly, interfacing with protocols such as ActiveX would make Gamut's behaviors available in other applications. Simply supporting a wider range of media such as animation and sound would significantly broaden the range and quality of the games built with Gamut.

Finally, the research in Gamut suggests directions that future programming-by-demonstration practitioners might proceed. First, it seems that future research should be more concerned with basic problems of feedback, dialog, and inferencing, and should not be nearly so concerned with what domain the application covers. For instance, providing better inferencing over a broader range of types such as written text would benefit PBD in any domain. Likewise, there are opportunities to improve PBD interfaces significantly. It would be good to reduce the need for the developer to draw guide objects and to give the developer a better sense of what is being created. There are still significant problems in programming-by-demonstration that only future research can solve. By exploring these problems, PBD researchers are probing the task of programming computers at its most fundamental levels.

Chapter 9: Discussion and Conclusion

Gamut's techniques provide an efficient and understandable method for nonprogrammers to create computer software. This thesis has shown that programming-by-demonstration is a feasible method of programming and that PBD inferencing can be improved significantly by applying the right combination of hint-providing interaction techniques and Artificial Intelligence algorithms. This final chapter discusses some of the implications of the presented techniques. Gamut has many advantages over previous systems as well as some disadvantages. The chapter will compare Gamut's techniques to the prior state of the art.

9.1 Advantages and Disadvantages of the Nudges Technique

Since the nudges technique is Gamut's means for gathering examples (see Section 4.2.1), the characteristics of nudges tend to dictate how the developer relates to the system. Gamut is meant to provide a feeling of openness, that anything may be demonstrated to the system as soon as the developer considers it. Whether Gamut achieves this effect, of course, depends on the developer's personal tastes, but the characteristics listed below are meant to achieve an open feeling, encourage the developer to be informative, and help the developer feel in control.

9.1.1 Incremental Demonstration

Gamut's interface and inferencing process is designed to be completely incremental. That is, the developers control which behaviors they want to demonstrate and can demonstrate other behaviors before previous ones are complete. Developers only add new examples as they find that the system has not yet inferred what is desired. This is a departure from other systems that require all examples for a given input event to be demonstrated at once.

Requiring the developer to demonstrate all examples at once is simply unreasonable. It is very difficult for anyone to know ahead of time all the examples that will be required for any but the most trivial behaviors. Programming is sometimes a process of trial and error which is true even for written programming languages. It is not likely that the developer can make a behavior work perfectly in only one try. There will always be some small detail that the developer forgets, and if the system makes the developer start over each time, it can be tremendously frustrating. With an incremental system, the developer need not worry about details until they are needed. If a detail is forgotten, Gamut allows it to be added later by demonstrating more examples.

When the developer is forced to create multiple examples in rapid succession, it becomes difficult to configure the state so that all examples can be demonstrated. For example, Gamut's user stud-

ies suggested that developers do not always understand that they must represent an application's state (see Section 7.1.3.4 and Section 7.2.4.4). To represent state, the developers must create guide objects, but sometimes they do not realize that a guide object is needed. In an all-at-once interface, the system could not tell the developer that something is missing. It could not know that a problem exists at least until it has seen all the examples. Furthermore, even if it could pinpoint which example was causing the problem, the developer would have lost the proper context for that example (since he or she had to make others as well) and would not likely know how to respond to the system. In an incremental environment, the system can ask the developer questions as each example is demonstrated. This keeps the developer in the proper context and helps to resolve problems as they occur. However, an incremental system has the disadvantage that it can ask questions too early. For some cases, an all-at-once system can resolve a conflict by examining later examples. In an incremental system, the system would not know if more examples will ever arrive so it must ask questions right away.

Using incremental demonstration also helps the developer test the application more often. Gamut lets the developer test behaviors as soon as the first example is demonstrated. This is aided by Gamut's not having a Run/Build switch so the developer does not have to ask explicitly for permission to test the application (see Section 4.1.3.1). For instance, each time the developer makes an example for a button, the button can be pushed right away to see if it works. Thus, the developer can begin by demonstrating a behavior's overall interaction and fill in the details as necessary.

However, giving some developers too much control can cause confusion. The nudges technique does not have much structure. For a beginner seeking guidance, an incremental approach can leave the person feeling lost. Gamut does not provide a high level structure to tell the developer about all the pieces that must come together to build an application. When one of the pieces is missing, the system presents the developer with questions that he or she may not be ready to answer. In other words, the incremental approach cannot always help the developer find and fix mistakes. This is discussed further in a later section on debugging, Section 9.3.

9.1.2 Immediately Correcting Behaviors

Another desirable consequence of Gamut's incremental demonstration technique is that it decreases the impact when the system or developer inevitably makes mistakes. The nudges technique portrays demonstration as a teaching process. The computer is alternately told to Do Something or Stop That which presumes that the computer does not know any better at first. This is more acceptable than an alternate situation where the developer gives the computer a complete description of what to do. If the computer makes a mistake, it can appear as though the computer is purposely ignoring what the developer told it to do.

The process for programming Gamut becomes one of continuously testing behaviors until the system makes a mistake. As soon as the system errs, the developer immediately pushes a nudge button and gives the system a new example. Setting up the application to be in different data configurations becomes part of the testing process and no longer appears to be part of setting up a new demonstration. In other words, when an error occurs, the system will already be in the right configuration to show the error's context and the developer need not manipulate it further to generate the new example's context.

As a result, creating examples becomes quicker because the time to set up examples overlaps the time to set up testing situations. One disadvantage of the increased speed, though, is that a developer can make a mistake just as quickly as making a new example. Thus, after making a mistake, the developer might continue demonstrating several examples. As a result, when the mistake finally shows up to the developer as an undesired behavior, it is not always clear that the problem results from a mistake and is not part of the system's process of learning. It is not always possible to cover up a mistake with extra examples as mentioned in Section 9.1.5; thus, the developer is sometimes presented with a conflict that will not go away unless the developer uses Replace to fix the problem (see Section 4.2.3.2) or deletes the behavior and starts over.

9.1.3 Hints Let Developer Provide Critical Knowledge

A key contribution of Gamut is how it uses hints to create new relationships. Examples in Gamut are annotated by the developer by highlighting objects. These objects reflect the state on which a behavior depends and allows that state to control diverse actions including behaviors that do not necessarily affect the highlighted objects. In Gamut, hints are absolutely necessary to create the kinds of behavior needed to build complete game applications. Hints serve to limit the system's search space so that the number of relationships that must be considered for any given example is reduced. Gamut also uses hints as a way to resolve ambiguities. When a new example contradicts the original actions in a behavior, Gamut asks the developer to give hints to resolve the conflict (see Section 4.2.3.2). Hints are a good interaction technique because they allow the developer to reveal critical knowledge to the system without requiring the developer write code.

When the system finds a conflict, it poses a short question that only concerns the conflict. The developer need not know the exact form and position that the conflict occupies in Gamut's code. Hence the messages can be made fairly natural sounding. One thing the messages cannot do, however, is predict the answer. This can make the messages sound too high level and vague. More work is needed to tune the system's message feedback to produce fewer confusing questions.

Using hint highlighting can also have disadvantages. For instance, the developer may not know what objects to highlight, especially if the needed state has not been created. Sometimes the developer will choose to highlight an object at random rather than try to understand the issue. This is what I called "flailing" and it occurred several times during the user studies. It is not clear how to prevent flailing. It may help if Gamut provided better feedback so that the developer would better understand the consequences of his or her actions.

Another issue concerns the ambiguity of the highlighted object. Gamut does not necessarily know how the developer intends a highlighted object to be used. The system may use the object in an unintended way or it may use properties of the object the developer does not want. In general, this is not a problem because the system's description algorithm tries to use the highlighted objects in several different ways (see Section 6.5). If one of the descriptions uses the highlighted object in the wrong way, another might not and the system will eventually begin to use the correct description. It might also be helpful if the developer could refine what is highlighted. For instance, the developer might only highlight one of an object's properties using the property editor if the other properties might confuse the system.

9.1.4 Creating Negative Examples

Gamut can interpret examples as positive or negative without having to be labelled by the developer. In order to infer disjunctive clauses (Or-like logic as well as And-like), a system must receive both positive and negative examples. Producing positive examples via demonstration is relatively straightforward. Most developers understand that an example can tell the system to perform some behavior; however, the process for providing a negative example, i.e. telling the system *not* to perform a behavior, is more difficult. Thus, a system where the developer can produce negative examples easily is desirable.

In Gamut, the easiest way to demonstrate a negative example is to use Stop That. By design, the Stop That nudge is invoked in mostly the same way as Do Something. Both nudges send the system into Response mode, so the developer is not likely to consider them differently. Participants in the final user study (see Section 7.2.4.1) did not have much trouble using the Stop That nudge. Some even preferred to use Stop That over Do Something even in cases where Do Something might seem more appropriate. Using Stop That to undo an errant action seems much simpler than the techniques in other systems, such as Inference Bear [30], that require the developer to specifically note whether a new example is positive or negative. The developer is spared the trouble of learning how negative examples differ from positive and can focus on correcting the system's mistakes.

9.1.5 Problems with Sloppiness

Situations that developers found to be difficult often occurred when the developer created an example containing a mistake but does not realize it. Gamut must assume that all examples are correct, that the positions where objects are moved and values that properties have been given are exactly as they should be. Gamut does not try to "clean up" an example and adjust objects so that they line up better or modify the state in any other way. As a result, when the developer is sloppy and produces examples that do not quite match the intended behavior, Gamut will try to infer how the behavior should reproduce every imprecision that the developer demonstrates.

Instead of recognizing that the system is trying to infer the behavior of a mistake, the developer may assume that the system is operating normally and that he or she just needs to highlight the objects on which the behavior depends. Unfortunately, those highlighted objects might then be inferred by the system as ways to explain the error and not be used to explain how the behavior actually works. As a result, later when the developer demonstrates a correct example, the system will notice that the highlighted objects are not sufficient to fully explain both the correct behavior as well as the mistakes made earlier. The developer would then have to use Replace to force the system to accept the new example.

Handling sloppy examples is a difficult problem to solve for Gamut. The most obvious solution is to provide better feedback that would, hopefully, show the developer that the system is not inferring what is intended. In the usability tests, the most common mistakes were caused by misaligned objects. For example, if an object was supposed to be centered over the end point of an arrow, the developer would sometimes miss. For these kinds of errors graphical feedback like snap-dragging [6] or the feedback in Karsenty's system called Rockit [46] might be useful (see Section 8.1.4). Another common mistake is forgetting to modify some or all of the objects in the scene. For instance if two objects are supposed to alternate turns, the developer might remember to stop one object but forget to move the other. To handle this, Gamut might use a more forgiving

representation for conditions within actions or it might recap or mark the objects that are affected each time the developer makes an example.

Currently, each feedback solution mentioned is an *ad hoc* solution that would solve known problems and would not necessarily be generally helpful. It is not clear if there is a single, uniform feedback mechanism that could handle most forms of developer sloppiness in a PBD setting. At present, it seems that a general solution is not possible since different problems seem to require giving the developer different kinds of information. However, it should be possible to integrate several different feedback strategies so the most common demonstration problems are reduced.

9.2 Comparing Nudges To Other Systems' Designs

The nudges technique introduces fewer concepts than the techniques used by other demonstrational systems. Gamut requires a single mode switch whereas other systems have used a variety of modes and constructed examples using multiple phases. A number of systems use an "augmented macro recorder" design that uses at least three modes. Others require a "code revision" phase where the developer reads the code created by the system, revises the data descriptions, and augments the code with conditional and iterative expressions.

9.2.1 Augmented Macro Recorder

A standard interaction used by other systems involves an "augmented macro recorder." A picture of one augmented macro recorder dialog is shown in Figure 9.1. A macro recorder is a dialog that several commercial systems have used to let the user record short sequences of commands that can be replayed as a unit. A macro recorder is itself a metaphor based on tape recorders and use the same play, stop, and pause button style of interface. The system can replay a sequence of commands when the user selects the macro and presses play. An augmented macro recorder adds a separate "stimulus" phase. Figure 9.2 shows an image of the same macro recorder as before as the developer prepares to enter the stimulus phase. The phase is entered by pressing the button marked "Start Recording Trigger." In the stimulus phase, the recorder captures the event that causes the behavior to occur. Normally, the developer is expected to mimic the exact event that the player will produce. When the stimulus is complete, the recorder changes mode to record the response which is analogous to Gamut's Response mode.

The recorder method breaks down under a number of circumstances. For instance, when the stimulus event is a mouse drag, the developer at some point must mimic a mouse move event. However, in order to switch the recorder out of stimulus mode and into response mode, the developer would have to lift the mouse key in order to push a button in the recorder dialog. One remedy for this problem has been to incorporate a time-out in stimulus mode. The "Time Left" bar in Figure 9.2 is used for this purpose. To use the timer, the developer gets ready to record, holds the mouse in the correct configuration until the timer runs out, and then can proceed. In Gamut, the developer uses the player input icons (see Section 4.3.5) to generate the various mouse events. Thus, a drag event may be shown without needing to actually drag the mouse.

Furthermore, the recorder method makes revising code difficult. A developer will commonly find mistakes in the application's behavior while the application is running and being tested. If the system uses the macro recorder method, then the developer must recreate the stimulus event for the recorder's stimulus mode. That means that the developer must first rearrange the state of the

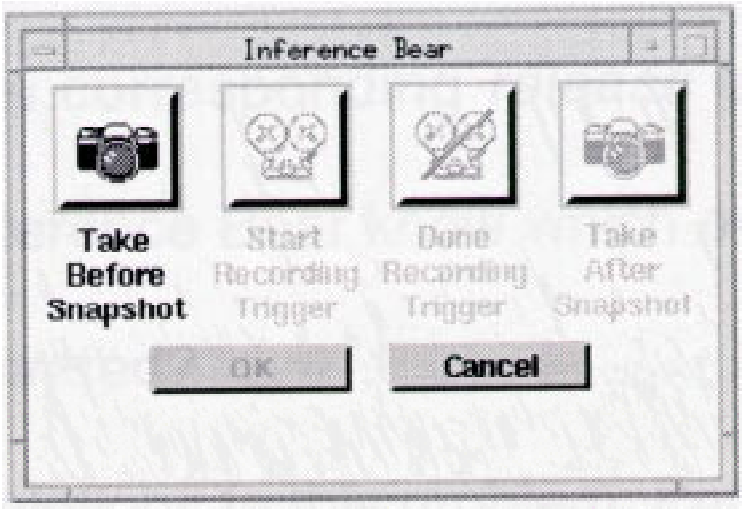


Figure 9.1: The augmented macro recorder dialog in Inference Bear [30] (reprinted with permission).

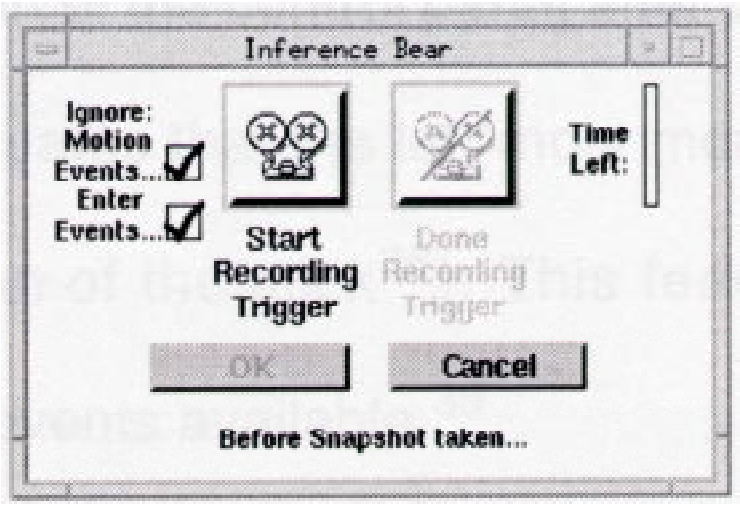


Figure 9.2: The augmented macro recorder uses an extra phase to record the stimulus event [30] (reprinted with permission). In Gamut, the stimulus is implied at the time the developer pushes a nudge button.

application to be as it was before the error occurred which implies that the developer has to know all of the state the behavior affected. By the time the developer is ready to revise the behavior, the circumstances that caused the problem to occur may have been forgotten. In Gamut, the stimulus event is implied by the last event the developer performed before entering Response mode. There is no need to set up for a stimulus event so the state of the application can be used as it is. The developer does not have to replay the event or rearrange the state of the application.

It is also difficult to generate negative examples using an augmented macro recorder. Starting the recorder implies that after stimulus mode will come response mode. But sometimes there is no response in a negative example. The developer may be confused that the recorder must be run in

order to show nothing happening. Some systems require developers to express directly that they are creating a negative example. The dialog in Figure 9.3 uses a pair of buttons to indicate whether an example is positive or negative. Other researchers have reported problems where the developer did not understand the implications of negative examples, and did not know all the circumstances where a negative example is required [30]. Gamut's Stop That interaction makes negative examples easier because the situation where the example applies is directly apparent.

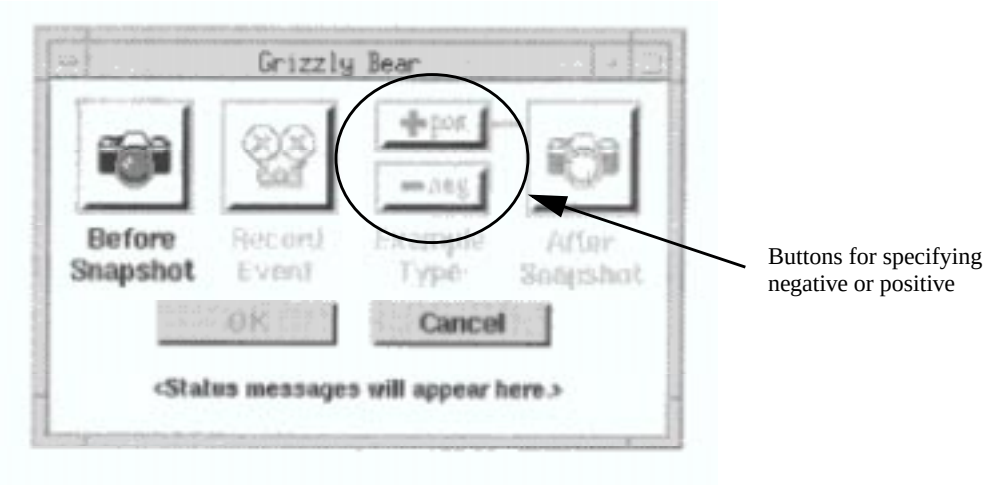


Figure 9.3: The dialog for training negative and positive examples for an expression in Inference Bear [30] (reprinted with permission).

9.2.2 Editing Generated Code

Many systems use dialogs to edit code created by the developer through demonstration (see Figure 9.4). In fact, these dialogs often provide the only way to add conditional modes and player input to a behavior. The developer is still required to create the data objects that the modes use to determine their state, but the systems are not able to infer how to incorporate that state into application. Providing a display for the application code is actually reasonable. A good display may help the developer see mistakes and may even help the developer learn to program, but requiring a nonprogrammer to edit code to build application behavior is not necessary.

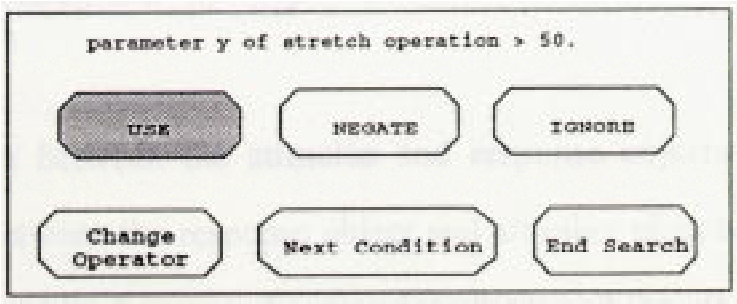


Figure 9.4: Code editor used in DEMO [96] to add conditional statements (reprinted with permission).

9.3 Debugging

Gamut is a demonstrational programming language. As with any programming language, a developer can produce bugs that must then be sought out and eliminated. In general, since the inferencing system assumes more programming responsibility than with written languages, finding and fixing bugs can potentially be more difficult. Since the developer is assumed not to know how to program, he or she cannot be relied on to search for bugs in the code. Instead, the system must allow the developer to fix bugs using the same mechanisms as were used to build the application in the first place.

9.3.1 Producing Bugs in Gamut

The bugs that are produced in Gamut are mostly the more severe semantic errors programmers make when they do not understand part of their application. PBD effectively eliminates many written language problems such as syntax errors and type mismatches, but producing bugs in an application's design and operation are still prevalent. High level, semantic bugs are some of the most difficult to correct. Fixing a semantic bug requires the developer to understand fully what the application should do at any given moment. It also requires the developer to be vigilant and watch all the effects of each behavior.

Only the developer really knows what the application should do, so the system must rely on the developer's examples to be correct. The system cannot distinguish good examples from mistakes. Also, the system will always produce a program that runs (assuming Gamut itself has no bugs). Thus, the system will always interpret examples to produce some functional program. The system has no basis to tell the developer that examples are inconsistent or make no sense because the system does not know what the developer wants to build.

Furthermore, nothing guarantees that the developer will ever catch a subtle error. An application with a large amount of internal state can be difficult to manage. The developer may need to watch many parts of the system at once. Furthermore, Gamut does not encourage the developer to structure data in any particular way. Developers can make an application's data as hard to read as they like. Missing a crucial change in the data can lead to the interface and the data becoming out of sync leading to the creation of bad examples.

When the developer does not know why the system is failing, a common strategy is to try to solve the problem by providing more examples. Basically, the developer watches the system's behavior and gives an example each time it does the wrong thing. This strategy can sometimes work. Basically, Gamut can learn to put the mistaken behavior into a branch of the code that never gets executed. The conditions on which the dead branch depends will require the application to enter a state that never occurs. For instance, the Pacman character is always yellow. A problem branch can become buried if ever its condition calls for the Pacman to become some color other than yellow. Pacman's color never changes, so the buried code is relatively safe. Of course, this kind of debugging produces "time-bombs" where if ever the application is modified slightly from the original, then an unexpected behavior can suddenly occur. Inferencing does not provide any obvious way to detect or remove problems caused in this way.

9.3.2 Techniques Which Improve Debugging

Gamut is not completely defenseless against bugs. Its techniques usually allow the developer to debug behaviors without too much hassle. If the developer actually knows that he or she caused a mistake, then repairing the mistake is relatively easy.

First, all of the developer's application is represented with visible objects. Gamut cannot produce application state on its own; thus, the developer must use guide objects. All guide objects have a visual representation so they can be inspected quickly. Each widget's most salient data is presented directly on the screen and the developer does not have to open dialogs to see it. Even card widgets have a window so that the developer can put interesting data in the place where it is most visible. The least visible object in Gamut is the deck widget. The developer cannot see the cards that are below the one being viewed except by using the deck editor. It might be worthwhile to experiment with other designs for decks to make their data more visible.

Having visible data can improve the developer's chances of detecting a problem. Being able to see how the data changes is the key feature of debuggers in most other languages. When the developer sees a problem, the nudges technique usually helps make correcting the problem easy. The developer can immediately create a new example using the context the application has already provided and instantly revise the behavior. So, although Gamut cannot force the developer to see a problem, it can be used to correct the problem quickly.

The time controls (see Section 4.4.2.1) also help the developer debug applications. Sometimes the developer will not notice a problem until long after its original cause has transpired. The time control allows the developer to go back to the point where the problem first occurred. This technique was advocated by Lieberman in his ZStep 94 system [51]. The developer can also demonstrate a new example after using the time controls. If the developer takes several steps back in the application's history and uses a nudge, the system will react as though the developer had just performed the event at that point in the history. The time controls also help the developer debug the application by making it easier to test a behavior repeatedly. For instance, the developer may demonstrate a new example, immediately use the time control to move back one step, and then try the same event again to see if it works. This kind of repetition can help the developer become confident in the demonstrated behavior and to fix bugs more quickly should any appear.

Gamut currently does not have any mechanism that forces the developers to test their applications. If the developer chooses not to look for bugs, the system has no way to know that bugs exist. The developer would still need to follow a regimen of testing in Gamut that is needed for other forms of software production. In other words, the developer would still need to apply a structured software engineering process to be confident that any delivered product met its criteria. This would become a greater concern if Gamut's techniques are ever applied to domains where correct software is more critical such as creating transportation systems or hospital equipment. It is possible that such processes might be built into the tool but this area has not been explored in this thesis.

9.4 Comparing Gamut's Inferencing To Other Systems

Gamut combines multiple inferencing algorithms to form a system that is stronger than any one of its component algorithms. This section discusses some of the issues in Gamut's inferencing algo-

rithms and relates Gamut's inferencing to that in previous systems. Gamut incorporates innovations in all three stages of its inferencing. In the first parsing stage, Gamut converts the developer's actions into a trace. The trace's simpler form allows the system to manage actions more readily while still allowing the system to represent creation and deletion. In the second stage, Gamut's propagation of values allows the system to form and revise long description chains. This allows Gamut's object description to be more advanced than others and lets it infer behavior that others cannot. And in the final stage, Gamut uses a decision tree learning algorithm based on ID3 to infer conditions automatically.

9.4.1 Contrasting Trace-Based Versus History or Snapshot-Based Inferencing

Gamut's method for reducing the demonstrated example into code differs from other PBD systems. Frank [31] classifies PBD inferencing algorithms into two fields: "snap-shot" and "history-based" inferencing. In snap-shot based inferencing, one only considers the before and after pictures of an example and ignores the process through which the before picture is transformed into the after picture. Basically, the algorithm looks at the picture as though it were a list of properties. The before state shows the values of the properties before the picture changed, and the after state shows the changed values. The goal of the inferencing is to create a procedure that transforms the set of properties into the after state. In history-based inferencing, each event that occurs between the old and new picture is recorded and is considered part of the program. The inferencing process treats the event list as macro and tries to infer the how the events in the macro are parameterized. The events themselves usually cannot be modified by the system. If the developer used a "move-line-start-point" event to move a line, then the system must always use the same event in its behavior. Gamut uses a combination of both these approaches which I call "trace-based" inferencing. A program trace is defined to be the minimum set of actions required to transform the previous state of the application into the desired state, making it similar to the snap-shot based approach. However, the form of the trace is a series of actions similar to the events found in the history-based approach.

Snap-shot based inferencing is useful because it simplifies the format of the result that the behavior produces. Since the algorithm only considers the final state of the interface, any procedure that generates that state is valid. The modified variables of the state and the variables on which they depend form a table-like structure that can be mapped using algebraic techniques. However, snap-shot based systems have difficulty when objects are created or destroyed. A created or deleted object changes the number of variables in the application. This breaks the table abstraction and forces the system to use a different kind of inferencing algorithm. Having to handle created and deleted objects requires the system to form descriptions that parameterize its algebraic tables. Such systems do not tend to have good object description facilities.

Event based inferencing is nearly the opposite of snap-shot based inferencing. Events are usually complicated structures that affect many facets of the interface at once. As a result, the order in which events are processed is very important. Also, events often perform several different operations as a unit. For instance, the group event will create a group object, place the set of grouped objects into the group's structure, and move the grouped objects' positions to be relative to the group. This combination of effects makes it difficult to anticipate when one event will override or modify the effects from a previous event. However, events that create and destroy objects are easy to represent in history-based inferencing. The create and destroy events are simply recorded in the macro as any other event would be.

Gamut's trace-based inferencing combines the advantages of both snap-shot and event based inferencing. Gamut uses events like history-based systems but transforms them into a simplified set of actions similar to a snap-shot based system. An action is like a simplified event that only performs a single operation. The effects of an action are designed not to overlap so that the system does not need to consider the order in which the actions are invoked. Keeping actions unordered is further maintained by Gamut using "anticipatory descriptions" that guarantee that any state needed by an expression is generated by the descriptions that the expression uses and not by other actions. Gamut's trace-based actions are stored in a list-like structure and the inferencing algorithm attempts learn expressions for the action's parameters. This event-like structure overcomes the problem snap-shot systems have with creation and deletion since Gamut can support create and delete actions. Gamut will infer descriptions for each action's parameters similar to a history-based system but since Gamut need not track dependencies between the actions, each description can be learned independently which helps to simply the inferencing heuristics. Altogether, Gamut's trace-based inferencing seems to be a superior method for learning in Gamut's domain.

9.4.2 The Benefits of Unordered Actions

A behavior's actions are ordered when there are implicit dependencies between them. For instance, if there are two Move/Grow actions in a behavior and the position of one Move/Grow depends on the result of the other, then these actions are ordered. Gamut's trace-based inferencing eliminates the dependencies between actions making the actions unordered which simplifies how Gamut's inferencing algorithms work.

9.4.2.1 Rearranging Actions

The most obvious benefit of unordered actions is that Gamut can rearrange actions arbitrarily when it adds and moves code. If Gamut allowed dependencies between the actions, then there could be restrictions on where code could be placed. For instance, placing a new action between two existing actions with a dependency could affect the value of the dependency and modify the results of the existing actions. Likewise, actions might have to be moved as a unit so that their dependencies are not broken. Finally, Gamut can deactivate actions by putting them into a Choice description. If Gamut deactivates an action that generates a result for other actions, but those other actions remain active, then Gamut will need some other way to generate that result without actually invoking the deactivated action. Since Gamut's actions do not have dependencies, this is not an issue.

As an example of how Gamut might deactivate an action on which other actions depend consider the behavior in Figure 9.5. In the behavior, the developer trains Gamut to create a copy of the arrow and place its starting point in the center of the circle. The developer also trains the circle to move to the end point of the arrow. With ordered actions, the circle's position obviously depends on the line and since the line must be created, its Create action must occur first. Assume that this behavior is functioning correctly. In a new example, the developer causes the behavior to occur as usual. Gamut would create an arrow and move the circle as it should. Then, the developer selects the arrow and presses Stop That as shown in Figure 9.6. This would cause Gamut not to create the arrow. In other words, Gamut would put the Create action for the arrow into a new Choice description that would act like an if-then statement. However, the developer has left the circle alone. The circle has still been moved to follow where the arrow would have been had it been created. How should Gamut describe the position of the circle? Since the arrow is no longer being

created, its position is no longer available. Still, the developer wants the circle to move, and apparently expects it to move in the same way as before. Gamut would need to have some way to derive the position where the arrow would be placed without actually creating the arrow. This is what anticipatory descriptions do. Therefore, having ordered actions does not eliminate the need for anticipatory descriptions or an equivalent abstraction.

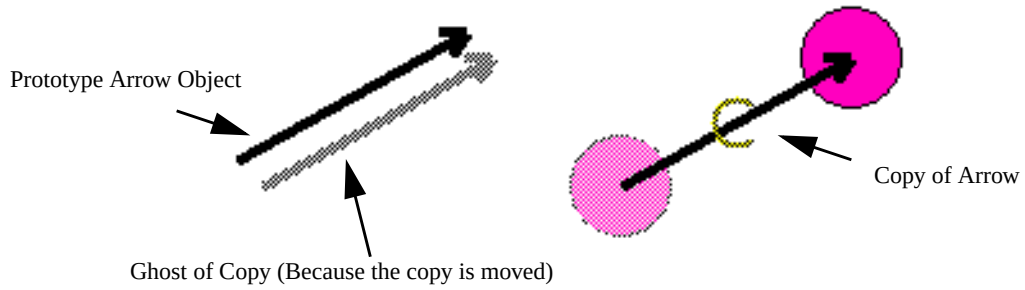


Figure 9.5: The developer trains a behavior that creates an arrow and moves a circle to its end point. The position of the circle thus depends on the arrow.

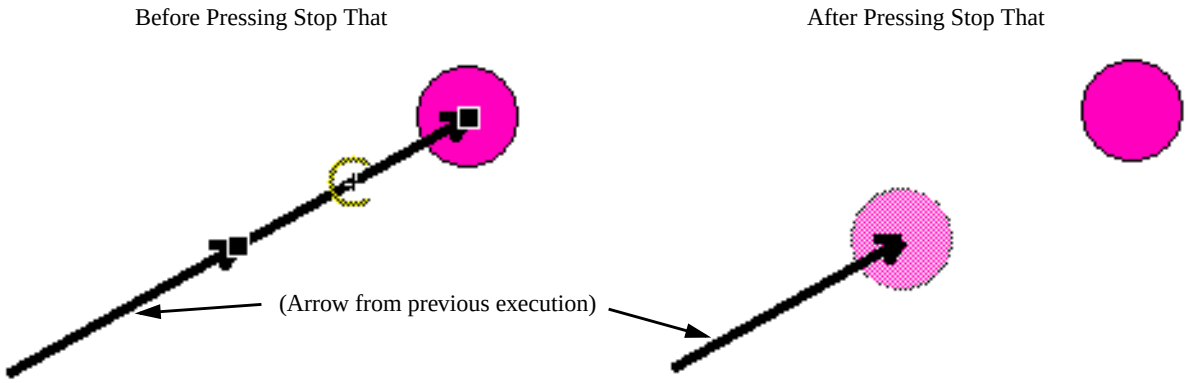


Figure 9.6: In a surprise demonstration, the developer tells the system to stop creating the arrow but continue to move the circle. The system then has to deal with the dependency some other way.

9.4.2.2 Assigning Behavior Affects to Actions

When Gamut removes the dependencies between actions, it guarantees that each action is independent. This means for each modification that occurs in a behavior there is exactly one action that causes it. Thus, when Gamut matches the actions in an example trace to actions in a behavior, it only needs to search for one action.

Similarly, when Gamut propagates a change to an action's parameters, it only needs to be concerned with the descriptions in that action. If the action were dependent on another, then Gamut might have to switch to the other action and try to revise it in order to achieve the proper affect. For instance, consider what would happen if instead of using Stop That in the example from Figure 9.5, the developer moved the circle to a different position as shown in Figure 9.7. Since the circle's position depends on the arrow, Gamut has to consider revising the position of the arrow, but since the developer did not modify the arrow, Gamut cannot revise it. When actions are kept independent, the decision whether to revise a parameter will not depend on whether another action is revised.

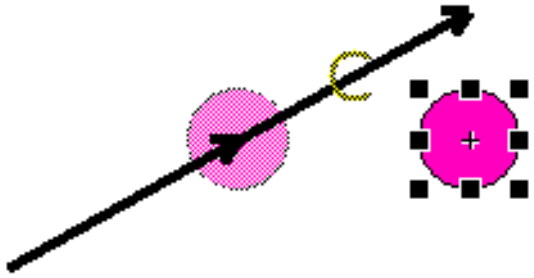


Figure 9.7: Instead of using Stop That, the developer changes where the circle moves. The system must consider the actions associated with the arrow because the circle's action originally depended on the arrow.

9.4.2.3 Eliminating Unneeded Code

Keeping actions independent allows Gamut to remove unneeded or redundant actions from a behavior. For instance, if the developer moves a single object twice during a demonstration, Gamut only needs to create a single Move/Grow action for it. The previous, intermediate move can always be eliminated since it provides no visible effect in the behavior and no other action will depend on its result.

Eliminating redundant code can make a behavior more efficient. Consider again the example in Figure 9.5. This time, the developer modifies the behavior by pressing Do Something and deleting the arrow as shown in Figure 9.8. A system that uses ordered events would learn to add a Delete action to the end of the behavior. Thus, the behavior would first create an arrow, use the arrow to position the circle, and then delete the arrow. On the other hand, Gamut would treat deleting the arrow the same as if the developer had used Stop That to remove it. This would eliminate the Create action from the behavior (by putting it in a Choice description). The player never sees the arrow be created so it does not have to be. Furthermore, creating and deleting objects tends to be more expensive to compute than simply changing an object's properties. Gamut will perform this optimization automatically whereas a system that uses ordered actions would have to use other methods to eliminate this inefficiency.

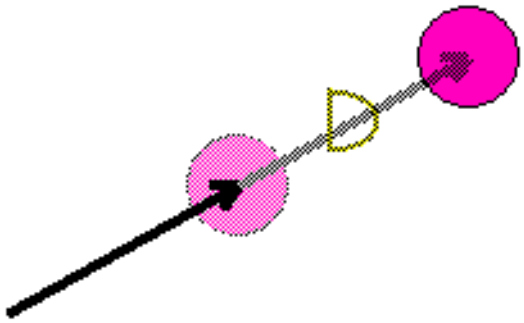


Figure 9.8: This time the developer selects the arrow and deletes it. The system learns to create an object, use its position, then delete it. However, since the arrow is never visible, it never needs to be created at all.

9.4.2.4 Anticipatory Descriptions Are Not Complicated

When the result of one action is used to describe part of another action in Gamut, the system encodes that result using descriptions. These descriptions are termed "anticipatory" because they anticipate the results of other actions. An anticipatory description is not different from other descriptions. Being an anticipatory description simply means that the system performed a special search in order to find it. It functions and is revised in exactly the same manner as descriptions created through other means.

There are two special descriptions used to represent created objects and randomized lists, the Created Objects and Objects From Action descriptions. These descriptions are not complicated, either. These simply allow information to be shared in multiple locations. For instance, if a set of created objects needs to be counted in one place and have their color changed in another, the Created Objects description will refer to the proper set of objects in both contexts. The only difference in Gamut from standard practice is that these descriptions function independently of actions. If the set of created objects has not yet been created when a Created Objects description is evaluated, the description will create the objects. Similarly, an Objects From Action description will randomize a set of objects if necessary. The Create and Shuffle actions refer to the same data as the descriptions so they can be prevented from performing an operation twice.

9.4.3 Object Descriptions

Inferring object descriptions is a well-known problem in programming-by-demonstration. A behavior as simple as telling an object to follow a path is not possible in other systems that lack a good object description facility. It is not that path following is so hard to implement. For instance, in other systems that do not use inferencing, such as Cocoa [22], path following is relatively easy to implement (the developer would define four rules that would move an object in the four cardinal directions). Path following is a good example, though, because so few other PBD systems that do perform inferencing can handle it.

For instance, DEMO II [26] and Pavlov [97] (see Section 2.1.2.2) provide only constant object descriptions. This means that the system cannot describe how an object relates to a path element. The current path element that the object should follow changes depending on the position of the object. It also means that the systems cannot refer to created or deleted objects because these are also variable. In Pavlov, the developer can create a table of actions that can be replayed. If the developer creates a series of movement actions that travel around a circuit, the system can replay the actions to make it seem like the object is following the path. However, this kind of path following is not very powerful. If the path were a complicated network or had the ability to change, a fixed list of movement commands could not traverse it.

Grizzly Bear [30] has more powerful object description facilities than DEMO II and Pavlov. Basically, it allows the developer to assign new properties to objects and can use those properties to pick groups of objects. Though this is useful, it cannot be used to implement path following. When an object follows a path, it must select the current path element to follow. Only one of the path elements must be selected at a time so giving all path elements a special property will not work. One also cannot give a special property to only the needed path element because, in order to do so, the modified element would still need to be described. Grizzly Bear's descriptions are analogous to a single link of a Gamut description. The system can describe a single level of indirection but cannot chain descriptions together to form more complex descriptions like Gamut.

On the other hand, Cima [55] has a relatively advanced object description facility. It can pick out portions of text from a document. A path following behavior in the text domain might be moving a given piece of text to the location referred to by a cross-reference. Unfortunately, Cima cannot perform path following either (or a textual domain equivalent) because it does not support actions that can modify the document. I should note, however, that among the systems listed, Cima is probably the system that would be most easily modified to support path following. In other words, it seems to have all the elements needed to perform path following and only needs to able to use it.

9.4.4 Automatically Inferring Conditions

Being able to form conditional behaviors by demonstration is one of Gamut's strongest features. Unlike other systems, Gamut can form relationships with disparate objects and use them to make behaviors perform widely different kinds of actions (see Section 5.5). For example, Gamut can use an checkbox to define a modal relationship with a monster behavior. When the checkbox changes, the monster can be made to alternatively chase or run away from the player's character. The monster behavior need not affect the checkbox.

Defining the term "conditional behavior" is somewhat difficult. For instance, a behavior that moves a rectangle a constant distance to the left could be considered conditional. After all, its destination position is different when the rectangle starts in a different location. The kind of conditions being discussed here are more general. Gamut can infer conditions that cause behaviors to perform multiple, different activities based on objects that need not be affected by that activity. In a programming language, these kinds of expressions are usually formed with an if-then-else statements where the values that determine the predicate in the if portion of the statement may or may not be affected by the operations in the "then" or "else" parts.

Many systems have relied on the developer to add if-clauses (which are often called "guards") to the system's generated code. For instance in DEMO [96], the developer uses the dialog in Figure 9.4 to add guards. Pavlov [97] uses a similar design, but instead of adding conditions by selecting from a list, the developer announces to the system that a condition is being demonstrated and then manipulates the object that affects the behavior. Of course, this ties conditions and events together strongly. If the object that affects the condition is not affected by the behavior, then it would not be possible to demonstrate its role. Consider a behavior that depends on the value of a number box. When the player pushes a button, the game should perform one behavior if the number box equals four but should do something else otherwise. Pavlov would require the developer to change the value of the number box in order to show that it affects the behavior, but since the behavior is initiated by a button and not the number box, this would not be possible.

Like Gamut, Grizzly Bear and DEMO II can both add guards to their expressions automatically but they are restricted to inferring relatively simple conditional expressions. DEMO II can only test relationships between constant objects. For instance, DEMO II can infer whether two specific line objects intersect, but it could not determine whether the objects that two arrow lines point to have the same color. Grizzly Bear can examine a constant property of a single object or alternatively all objects in the application to see if they equal a constant value. This can be used to make simple modes and palettes if the developer is willing to add special properties to those objects involved with selecting a mode. Furthermore, both these systems can only add guards to the topmost level of their code. This is akin to being only allowed to write if statements in the main body block in a programming language and not in procedures or nested code. In principle, it would still

be possible to write complete programs in this language but it would not be possible to write conditions near the part of the code where the conditions actually occur. Any code that would normally occur before the condition would have to be repeated in each branch of the condition (see Figure 9.9). Gamut infers conditions that are embedded into descriptions as well as at the top-level actions of a behavior. This means that code need not be repeated and that the conditions will only affect the code that is actually conditional.

```
if (test1) then
  Move Rectangle to Position1
  if (test2) then
    Move Circle to Position2
  end if
else
  Move Object to Position3
end if
```

```
if (test1 and test2) then
  Move Rectangle to Position1
  Move Circle to Position2
end if
if (test1 and !test2) then
  Move Rectangle to Position1
end if
if (!test1) then
  Move Object to Position3
end if
```

Figure 9.9: When a language only allows conditions to be added to the top level of code, some code may have to be repeated and the tested expressions can become more complicated. On the left, the code nests a condition within one branch of another to be near the expression it affects. On the right, the condition is moved to the top level requiring code to be copied.

9.4.5 Gamut Is Turing Complete

One of Gamut's contributions is that it is the first purely demonstrational language to be Turing complete. Turing completeness is a mathematical concept that says whether or not a machine can be used to compute various expressions [32]. Theoretically a Turing complete language (and the machine on which it is implemented) is computationally equivalent to any kind of computer and can be used to compute any computable expression. Written languages are almost always Turing complete. Similarly, graphical programming languages such as Fabrik [43] have been Turing complete as well. On the other hand, PBD systems have always required that developers modify code at some level in order to be Turing complete.

To prove that Gamut is Turing complete, one only has to show that Gamut can build Turing machine simulations. The Turing machine depicted in Figure 9.10 uses lines and circles to represent a finite state machine. It uses a row of squares to represent the writable tape. The machine above performs the simple act of reversing the colors of the tape's squares from red to blue and *vice versa*. Gamut can build this simulation because it can be used to make objects follow paths conditionally. The motion of the tape head is conditionally based on the state of the finite state machine and the state machine is conditionally based on the position of the tape head.

9.4.6 Replacing the Decision Tree Algorithm

On the whole, the decision tree algorithm has worked very well. So far, the technique has not encountered insurmountable difficulties that would indicate that Gamut should use a different algorithm. In fact, the decision tree algorithm has worked well enough that very little thought has been put into whether or not it can be replaced. Theoretically, any inductive AI algorithm can be switched with the decision tree algorithm and put into Gamut instead. The table of examples used

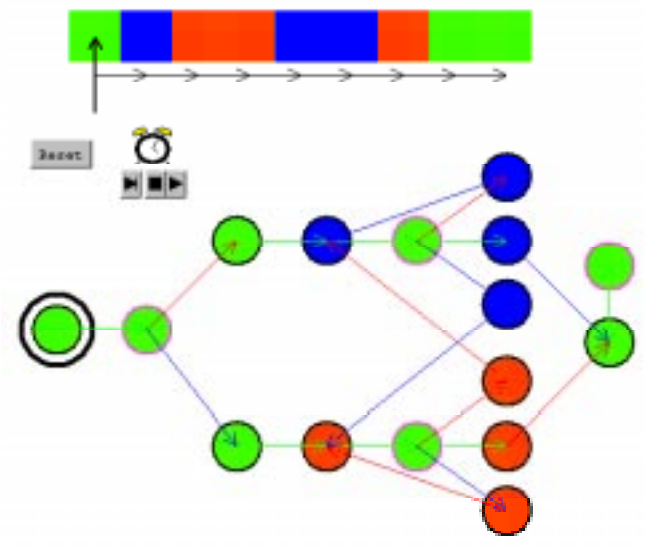


Figure 9.10: This Turing machine has been created completely using only demonstration. It contains a finite state machine at the bottom that controls the tape along the top of the display. The effect of the program is to reverse the colors of the tape from green to red and back again.

by decision tree is standard. Induction algorithm from neural networks [63] to FOIL [83] use the same data.

The advantage of using other algorithms might be faster learning curves for certain situations. Certainly some algorithms would be much slower than decision trees in Gamut's domain. For instance, neural networks [63] would probably be a poor choice. A neural network learns a set of examples by balancing the values in a network of nodes. Gamut would have to adjust the size of the network as well as generate attributes for the algorithm. Also, a neural network's learning curve tends to be slow and more suited to large amounts of noisy data. Gamut's provides only a few examples which are essentially noise-free. Another possible replacement would be Mitchell's concept-space algorithm [65] though it would probably not work well. Although a concept-space can learn very quickly, its internal structure cannot represent the full range of logical operations. For instance, concept-space cannot learn an exclusive-OR relationship. On the other hand, algorithms like FOIL [83] that learn disjunctive clauses for first-order logic statements would likely substitute well in place of decision trees. Cima used an algorithm called PRISM [13] that learns clauses similar to FOIL.

The main differences between algorithms like FOIL, PRISM, and decision tree learning (ID3) is that they use different heuristics for dividing the set of examples. ID3 uses a statistical value called information gain (see Section 6.6.1) to select features whereas PRISM chooses the feature that maximizes the probability that an example belongs to its assigned class. Hence with PRISM, the system can assume the developer made a mistake and assign an example to a different target class if it chooses to do so (whether the example is really a mistake or not). Gamut cannot do this and relies on the developer to use Replace to fix mistakes. FOIL works by picking the smallest set of features that when combined with logical-And includes the largest number of positive examples and excludes the largest set of negative examples. If not all the examples are covered it adds another clause (combined with logical-Or) to cover more examples until all examples are covered.

Like ID3, FOIL cannot exclude examples, but its statistical evaluation is more similar to PRISM. If one were to replace ID3 with FOIL in Gamut, the system would likely behave very similarly as it does now.

9.4.7 Using Other AI Algorithms

Gamut uses a collection of several different inferencing algorithms to accomplish its tasks. Some of these tasks might be improved using different algorithms. However, it is unlikely that all of Gamut's inferencing might someday be replaced with a single algorithm. In the past, attempts to use a single algorithm such as DEMO's algebraic tables have only allowed such systems to infer limited behaviors. Systems that have combined several approaches, such as Grizzley Bear, have tended to be more powerful.

Gamut contains two major search algorithms besides the one that uses decision trees. One is the process that selects descriptions based on objects highlighted as hints. The other is the recursive difference algorithm that searches backward through a behavior to enumerate the various ways it might be revised. The first algorithm has no name because it is a generic heuristic search. Gamut encodes these searches as monolithic procedures that run a battery of tests on the list of highlighted objects. However, it would be possible to recode this search as a "planning algorithm" as defined in AI [40]. A planning algorithm attempts to find a goal state given a set of initial conditions and a set of operators that transform a state from one to another. The value that Gamut wants to achieve would be the goal state and the set of highlighted objects would be the initial conditions. The operators in this algorithm would be Gamut's various description expressions that transform objects and values from one value into another. The planning algorithm could then try to find a way to convert the objects into the desired value. The STRIPS domain is one of the common tasks that planning algorithms attempt to solve [25]. STRIPS is a planning domain that uses first order logic to encode information such as the connections between rooms in a building. Gamut's descriptions could probably be converted to use STRIPS-like rules. This way current planning algorithms might be applied. The advantage of using AI planning algorithms is that they can learn to perform faster as they see more examples of the domain. Explanation-Based Learning [23], for example, keeps records of how critical planning situations were solved in order to solve future problems better. The disadvantage of using planning algorithms is that searches in Gamut's domain are normally directed by the types of the desired values are not as complicated as in domains where these algorithms are normally applied. Using a full-fledged planning algorithm may be more work than is required.

Gamut's recursive difference algorithm also uses search heuristics and may also benefit from using other AI algorithms. For instance, a critical issue is choosing which description in a set of several should be selected for revision. Gamut maintains a list of possible descriptions for parameters about which it is uncertain. The unused descriptions are stored as "dead code" so that they will not be executed when the behavior is run (see Section 6.5.6). Knowing which description to revise is similar to "belief revision" in AI [94]. A belief revision system ranks and prioritizes a set of logical statements to determine which statements the system "believes." Instead of logical statements, a belief revision system might also be applied to rank and determine which descriptions Gamut believes are true. This would require the system to maintain more information about how each description was generated. For instance, the system might have to store the originating examples from which each description was created. This way when a new example is presented, the system can revise its beliefs, that is, it can review the connections between the original

descriptions and their examples with the current example to see which is stronger. This could result in better learning overall but would also probably require significantly more memory to achieve.

One final area in which other AI techniques might be beneficial is in determining what question to ask the developer when Gamut finds an inconsistency. Currently, Gamut uses a stock textual form which depends on the description where the inconsistency occurred. Typically, Gamut asks questions when the developer forgets to highlight objects. It might be possible to generate better questions if the system had a better model of the kinds of mistakes developers make. This would require the system to probe the existing parts of the behavior to see if it can determine what sort of task it might be performing. Then, it could ask the question in a way that is relevant to the behavior's task. This might be accomplished using a cognitive model. Often, cognitive models are probabilistic judgements of what the developer is doing. These are represented using structures such as a Baysian network [15]. More involved cognitive models have been used to determine mistakes that a student makes such as ACT [2] and ACT-R [3]. These might also help the developer diagnose problems and make the system appear to be more responsive and intelligent.

9.5 Readability of Gamut's Generated Code

Gamut's internal language has more in common with data structures than written languages (see Section 6.1). The developer is not expected to read the behaviors that Gamut produces nor modify the code directly. Not surprisingly then, Gamut's language is quite unreadable and is never displayed to the developer. (A code display does exist but it is used to debug Gamut.) Several factors contribute to this language's difficulty to read. First, the code can contain several alternative descriptions for a single value. Though only one alternative is ever active at a time, the code still holds onto alternates in case they are needed. Hand-written code will only contain instructions that the developer expects to execute.

Another factor that hurts readability is that the system has to be able to manipulate the code quickly. The extra syntax and structure cues that make code readable often hurt machine readability. The system must be able to quickly scan the code to determine how it compares to a new example and to reuse code that already performs desired computations. Searching code is easier when the language has a regular structure. Gamut's code is much like a parse-tree in which the syntax of the language is already compiled away.

Finally, often the correct code for a behavior will not reflect the way a developer would articulate the behavior's actions. For instance, in the game of Tic-Tac-Toe, the computer draws a line over the three matching objects when a player wins. The developer might expect the description for the line's location to be "draw a line over guide object line that intersects three circle objects," but this ignores the fact that one of the circle objects was just created (and did not exist when the line needs to be drawn). Gamut's actual code reads more like "draw a line over the guide object line that intersects the object that was just created by the mouse click and intersects two objects that have the same color as the object just created." Note that the number three does not appear in the actual description. The actual description, though long-winded, captures the simultaneity of drawing both the circle and the line at once which the simpler sentence does not. It is similarly difficult to convert Gamut's decision trees (see Section 6.6) into English descriptions. A decision tree cre-

ates a tree data structure that would probably be too confusing for a nonprogrammer. A substantial amount of transformation would be required to describe the effects of a decision tree.

In Section 8.2, a potential method for displaying Gamut's code is discussed. It tries to eliminate the readability problem by only showing a portion of a behavior's code at any one time. The design has not been implemented, however.

9.6 Meeting The Criteria

Gamut needed to accomplish three criteria to meet its goals. It had to be possible to build applications without using a written programming language. It had to be able to build more complicated behaviors than previous systems. And, it had to be usable by nonprogrammers. Gamut has fared well in all three of these goals.

No Written Programming Language: Gamut can be programmed completely by demonstration. The developer is never shown or asked to modify code. When the system asks developers questions, it refers to objects in the application and provides high-level descriptions of what the objects are doing. The developer does not need to make decisions concerning code or need to annotate the code to make complex behaviors.

To accomplish this, Gamut uses an efficient demonstration technique called nudges that allows the developer to demonstrate examples with a minimum of buttons and mode switches. Hints and guide objects allow the developer to tell the system all the information it needs without writing the information out in a language. Gamut also provides new widgets such as cards and decks that allow the developer to represent complex data that would otherwise be difficult to do.

Building More Complicated Behaviors: Gamut can infer descriptions where values can depend on descriptions of other values and objects. These descriptions can form arbitrarily long chains. Furthermore, Gamut can infer conditions within behaviors automatically. The objects on which a behavior depends need not be affected by that behavior. Other systems can perform some operations similar to Gamut, but none are as comprehensive. Furthermore, few have as sophisticated an object description model. (The models of Cima [55] and Grizzly Bear [30] are similar though their inferencing is not as advanced.)

Gamut allows the developer to use guide objects to specify relationships that would be too difficult to infer. It also lets the developer give the system hints to reduce the amount of searching the system must perform. Gamut uses the card and deck widgets that allow the developer to represent complex data and to provide randomness. With these techniques, Gamut has been used to build many complex behaviors such as full Tic-Tac-Toe and Hangman games as well as significant parts of other games such as Pacman (the monster behavior) and Chess (moving and taking pieces). Several other games have been made such as G-bert which is a reduced version of Q-bert, Safari which is a matching game, moving an object in a maze in several different ways, and building a Turing machine simulation.

Usable By Nonprogrammers: Gamut has been tested in a formal setting with nonprogrammer students (and staff) from Carnegie Mellon University. Several of these participants were able to learn the system and complete the two test tasks with only three hours to work. One of

the participants was asked to build a third task that required her to use more of the system's interface techniques. She was able to use the new techniques without much difficulty.

Though most tests were successful, one participant did not do well at all, and all participants had some difficulty at various points. This shows that the techniques in Gamut can still be improved and that building applications can still be a difficult task even when the tools give as much assistance as Gamut does. Still, the experiment showed that nonprogrammers can use Gamut to demonstrate behaviors.

9.7 Final Thoughts

Gamut has been a successful project and has produced many interesting results. The result that I most hope to achieve, though, is to show that programming-by-demonstration is a feasible and useful approach for making software. This project has certainly shown that PBD allows nonprogrammers to create graphical applications. All of Gamut's techniques have been geared toward practical use beyond this thesis. Some of Gamut's specific techniques could find new applications such as the nudges technique which could be used to record macros as well as to demonstrate examples for many inferencing systems. The card and deck widget could fit into other systems without difficulty. The manner in which Gamut's inferencing uses hints could be emulated by other systems to improve their accuracy and speed. Finally, integrating multiple inferencing algorithms and interaction techniques enables a system to address a broader range of applications than a system that uses a single technique. Using multiple approaches allows one to divide a large problem and solve it using techniques that are best suited to each individual part.

Appendix A: List of Created Programs

This appendix lists several of the programs that were created with Gamut for testing purposes. Here, the focus is on the applications that were used for important purposes or that had more complicated behaviors.

A.1 G-bert

G-bert (see Figure A.1) is a simplified version of the arcade game, Q*bert. The player controls a character that jumps on the tops of a pyramid of cubes. A ball object will periodically fall from the top and randomly bounce down the pyramid as well. The player's task is to collect all the square markers on each of the cubes. If the player's character is struck by the ball, the player loses a life and when the player loses three lives, the game is over. If the player collects all the markers, the game is won.

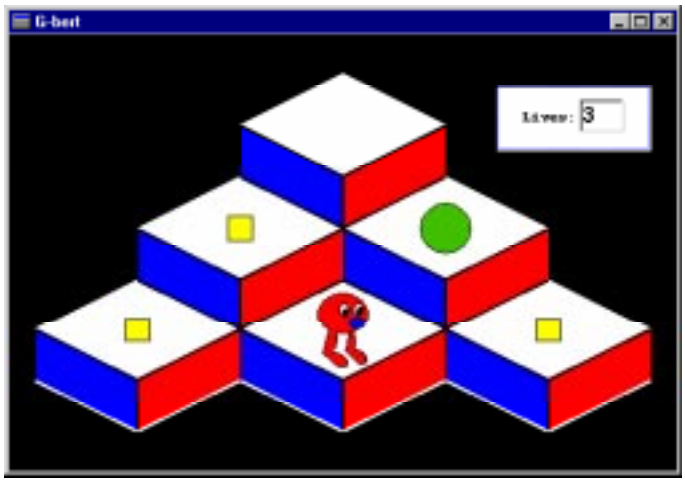


Figure A.1: The G-bert game consists of a pyramid of cubes (drawn with bitmaps). The player's character jumps from cube to cube collecting markers and trying to avoid the balls falling down from above.

A.2 UFO Shooter

UFO Shooter (see Figure A.2) is a simple game where the player tries to shoot down a UFO object. The player's ship on the left shoots bullets that travel across the screen. The UFO object

moves randomly but is not allowed to move off the screen. If the bullet hits the UFO, the UFO disappears for a while and then will reappear in the center of the screen. Currently, the UFO does not shoot back. This application was created to demonstrate that Gamut can be used to build shooting behaviors and as an application to show in a video.

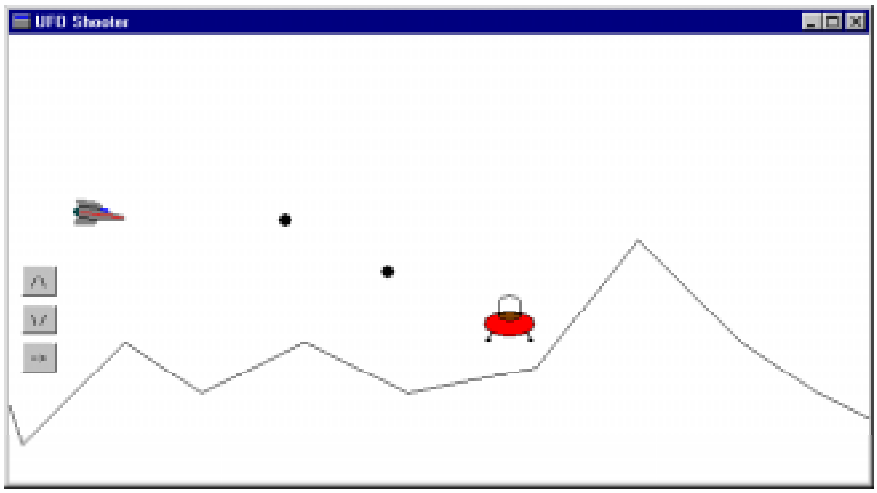


Figure A.2: The game, UFO, shooter, is a simple shooting game similar to Space Invaders. The player's ship on the left shoots at a randomly moving UFO object. The buttons on the left move the ship up and down and shoot the bullets.

A.3 Pacman

The Pacman application (see Figure A.3) is a simplified version of the arcade game, Pacman. The application possesses a single monster than can either chase or run away from the Pacman character. The player controls Pacman's direction using the arrow keys. The Pacman eats the dots on the screen as it passes over them. If it eats a gold colored dot, the monster turns blue and runs away. After a few seconds, the monster will turn back to red and start chasing Pacman again.

A.4 Safari

The Safari game (see Figure A.4) was created to be a simple application for the participants to build in the final usability experiment. It is a matching game designed to be similar to certain educational games. The system asks the player a question by randomly shuffling two decks of cards. The player then responds by pushing the Yes or No button and the system shows whether the answer is correct.

A.5 Pawn Race

The Pawn Race game (see Figure A.5) was also created as one of the final usability experiments. It was based on the game of the same name that was performed in the paper prototype study. The participant had to teach the system to move two circle objects the number of spaces on the die.

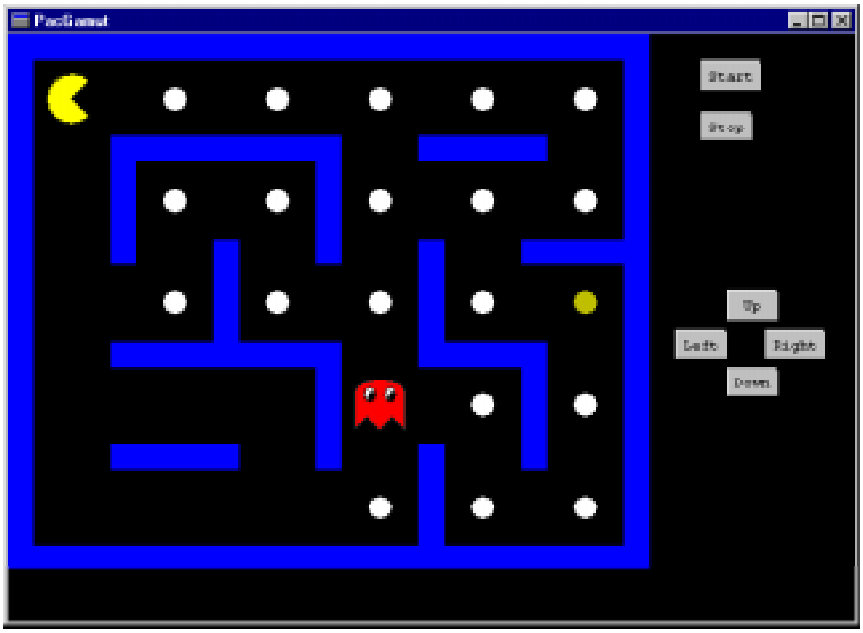


Figure A.3: In this Pacman game, the monster chases the Pacman object, while the Pacman tries to "eat" all the dots on the board. The player controls which direction Pacman faces. When Pacman eats a darker colored dot, the monster runs away from Pacman.

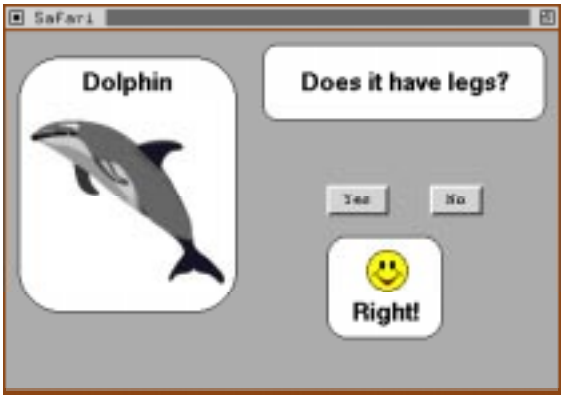


Figure A.4: Safari is a simple educational game. The computer asks questions about animals to which the player must answer yes or no.

A.6 Water Drop

The Water Drop game (see Figure A.6) was used to demonstrate how Gamut is used in a video. It is essentially identical to the ball behavior in G-bert (see Section A.1). Instead of using a deck of cards to determine the object's direction, the application used player input. The goal was to send the drop into the bucket at the bottom of the pyramid.

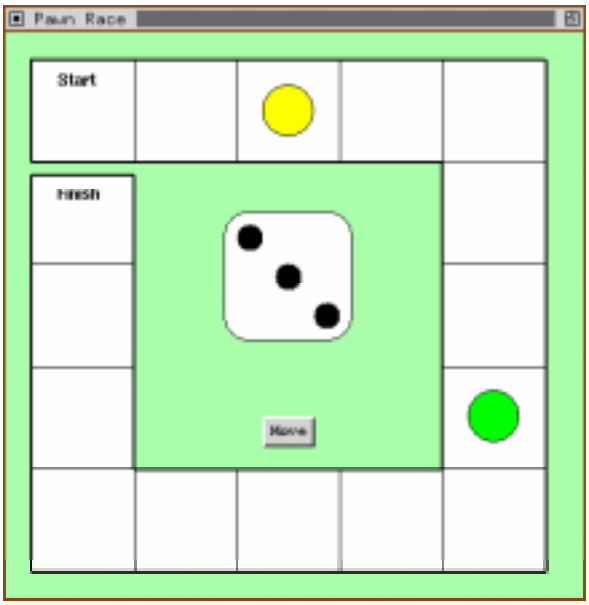


Figure A.5: Pawn Race is a Parcheesi-like board game where two players' pawns race each other to see who reaches the end first. When one player's piece lands on the other's, the landed on piece moves back to the start.

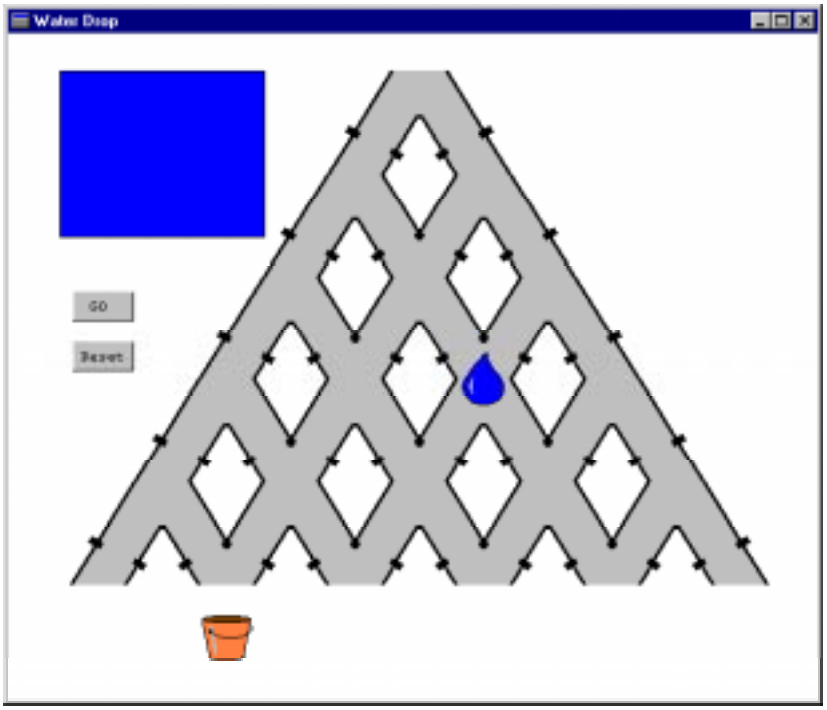


Figure A.6: In the Water Drop game, the player controls which direction the water droplet falls along a series of pipes. The goal is to get the droplet into the bucket at the bottom. The player chooses the droplet's direction by clicking on the blue rectangle at the top left.

A.7 Hangman

Hangman is a simple application to build in Gamut (see Figure A.7). The images of the gallows are stored in a deck that is made to switch to the next image when the player guesses wrong. The player's guesses are noted by having the player click on the letters along the bottom. When the clicked on letter matches a letter in the puzzle, the corresponding letters in the puzzle are colored black.

Figure A.7: In this Hangman game, the computer selects a word and the player must guess which letters are in the word.

A.8 Tic-Tac-Toe

The Tic-Tac-Toe application (see Figure A.8) turned out to be more complex than was first realized. This application was used to develop a number of inferencing improvements. Determining the winning condition required Gamut to be able to count objects that fulfill an complicated set of properties. It also required Gamut to recognize that lines can be drawn in either direction and still look the same.

A.9 Draw

While preparing for a video, my advisor was asking about how Gamut handles dragging events. The question arose whether Gamut could be used to make a drawing program. As a quick demonstration, I creating the drawing program shown in Figure A.9. The player draws in the window by dragging the mouse. The player can also change the color of the pen by clicking on the palette and can clear the screen by pushing the "Clear" button. The program was produced in less than five minutes and served as an example for the video.

A.10 Turing Machine

In order to demonstrate that Gamut is Turing complete, a Turing machine emulation was constructed (see Figure A.10). The application has two parts, the upper portion is the "tape" that

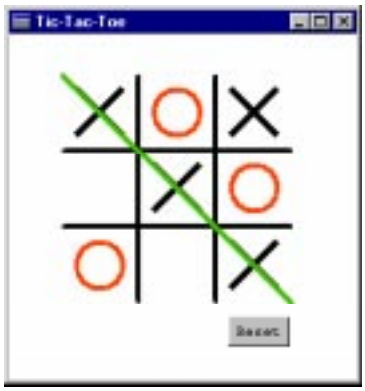


Figure A.8: This is Tic-Tac-Toe. Players alternate turns marking X or O in the grid of squares. The first player to get three in a row wins.

Figure A.9: In this drawing program, the player draws by dragging the mouse. The player can change the color of the line by clicking on the palette.

records the state of the Turing machine's progress. The lower portion is a finite state machine that determines how the tape is modified. The color of the circles' fill style in the finite state machine determine what color to put in the tape. The circles' line style determines whether the tape's "head" moves forward or back. Finally, the color of the box to which the tape head points determines which way the finite state machine's marker travels.

A.11 Left-Hand Maze Follower

This application was a simple test of Gamut's inferencing capabilities. The left hand maze follower is a circle that follows the left wall of a maze. The application was created using two methods. The first version (not shown) involved a deck of arrow lines. When the movement arrow crossed a wall, the deck would select the next arrow, and when the movement arrow did not cross a wall, the deck would select the previous arrow and the circle would move. When decks became opaque, this version was not easy to build. So, a new version was constructed (see Figure A.11) that used guide arrow lines to determine which way the movement arrow would go.

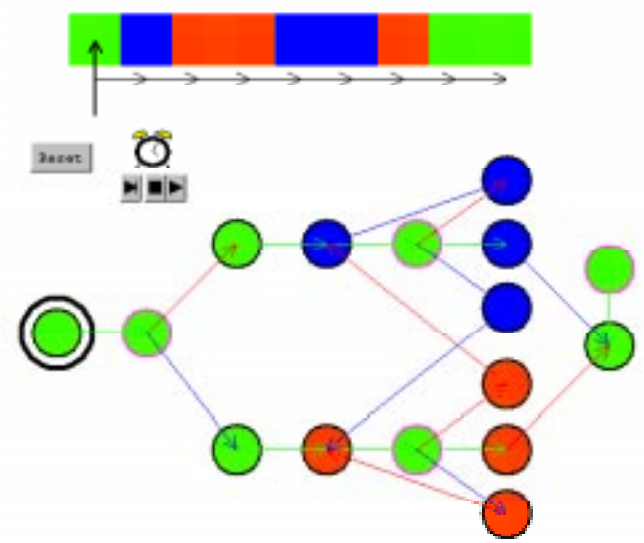


Figure A.10: This Turing machine simulation has a tape in the upper portion that records the program state and a finite state machine in the bottom portion to tell the Turing machine what to do.

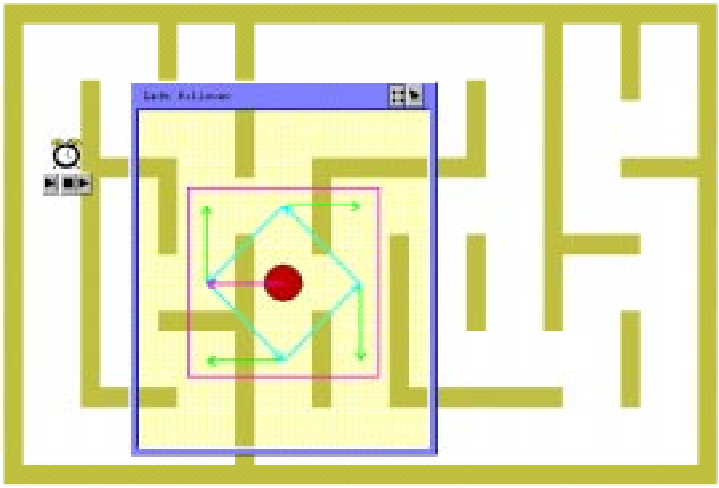


Figure A.11: In the Left Hand Maze Follower application, the circle follows the left wall of the maze by progressively testing the walls with arrow lines.

A.12 Monster Hunt

The Monster Hunt game (see Figure A.12) is a prototype that could have been used for the final usability experiment. It also appeared in video clips. It consists of a player character (a rectangle) that is moved using four directional buttons, and a monster that moves randomly. The player and the monster are not allowed to move through walls. This application was also used to test behaviors for the Pacman application.

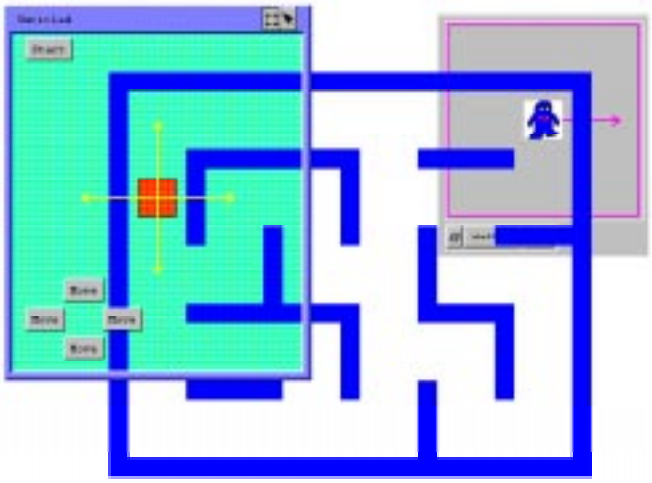


Figure A.12: In the Monster Hunt game, the player moves a circle object and the computer moves a monster object randomly.

A.13 The Q Series

At an early stage of Gamut's implementation, there was a question about how intelligent Gamut could be at learning path following. The question was later reduced to whether or not Gamut could be used to demonstrate the ball behavior in a game like Q*bert [35]. In Q*bert the ball object bounces down a pyramid of cubes randomly choosing which way to fall, see Section A.1. The problem was categorized into three levels of difficulty depending on the configuration of the application. The simplest version was Q1 (see Figure A.13a) in which the path elements themselves have intrinsic differences that Gamut can recognize. In this case, the colors of the left and right lines were colored differently to indicate different directions. In the next case, Q2 (see Figure A.13b), the path elements are identical and the paths are marked using separate objects. In Q2, the ends of the arrows are marked with circles, the goal of the inferencing was to make Gamut choose the lines connected to the red-filled or blue-outlined circles. In the final, unimplemented level, Q3 (see Figure A.13c), the ball follows two lines at a time. The system would have to choose the first arrow based on one criteria and the second arrow based on a different criteria. This would require Gamut to be able to apply a decision tree expression that depended on the length of the path traversed so far. Currently, Gamut only applies the decision tree expression once, so Q3 cannot be implemented in Gamut yet.

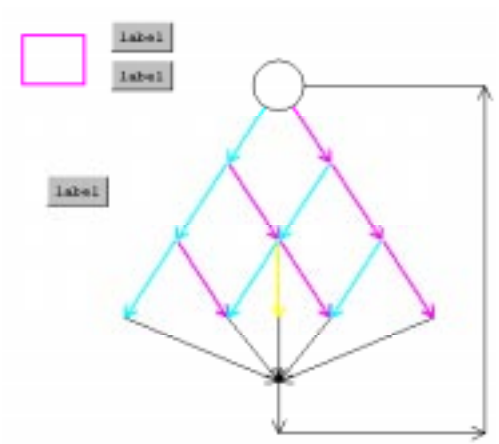


Figure A.13a: In Q1, the colors of the lines indicate which way the circle moves.

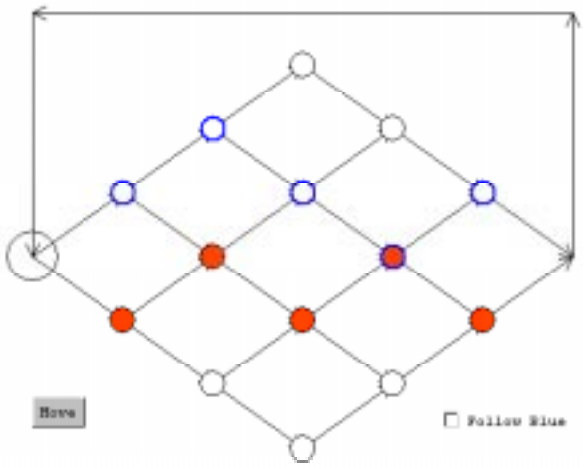


Figure A.13b: In Q2, the colors of circles at the ends of the lines indicate which way the circle moves.

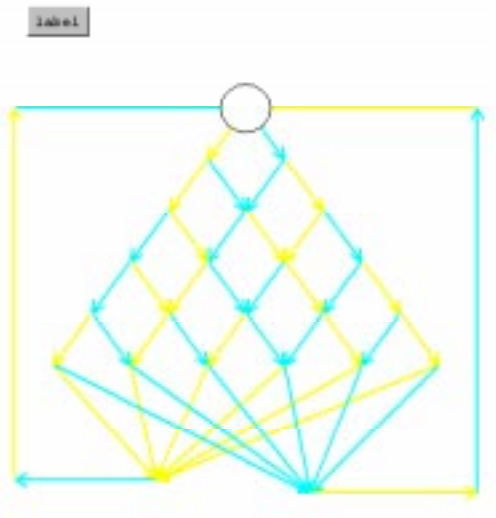


Figure A.13c: In Q3, the colors of the lines once again indicate which way the circle moves, but the circle must move two jumps at a time.

A.14 Parts of Chess

In this thesis' proposal, it was claimed that Gamut would be able to make a Chess game. Though Gamut's inferencing is likely strong enough to finish this game, each time Chess was attempted the amount of debugging required to finish the game was too much to continue. As time progressed, interest in making a full Chess game waned and the application was never finished. The Chess game shown in Figure A.14 had implemented alternating between two-players and dragging pieces. Landed on pieces would be moved off the board. However, the game did not restrict the players to make legal moves.

Figure A.14: This is the unfinished Chess game application. Currently, the only implemented feature is that the players can drag pieces on the board. Conceivably, much more of the game could be built.

A.15 Digital Digits

This application was written quickly in order to be shown in a video. It consists of three decks each containing ten groups that look like digital numbers (see Figure A.15). When the player pushes the button, the last deck would change to be the sum of the first two decks.

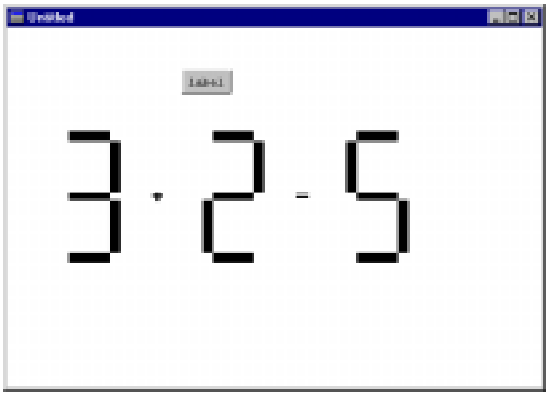


Figure A.15: The digital-looking numbers are stored in decks. The last deck was made to add together the values of the first two decks when the player pushed button at the top.

Appendix B: Paper Prototype Experiment Materials

This appendix includes all the material used to create this project's paper prototype experiment (see Chapter 7). The materials include the various forms the participant had to fill out as well as pictures of the items used in the experiment. The pictures were originally in color. They were cut out and pasted onto cardboard. Appendix C shows the results of this experiment.

B.1 Consent Form

This is the consent form that participant's were asked to fill out before the experiment began. It is based on the standard form that all studies conducted at CMU require.

Carnegie Mellon University

Consent Form

Project Title: Gamut
Conducted By: Richard G. McDaniel, Computer Science Department

I agree to participate in the observational research conducted by students or staff under the supervision of Dr. Brad Myers. I understand that the proposed research has been reviewed by the University's Institutional Review Board and that to the best of their ability they have determined that the observations involve no invasion of my rights and privacy, nor do they incorporate any procedure or requirements which may be found morally or ethically objectionable. I understand that my participation is voluntary and that if at any time I wish to terminate my participation in this study, I have the right to do so without penalty. I understand that I will be paid $10 for my participation when I have completed the experiment.

If you have any questions about this study, you should feel free to ask them now or anytime throughout the study by contacting:

Dr. Brad Myers
HCI Institute, School of Computer Science
412-268-5150
bam@cs.cmu.edu

You may report any objections to the study, either orally or in writing to:

Susan Burkett
Associate Provost
Carnegie Mellon University
412-268-8746

Purpose of the Study: I understand that I will be using a prototype interface of a game building tool. I know that the researchers are studying how people would use such a tool to build software. I realize that in the experiment, I will be asked to implement games using this interface for 1-2 hours. I am aware that I will be videotaped during the experiment so that the researchers can examine how I performed the tasks.

I understand that the following procedure will be used to maintain my anonymity in analysis and publication/presentation of any results. Each participant will be assigned a number, names will not be recorded. The researchers will save the data and videotape files by participant number, not by name. Only members of the research group will view the tapes in detail. No other researchers will have access to theses tapes.

I understand that in signing this consent form, I give Dr. Brad Myers, and his associates permission to present this work in written/oral form, without further permission from me.

_____ _____
Name Signature

_____ _____
Telephone Date

Optional Permission: I understand that the researchers may want to use a short portion of the videotape session for illustrative reasons in presentations of this work. I give my permission to do so provided that my name, face, and voice will not appear.

_____ YES _____ NO (Please initial here _____)

B.2 Survey

This is the survey that participants filled out before the experiment. It was used to obtain basic information about the participant and to gauge how interested the person would be about creating game applications.

Survey

Subject No.: _____ Age: _____ Sex: _____ Male _____ Female

Do you own a computer? _____ YES _____ NO

What do you use computers for?:

Do you use a computer in your work? _____ YES _____ NO

Can you program a computer? _____ YES _____ NO

Do you program computers regularly (as a profession or as a significant hobby)?
_____ YES _____ NO

Do you use a computer to play games? _____ YES _____ NO

If so, what games do you like to play?

Do you play non-computer games like board games or puzzles? _____ YES _____ NO

If so, what non-computer games do you like to play?

If it were easy enough to do, would you write your own computer games?
_____ YES _____ NO

If so, would you want to (check as many as you like):

_____ Design original games.

_____ Port non-computer games to the computer.

_____ Other. Please specify:

B.3 Questionnaire

This is the questionnaire that the participants were asked to fill out when the experimental session was completed. It asked the participant to rank how difficult Gamut was to use and asked the participant for suggestions on how to improve Gamut's interface.

Questionnaire

Subject No.: _____

Which task did you find the easiest to do? What made the task easy?

Which task did you find the most difficult? Why was it difficult?

Did you experience problems with any of the following:
1. Understanding how to carry out the tasks (check one):

_____ no problems _____ minor problems _____ major problems

Please explain:

2. Knowing what to do next (check one):

_____ no problems _____ minor problems _____ major problems

Please explain:

What are the best aspects of Gamut for the user?

What are the worst aspects of Gamut for the user?

What changes should I make to improve Gamut so people can use it better?

B.4 Experimenter's Spoken Directions

Before a session began, the experimenter would talk to the participant and give a basic set of directions. This list shows the points that the experimenter would make sure to mention each time.

Opening Statements, Directions

Description of the observation:

- You are helping me by trying out a mock-up of a game building tool.
- I am testing the interface; I am not testing you.
- I am looking for places where the tool may be difficult to use.
- If you have trouble with some of the tasks, it is the tool's fault, not yours. Do not feel bad; that is exactly what I am looking for.
- If I can locate trouble spots, then I can build a better system for people to use.
- This is totally voluntary. Although I do not know of any reason for this to happen, if you become uncomfortable or find this objectionable in any way, feel free to quit at any time.

Demonstration of paper prototype:

- This is a mock-up of an actual computer interface.
- I will act as the computer and change the mock-up as if the computer were responding to your actions.
- You can manipulate the interface by pretending your finger is the mouse pointer.
- Please tell me when you're using your finger to click on a button or select a screen item. It is sometimes difficult to tell when I'm just watching.
- If you want to draw something or have the interface record a value, write it down on the paper with this marker.
- I have found that I get a great deal of information from these informal observations if I ask people to think-aloud as they work through the exercises.
- It may be a bit awkward at first, but it's really very easy once you get used to it.
- If you forget to think aloud, I'll remind you to keep talking.

B.5 Post-Experiment Statement

When each experiment session was complete, the experimenter would debrief the participant using this statement.

Post-Experiment Information

Thank you for participating in our experiment. The purpose of the experiment was to see whether nonprogrammers could use Gamut's new programming techniques to build games. We wanted to see which parts of the interface are confusing and difficult to use. We want to see how a person wanting to make a game uses and understands Gamut's techniques and see if they can be composed to build useful behavior. Gamut has not been implemented, yet, and your help will be used to make Gamut's interface better.

B.6 Tutorial Section

The first task the participant performed was a tutorial designed to acquaint the participant with Gamut's functions. This material was presented on a single sheet of paper. The experimenter was allowed to help the participant as much as needed.

Skills Practice

- **Make a game piece move a step.**
- **Make a game piece move continuously.**
- **Create a small deck of cards.**
- **Use highlighting to demonstrate a choice.**
- **Create a button.**
- **Create a number box to count number of times button is pushed.**
- **Show creating a game piece when count reaches 5.**
- **Show erasing a game piece when count reaches 7.**

B.7 The Paper Prototype Tasks

There were three tasks that were part of the paper prototype. Each of these tasks were divided into bulleted sections that the participant was asked to perform in order. Each task was printed on an individual sheet of paper.

B.7.1 Pawn Race

Task 1: Pawn Race

- **Make it so that when the dice are rolled, a pawn moves the dice's number of squares.**
- **Add another pawn and make the dice rolling work for two players.**
- **When one player's pawn lands on the other, the other player should be moved back to START.**
- **Make the game end when the first pawn makes a complete circuit of the board.**

B.7.2 Pacman

Task 2: Pacman

- **Make the monster move through the maze.**
- **Make the monster chase the Pacman.**
- **When the monster touches the Pacman, make the player lose a life.**

- **When the player loses three lives, make the game end.**

B.7.3 Space Shooter

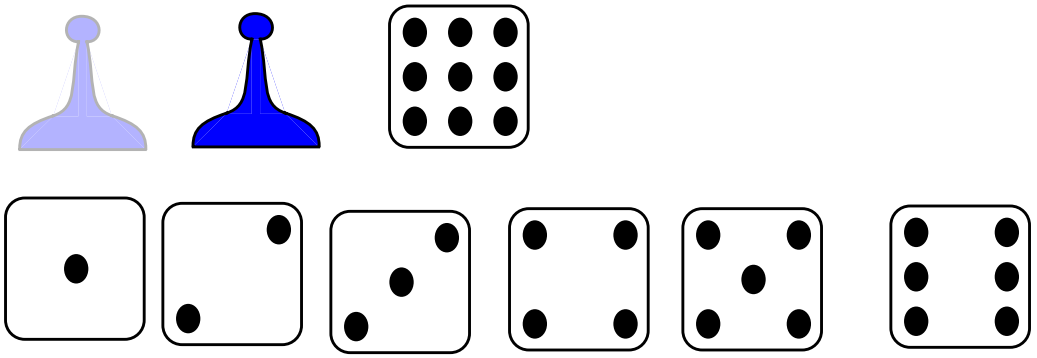
Task 3: Space Shooter

- **Make the UFO move about randomly.**
- **Make the fire button cause the spaceship to shoot a bullet.**
- **When the bullet hits the UFO, make the UFO disappear, then reappear somewhere else.**
- **Have the player win when three UFO's are shot.**

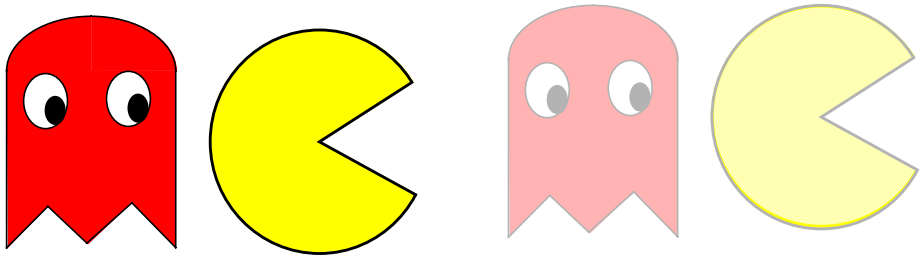
B.8 Paper Prototype Materials

These are pictures of the graphics used in creating the paper prototype's interface. Each task consisted of a background board and paper cutout pieces that would be used to act out the demonstration.

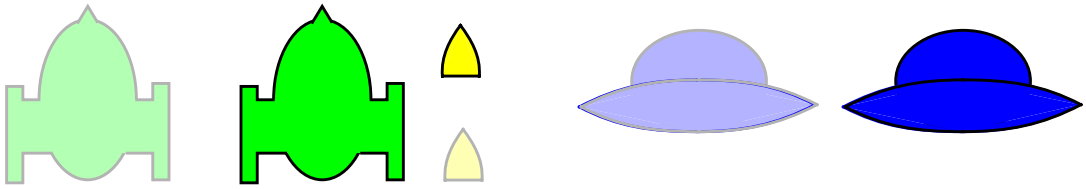
B.8.4 Task 1 Pieces



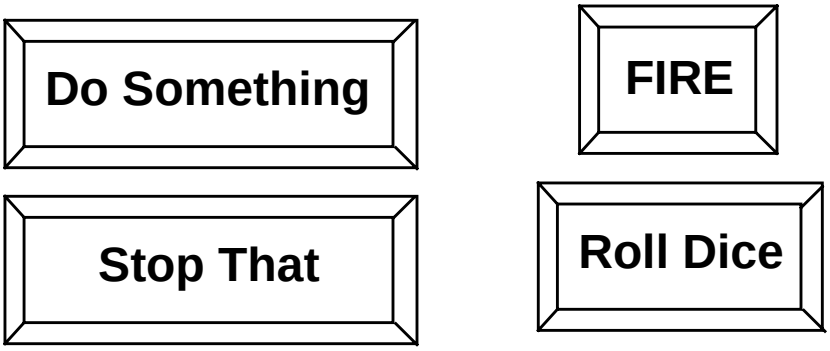
B.8.5 Task 2 Pieces

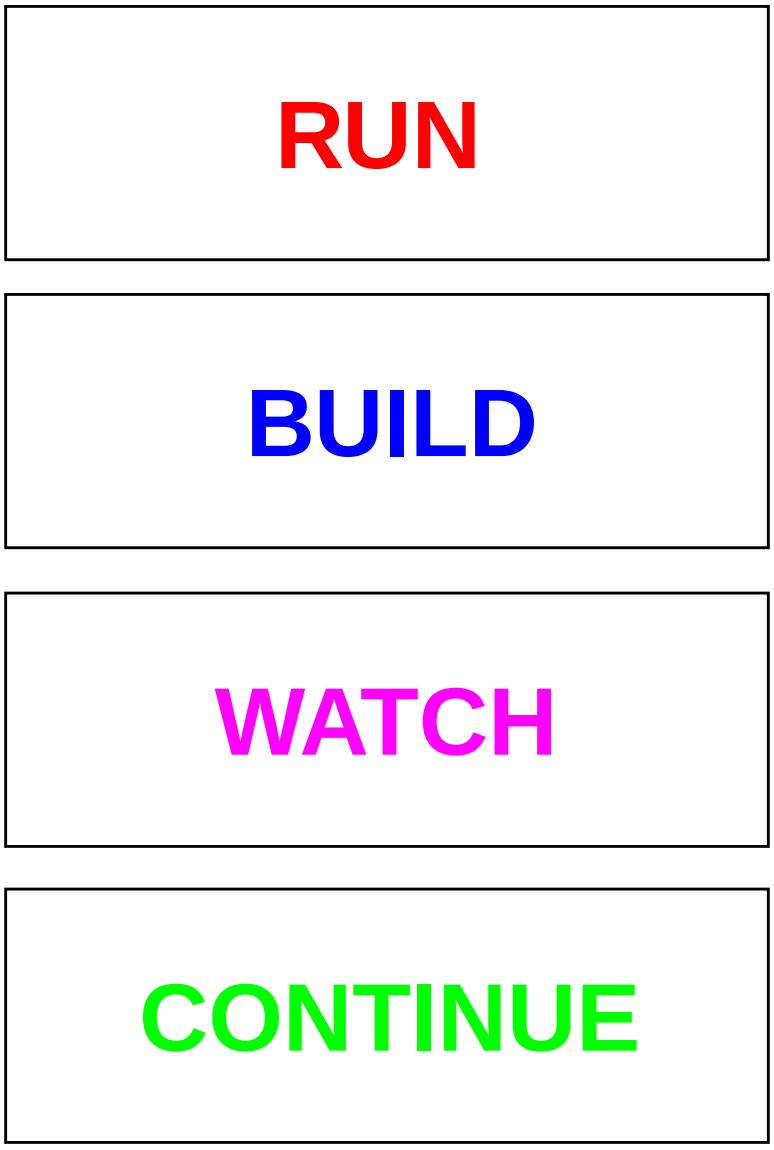


B.8.6 Task 3 Pieces

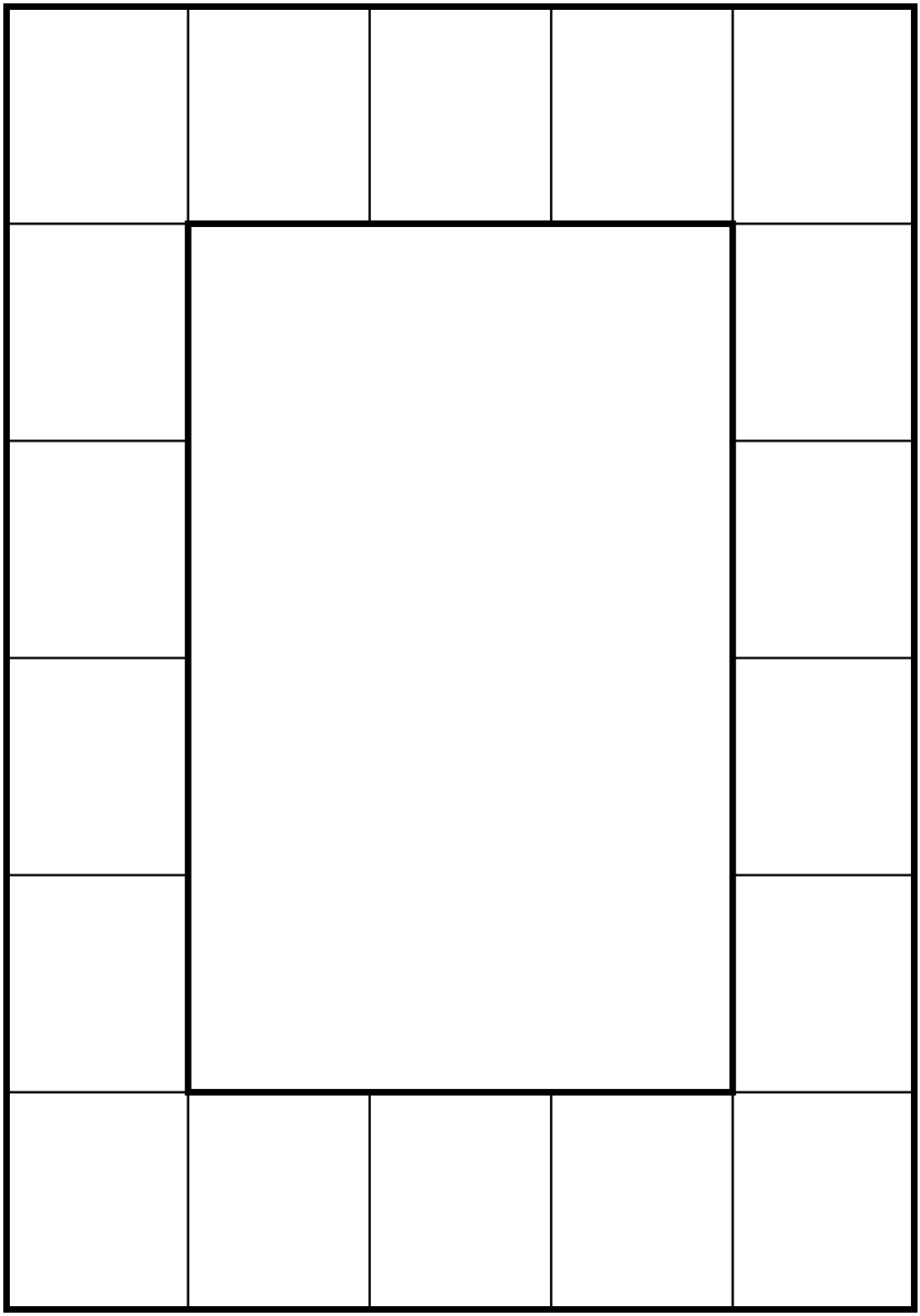


B.8.7 Buttons and Modes

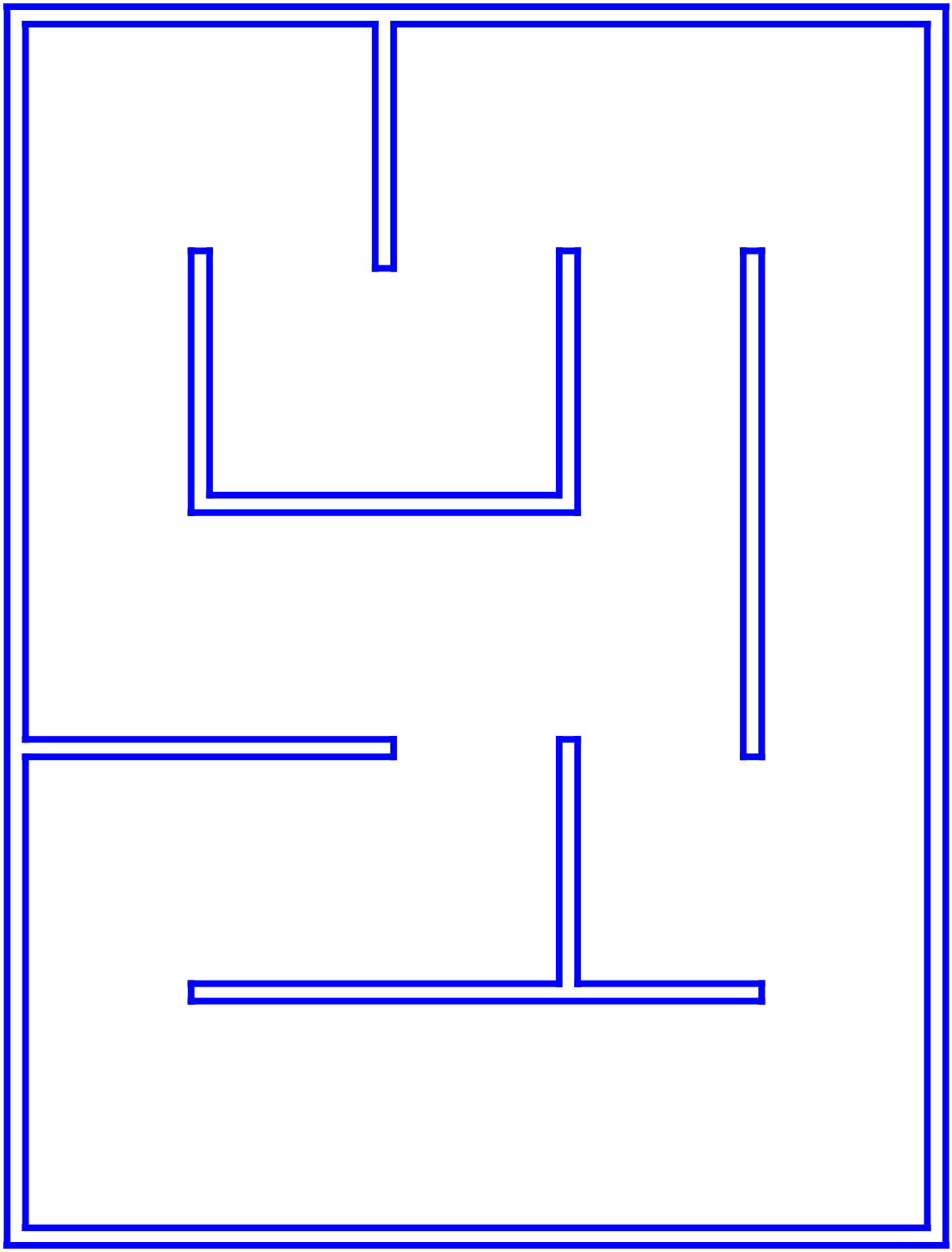




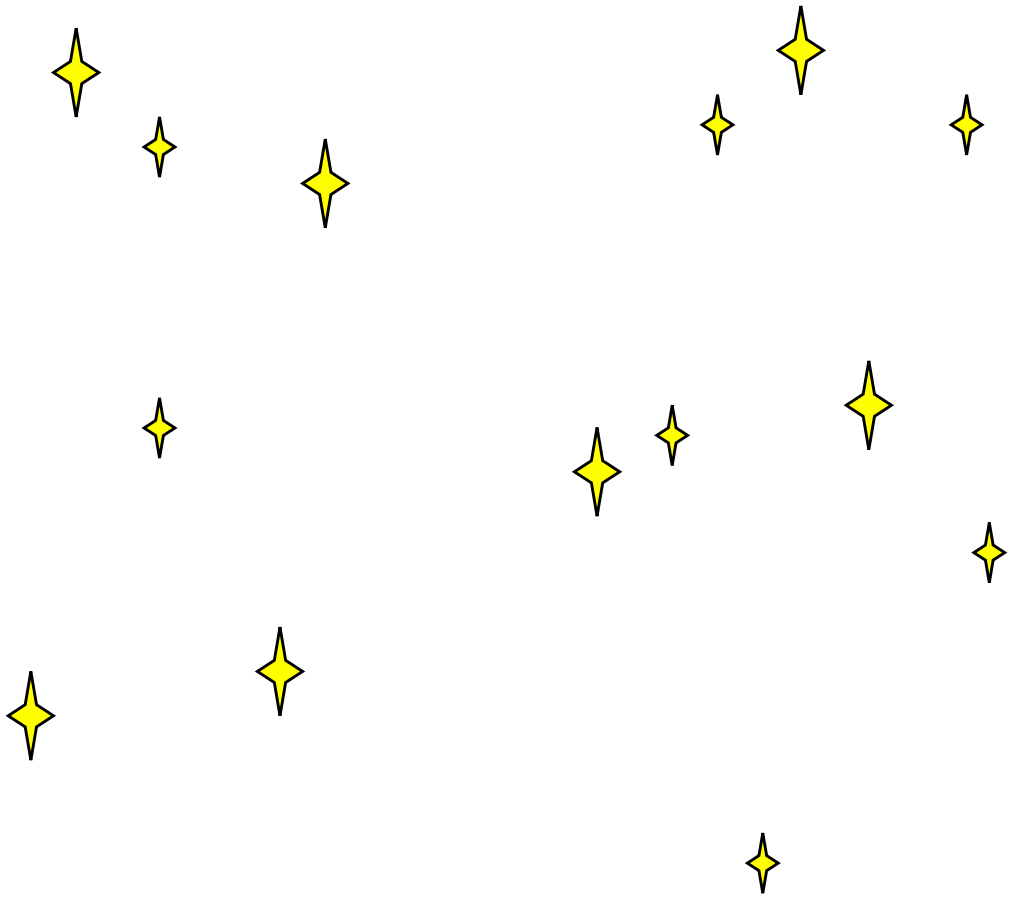
B.8.8 Task 1 Background



B.8.9 Task 2 Background

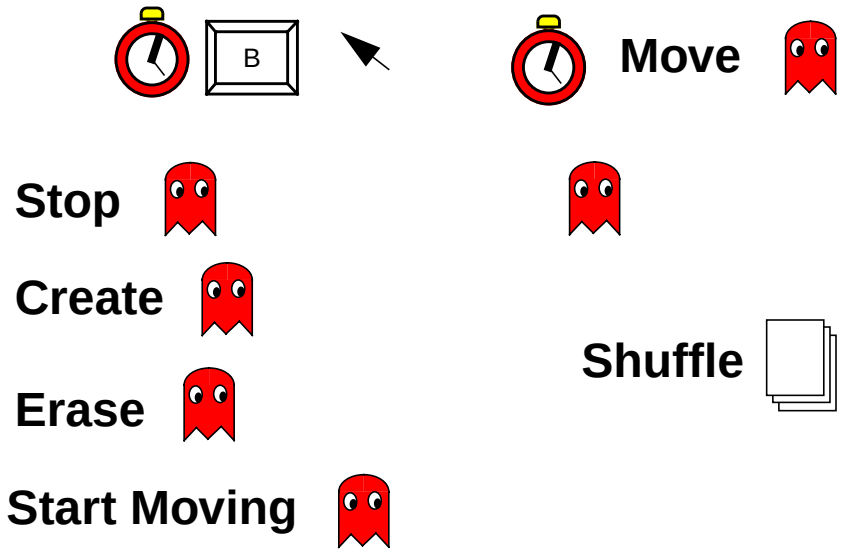


B.8.10 Task 3 Background



B.8.11 Time Line Dialog

B.8.12 Time Line Contents



Appendix C: Paper Prototype Experiment Results

This appendix lists the results of the paper prototype experiment. The results consist of the answers to the surveys and the drawings the participants made on the background sheet and their offscreen area sheet. Some participants also used cards to represent decks for some tasks. All written materials have been reduced for space. A total of five subjects participated in the experiment. In the results, each participant's answers will be labelled with the numbers one through five.

C.1 Form Results

The paper prototype experiment used two forms: the pre-test survey and the post-test questionnaire. In this section, each participant's responses are listed.

C.1.1 Survey

Subject No.: _____ Age: _____ Sex: _____ Male _____ Female
Age: (1) 26, (2) 21, (3) 21, (4) 21 (5) 17
Sex: (1) Male, (2) Male, (3) Male, (4) Female (5) Male

Do you own a computer? _____ YES _____ NO
(1) Yes, (2) Yes, (3) Yes, (4) Yes (5) Yes

What do you use computers for?:
(1) Word Processing, Number Crunching, Graphing, Occasional FORTRAN programming

(2) - PROCESS INFO
- CALCULATIONS
- WORD PROCESSING
- email (communication)

(3) Word processing
Desktop publishing
Game playing

(4) Word processing, e-mail, www

(5) Playing games
www
email

Do you use a computer in your work? _____ YES _____ NO
(1) Yes, (2) No, (3) Yes, (4) Yes, (5) Yes

Can you program a computer? _____ YES _____ NO
(1) Yes, (2) Yes, (3) No, (4) Yes, (5) Yes

Do you program computers regularly (as a profession or as a significant hobby)?
_____ YES _____ NO
(1) No, (2) No, (3) No, (4) No, (5) No

Do you use a computer to play games? _____ YES _____ NO
(1) Yes, (2) No, (3) Yes, (4) Yes, (5) Yes

If so, what games do you like to play?
(1) Strategic War games, chess once in a while, Solitaire, Taipei

(2) --

(3) Warlords 2
Stratomatic baseball
Loony Labyrinth pinball
Solitaire
(plenty of others)

(4) Card games, Spaceward Ho!, Warcraft, tetris....

(5) Wing Commander
DOOM
warcraft2

Do you play non-computer games like board games or puzzles? _____ YES _____ NO
(1) Yes, (2) No, (3) Yes, (4) Yes, (5) Yes

If so, what non-computer games do you like to play?
(1) occasionally Trivial Pursuit, Taboo, other word games

(2) --

(3) Trivial pursuit
(word games like outburst)
Monopoly
Life

(4) sometimes, crossword puzzles

(5) chess

If it were easy enough to do, would you write your own computer games?

_____ YES _____ NO

(1) Yes, (2) No, (3) Yes, (4) Yes, (5) Yes

If so, would you want to (check as many as you like):

_____ Design original games.

(1) Yes, (2) -- (3) Yes, (4) Yes, (5) Yes

_____ Port non-computer games to the computer.

(1) No, (2) -- (3) No, (4) No, (5) No

_____ Other. Please specify:

(1) No, (2) -- (3) No, (4) No, (5) No

C.1.2 Questionnaire

Which task did you find the easiest to do? What made the task easy?

(1) Pawn Race Task - task was easier b/c of simplicity in motion (simplicity being the clockwise path)

(2) PAWN

(3) 2nd -> Pacman - the 1st one, I was getting used to the system; the 3rd was more advanced

(4) make buttons - you just had to draw them and tell it what to do when they were pushed.

(5) Making the Monster follow the Pacman

Which task did you find the most difficult? Why was it difficult?

(1) Pacman - needed better understanding of how to use various tools together

(2) PACMAN / LEARNING CURVE

(3) 3rd -> UFO -> complexity of the tasks

(4) make the pieces go to start when they landed on each other (I had to wait for it to actually happen)

(5) making continuous motion I found it hard to grasp the concept

Did you experience problems with any of the following:

1. Understanding how to carry out the tasks (check one):

_____ no problems _____ minor problems _____ major problems

(1) Major, (2) Minor, (3) Minor, (4) Major, (5) Major

Please explain:

(1) Dove in too quickly - needed to reflect a bit more first on what boxes/buttons I might have to create. Some trouble figuring out how to implement more complex tasks.

(2) {AMBIGUOUS SOMETIMES IN MOVEMENT}

(3) Took some time to get used to the way each thing worked

(4) I don't know how to make the computer understand that something needs to be done If something else happens

(5) I had a lot of trouble figuring out how to make the system understand why I made things change their movements

2. Knowing what to do next (check one):
_____ no problems _____ minor problems _____ major problems
(1) Minor, (2) Major, (3) No problems, (4) Major, (5) Minor

Please explain:
(1) As above, needed to recognize that more aspects of the game are required than just the list of rules - for example, setting up the playfield eluded me.

(2) --

(3) --

(4) I can't tell the comp. how to accomplish a number of tasks at once

(5) I usually tried to figure out what didn't work.

What are the best aspects of Gamut for the user?
(1) Only need a handful of tools to potentially set up a fairly complex/fun situation.

(2) [participant circled the word Gamut] GUESS THING -> LEARNed what I wanted to do not always Right though.

(3) Simple interface & icons/buttons/etc.

(4) lots of dialogue boxes

(5) The system can learn from what the user does.

What are the worst aspects of Gamut for the user?
(1) Could use a few examples at the start of how to combine several tools to make an event happen (more complex ones than a single move, for example).

(2) What ORDER do I do what?

(3) It takes time to get used to - it probably needs a good manual

(4) NO if.

(5) Trying to make it understand what I did, and why I did it

What changes should I make to improve Gamut so people can use it better?

(1) Since most games tend to have barriers (the screen edge [admittedly, there are some exceptions], various obstacles), there could be a key at the start that automatically allows barrier definition.

(2) Some tasks have PREmaid buttons like picking #5 from the deck.

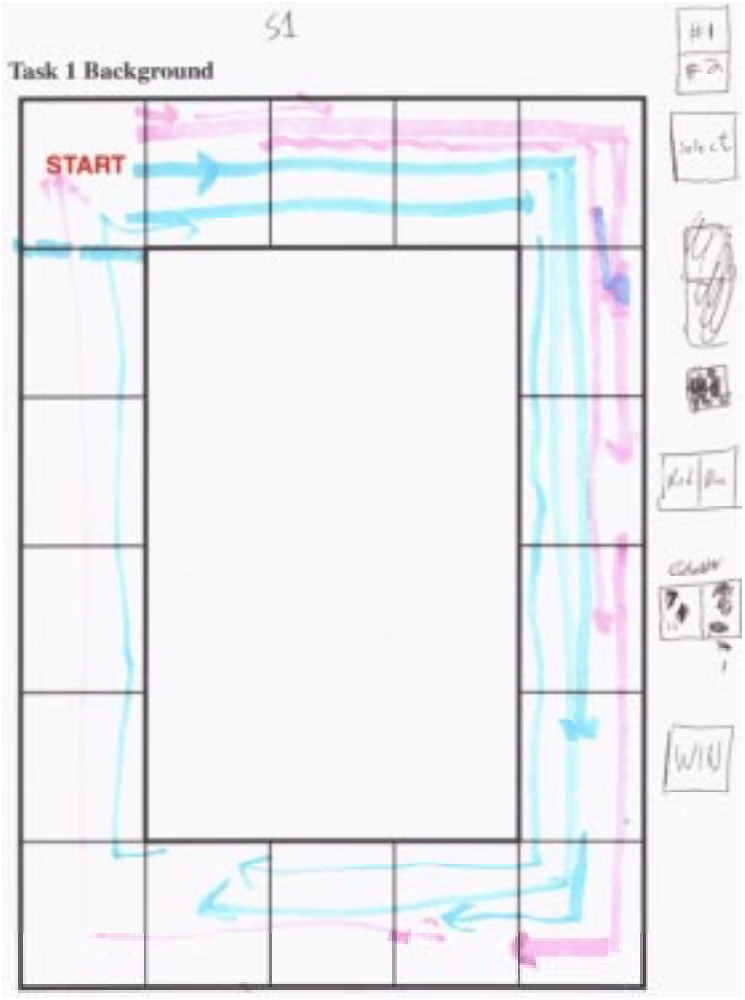
(3) Explain the logic the system uses - why it does what it does
Add more help/manuals to explain and for reference

(4) do an if thing

(5) Not rely on paths as much, for instantaneous motion.

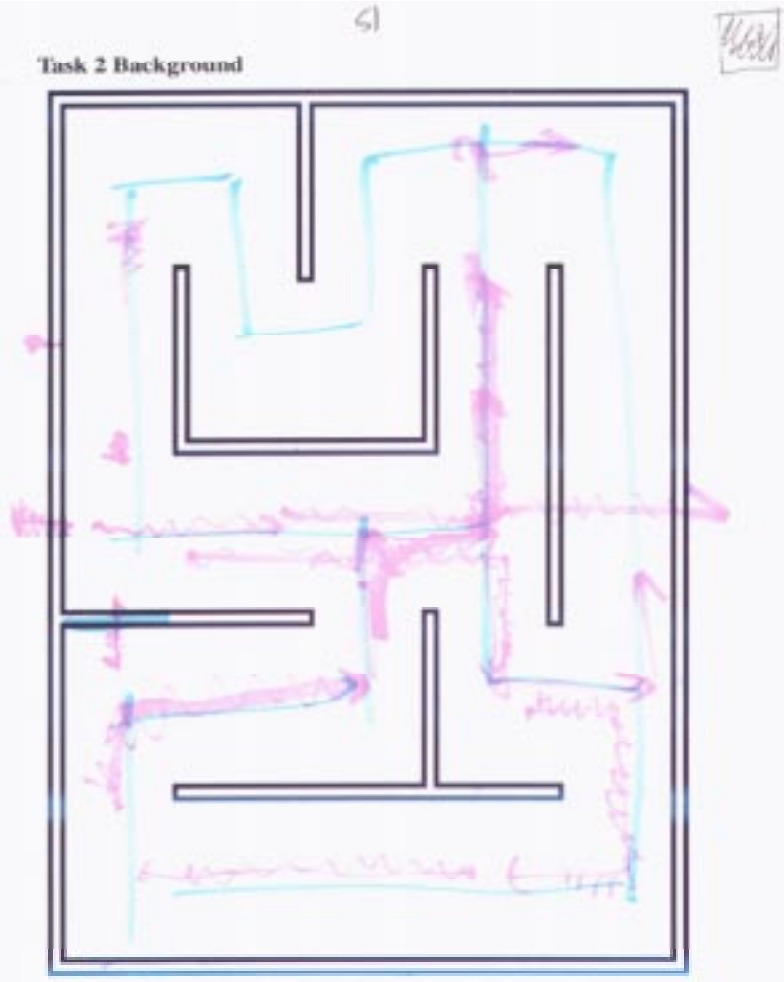
C.2 Participant One's Drawings

C.2.1 Task One: Pawn Race

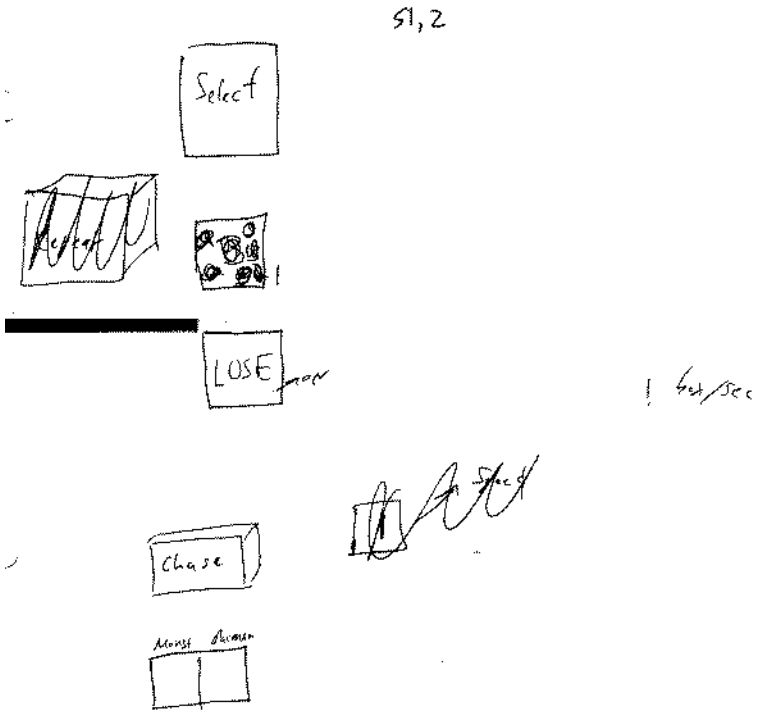


C.2.2 Task Two: Pacman

C.2.2.1 Main Board



C.2.2.2 Offscreen Area



C.3 Participant Two's Drawings

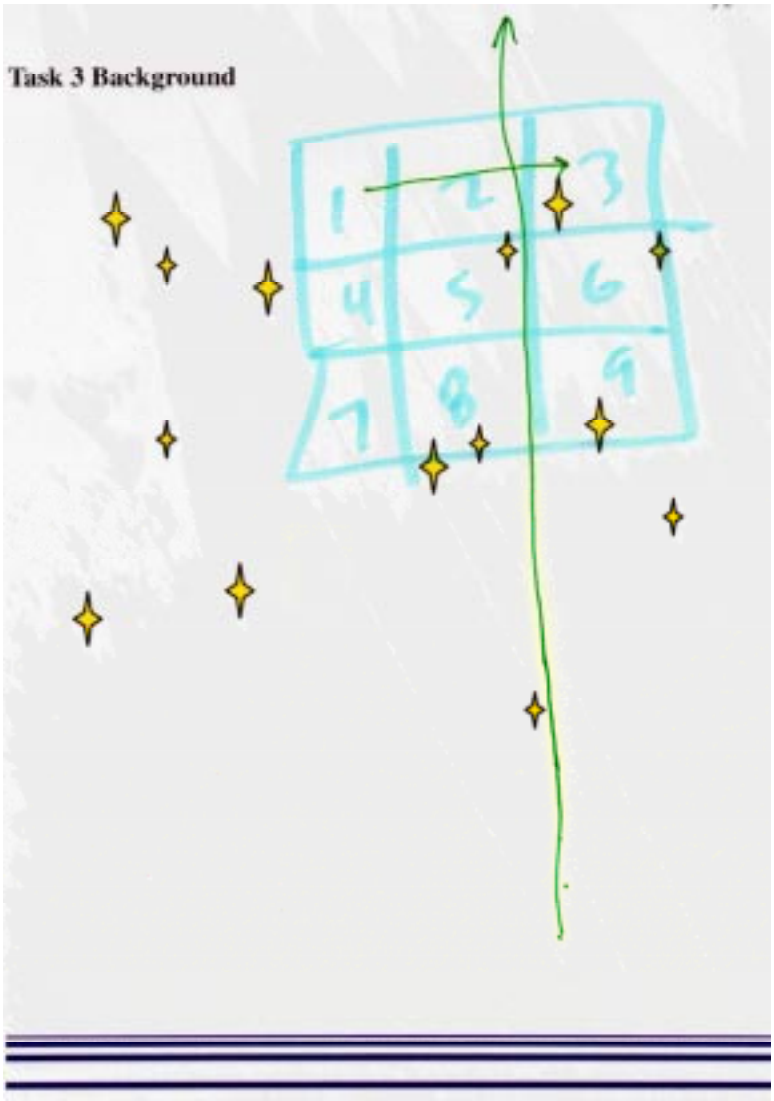
C.3.1 Task One: Pawn Race

C.3.1.1 Main Board

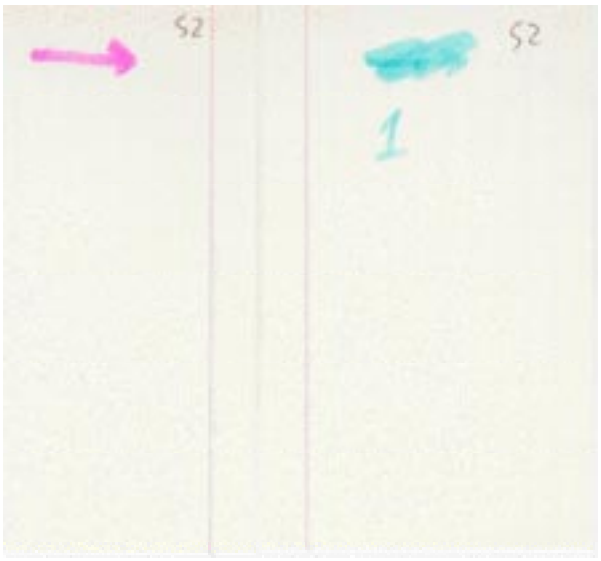
C.3.1.2 Offscreen Area

C.3.2 Task Two: Pacman

C.3.3 Task Three: Space Shooter



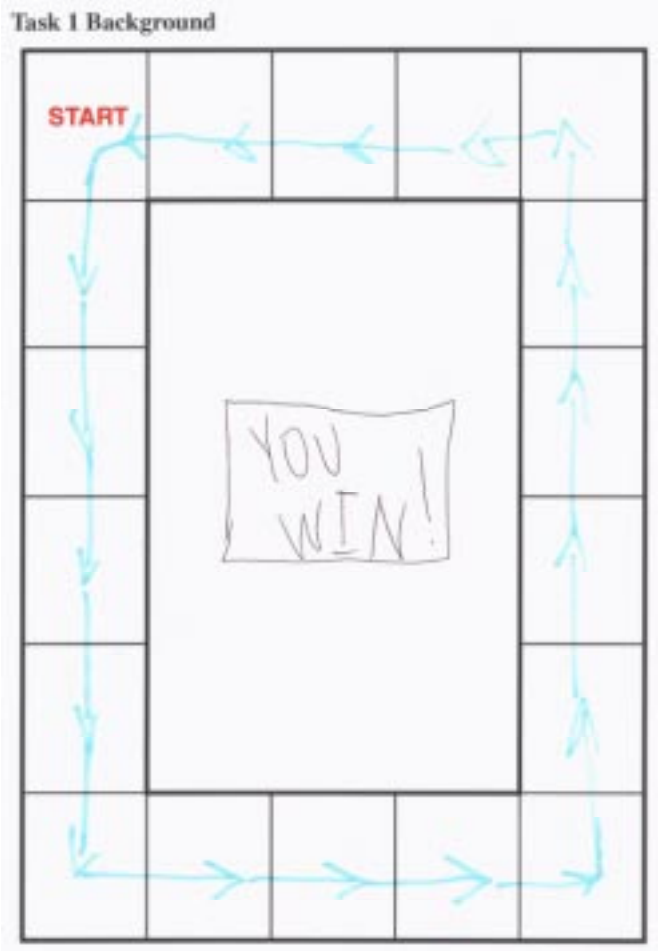
C.3.4 Participant Two’s Cards



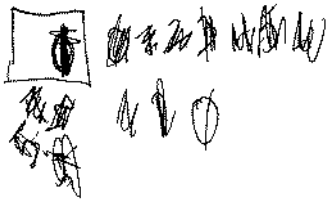
C.4 Participant Three's Drawings

C.4.1 Task One: Pawn Race

C.4.1.1 Main Board

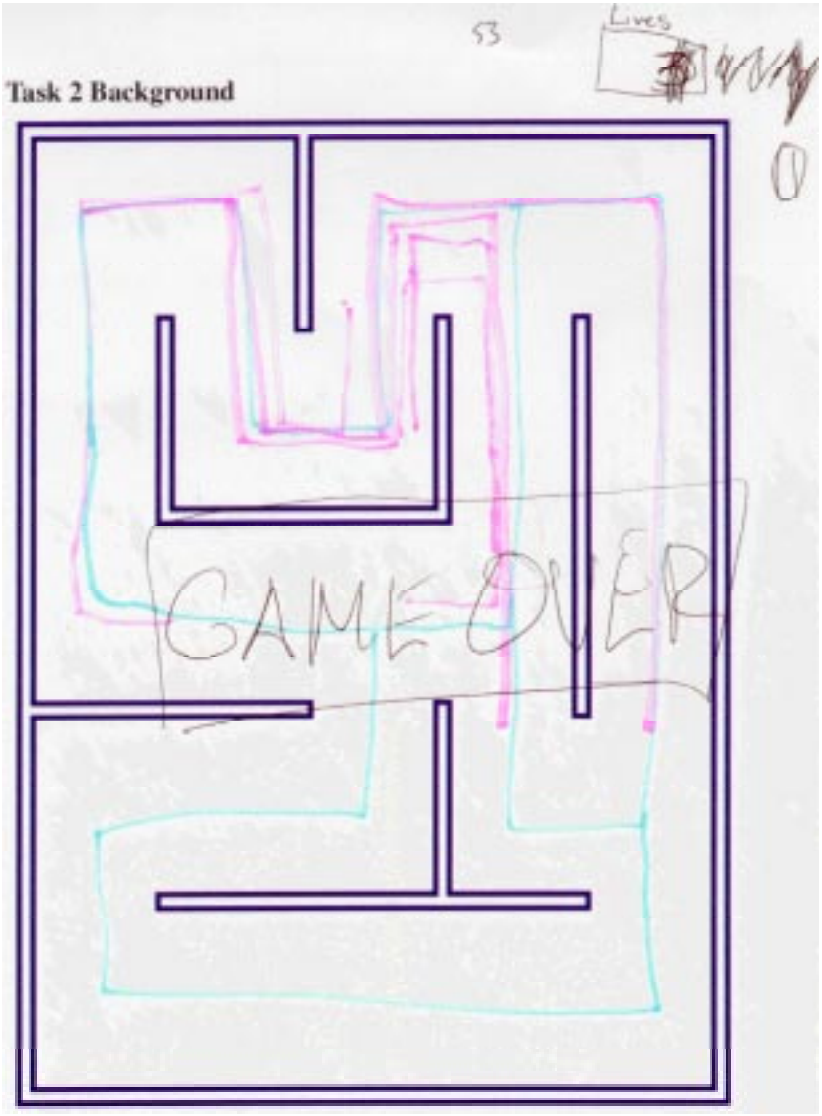


C.4.1.2 Offscreen Area



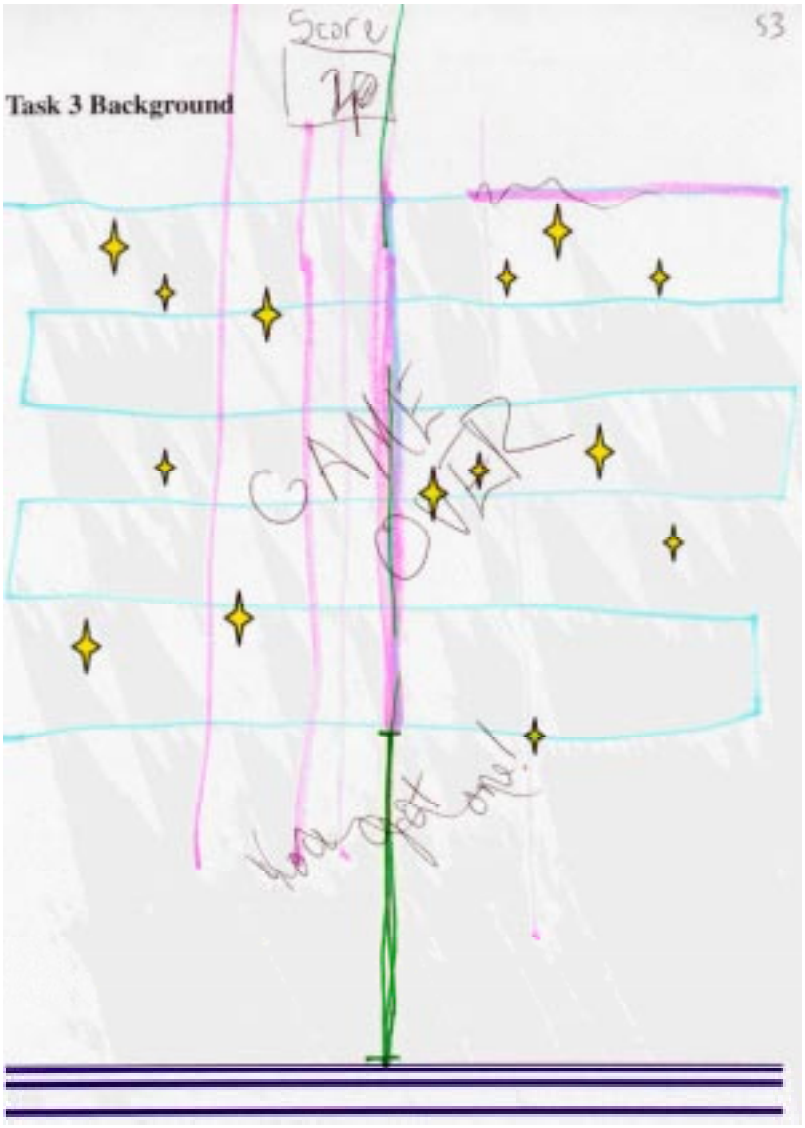
C.4.2 Task Two: Pacman

C.4.2.1 Main Board

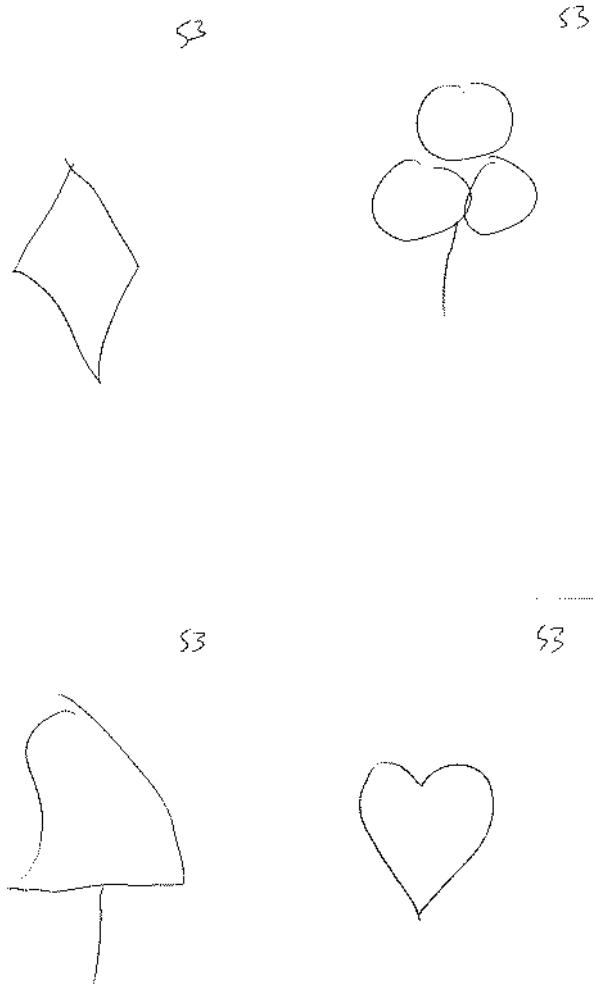


C.4.2.2 Offscreen Area

C.4.3 Task Three: Space Shooter



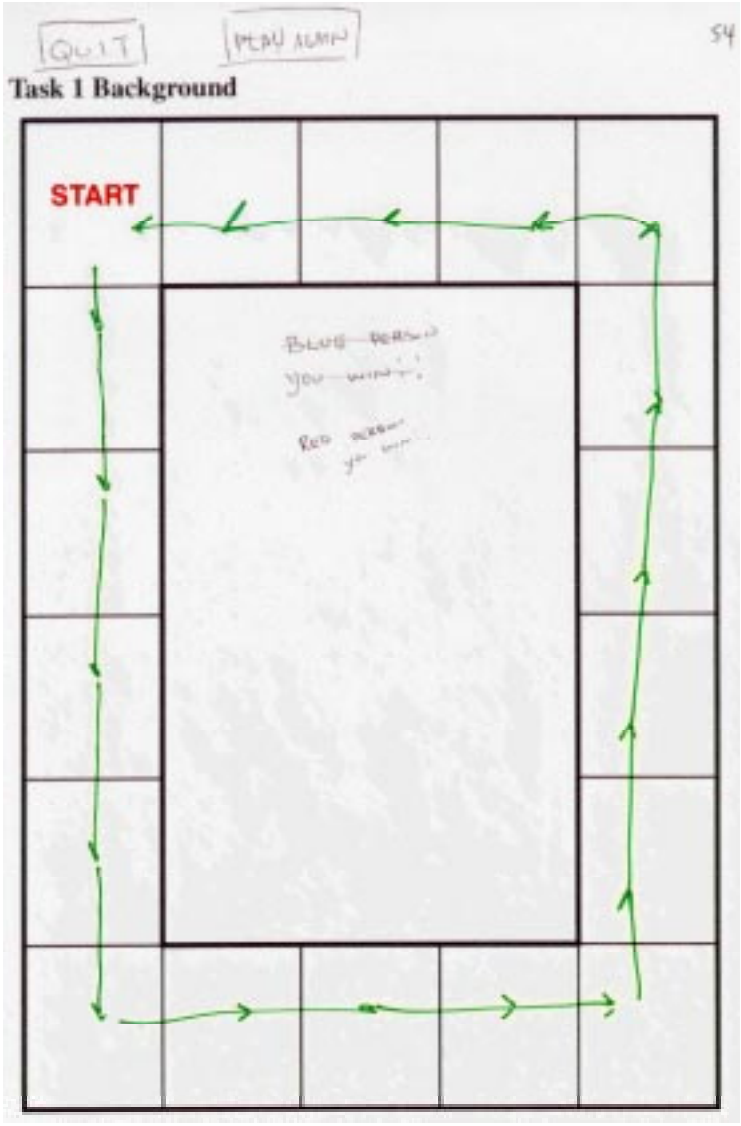
C.4.4 Participant Three's Cards



C.5 Participant Four's Drawings

C.5.1 Task One: Pawn Race

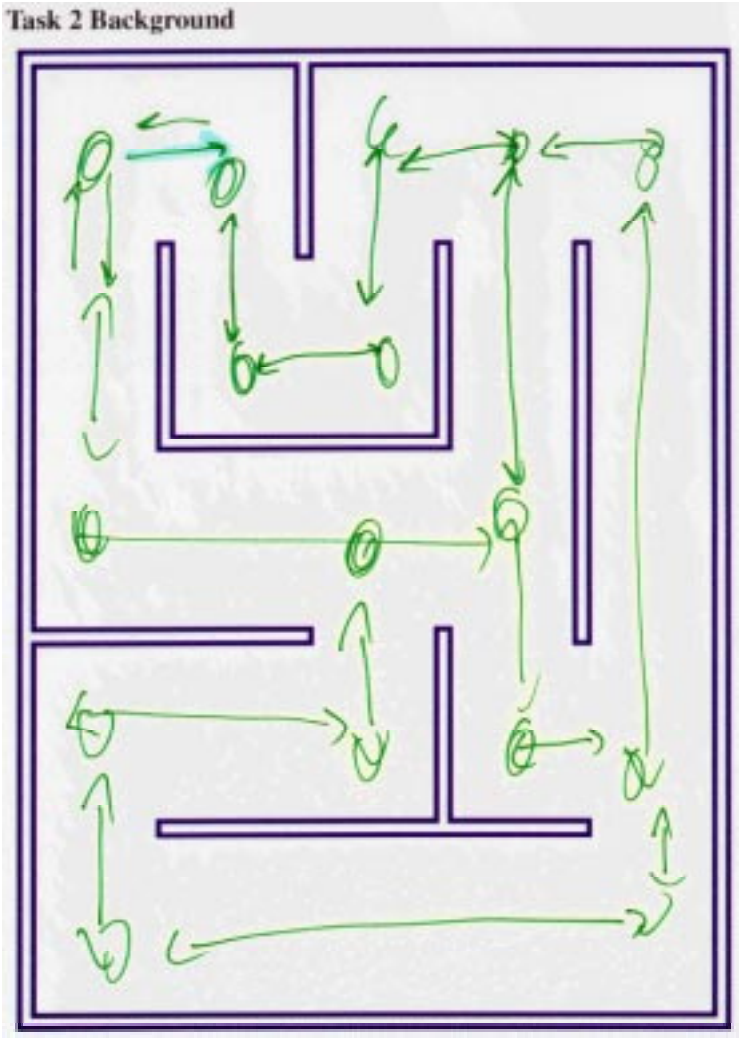
C.5.1.1 Main Board



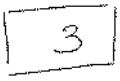
C.5.1.2 Offscreen Area

C.5.2 Task Two: Pacman

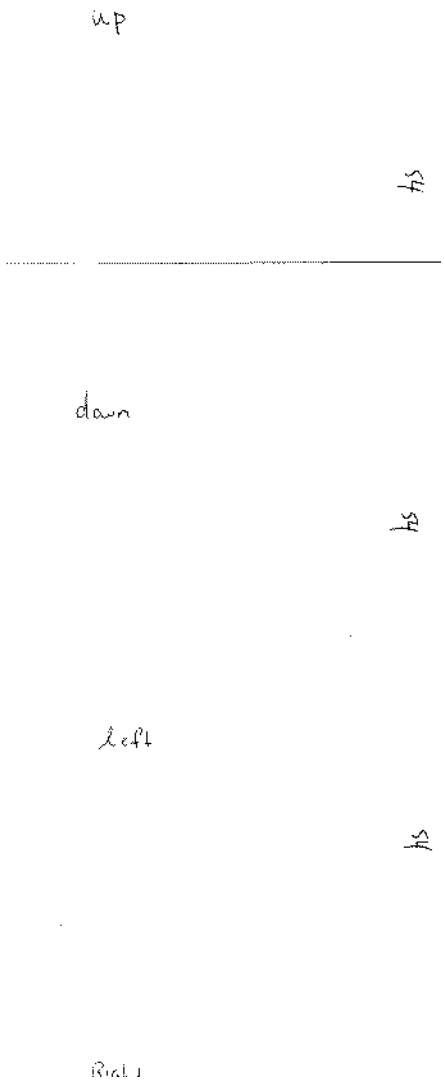
C.5.2.1 Main Board



C.5.2.2 Offscreen Area



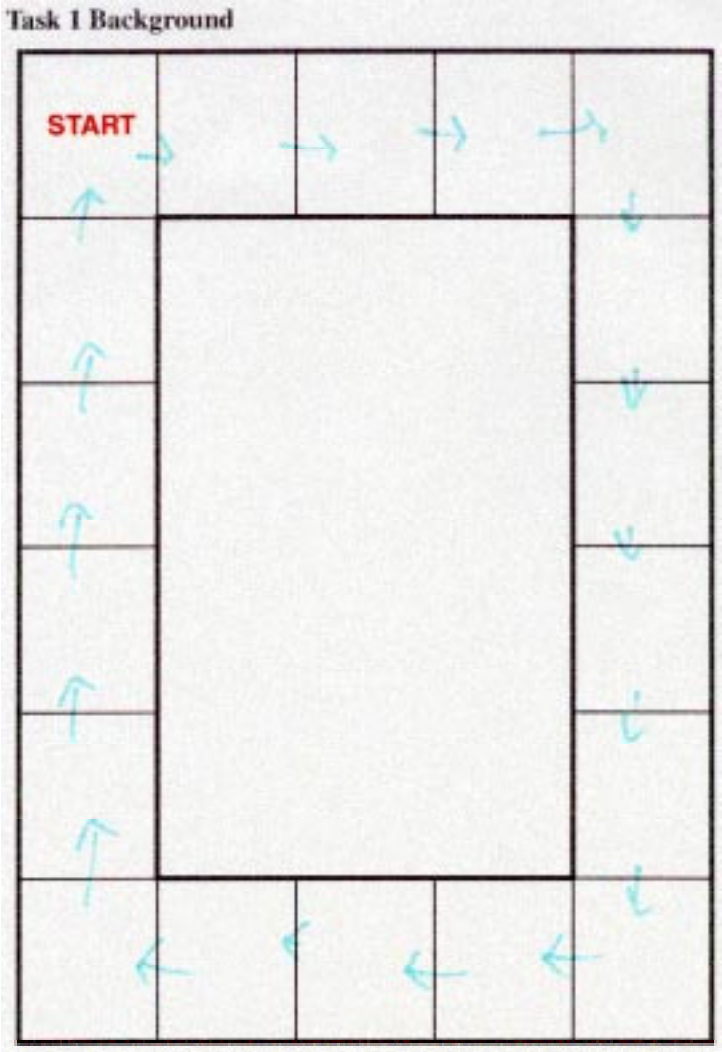
C.5.3 Participant Four's Cards



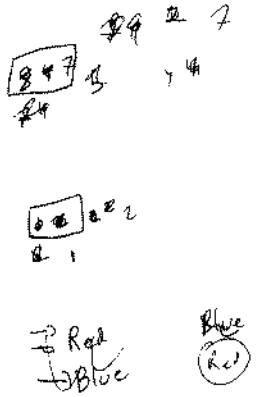
C.6 Participant Five’s Drawings

C.6.1 Task One: Pawn Race

C.6.1.1 Main Board

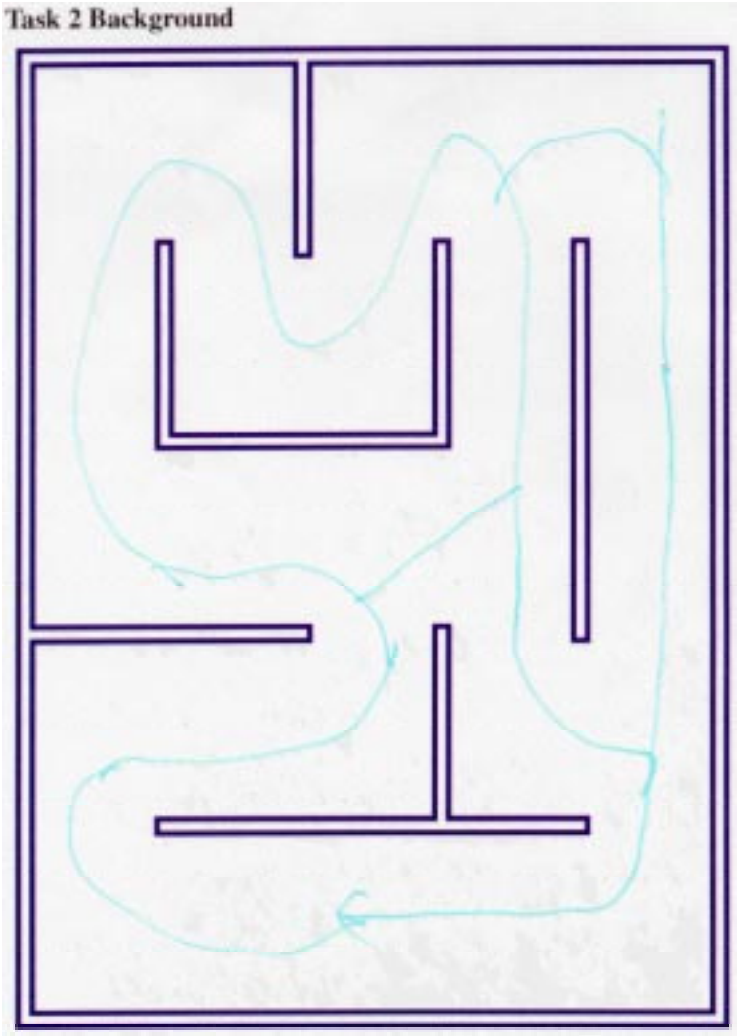


C.6.1.2 Offscreen Area



C.6.1.3 Task Two: Pacman

C.6.1.4 Main Board



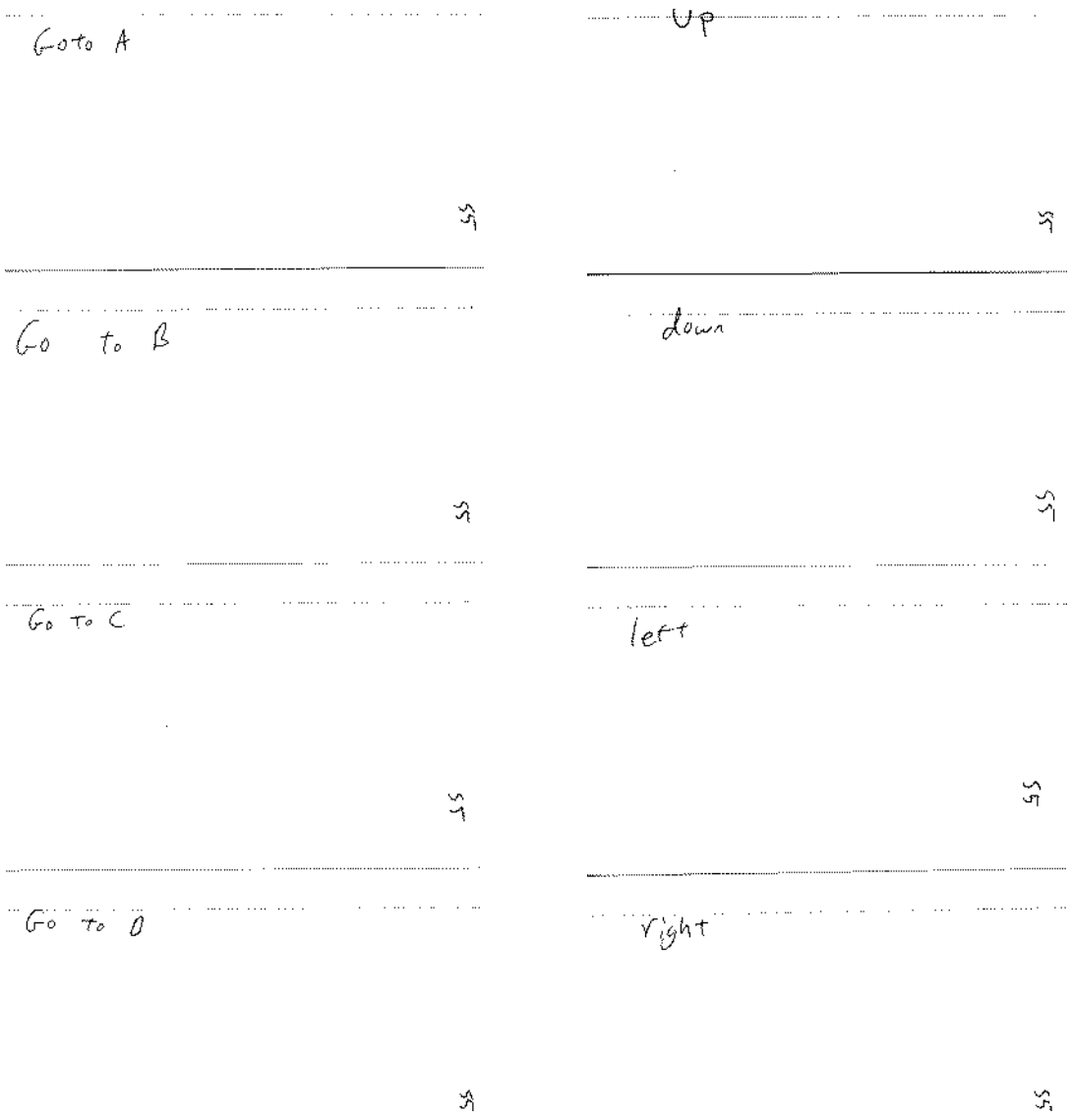
C.6.1.5 Offscreen Area

C.6.2 Task Three: Space Shooter

C.6.2.1 Main Board

C.6.2.2 Offscreen Area

C.6.3 Participant Five's Cards



Appendix D: Usability Experiment Materials

In this appendix, the materials for the final usability experiment (see Section 7.2) are listed. The materials consisted of consent forms, surveys, and a questionnaire similar to the paper prototype experiment's materials (see Appendix B). The task descriptions and the written tutorial are also included. Appendix E shows the results of this experiment.

D.1 Consent Form

This is the consent form that participant's were asked to fill out before the experiment began. It is based on the standard form that all studies conducted at CMU require.

Carnegie Mellon University
Consent Form

Project Title: Gamut
Conducted By: Richard G. McDaniel, Computer Science Department

I agree to participate in the observational research conducted by students under the supervision of Dr. Brad Myers. I understand that the proposed research has been reviewed by the University's Institutional Review Board and that to the best of their ability they have determined that the observations involve no invasion of my rights and privacy, nor do they incorporate any procedure or requirements which may be found morally or ethically objectionable. I understand that my participation is voluntary and that if at any time I wish to terminate my participation in this study, I have the right to do so without penalty. I understand that I will be paid $20 for my participation when I have completed the experiment.

If you have any questions about this study, you should feel free to ask them now or anytime throughout the study by contacting:

Dr. Brad Myers
HCI Institute, School of Computer Science
412-268-5150
bam@cs.cmu.edu

You may report any objections to the study, either orally or in writing to:

Dr. Paul Christiano
607 Warner Hall
Extension 6685

Purpose of the Study: I understand that I will be using an interface of a game building tool. I know that the researchers are studying how people would use such a tool to build software. I realize that in the experiment, I will be asked to implement games using this interface for 2-4 hours. I am aware that I will be videotaped during the experiment so that the researchers can examine how I performed the tasks. I know that the work I create with this tool will be saved for analysis.

I understand that the following procedure will be used to maintain my anonymity in analysis and publication/presentation of any results. Each participant will be assigned a number, names will not be recorded. The researchers will save the data and videotape files by participant number, not by name. Only members of the research group will view the tapes in detail. No other researchers will have access to these tapes.

I understand that in signing this consent form, I give Dr. Brad Myers, and his associates permission to present this work in written/oral form, without further permission from me.

_____ _____
Name Signature

_____ _____
Telephone Date

Optional Permission: I understand that the researchers may want to use a short portion of the videotape session or the games I build for illustrative reasons in presentations of this work. I give my permission to do so provided that my name, face, and voice will not appear.

_____ YES _____ NO (Please initial here _____)

D.2 Survey

This is the survey that participants filled out before the experiment. It was used to obtain basic information about the participant and to gauge how interested the person would be about creating game applications. It was based on the survey used in the paper prototype experiment (see Appendix B.2).

Subject No.: _____ Age: _____ Sex: _____ Male _____ Female

Do you own a computer? _____ YES _____ NO

What do you use computers for?:

Do you use a computer in your work? _____ YES _____ NO

Can you program a computer? _____ YES _____ NO

If so, please explain (Where do you program computers? What language(s) do you use?):

Do you use a computer to play games? _____ YES _____ NO

If so, what games do you like to play?

Do you play non-computer games like board games or puzzles? _____ YES _____ NO

If so, what non-computer games do you like to play?

If it were easy enough to do, would you write your own computer games?
_____ YES _____ NO

If so, would you want to (check as many as you like):

_____ Design original games.

_____ Port non-computer games to the computer.

_____ Other. Please specify:

D.3 Questionnaire

This is the questionnaire that the participants were asked to fill out when the experimental session was completed. It asked the participant to rank how difficult Gamut was to use and asked the participant for suggestions on how to improve Gamut's interface. This was also based on the materials used in the paper prototype experiment (see Appendix B.3).

Subject No.: _____

Which task did you find the easiest to do? What made the task easy?

Which task did you find the most difficult? Why was it difficult?

Did you experience problems with any of the following:
1. Understanding how to carry out the tasks (check one):

_____ no problems _____ minor problems _____ major problems

Please explain:

2. Knowing what to do next (check one):

_____ no problems _____ minor problems _____ major problems

Please explain:

What are the best aspects of Gamut for the user?

What are the worst aspects of Gamut for the user?

What changes should be made to improve Gamut so people can use it better?

D.4 Experimenter's Spoken Directions

Before a session began, the experimenter would talk to the participant and give a basic set of directions. This list shows the points that the experimenter would make sure to mention each time.

(The basic instructions below are based on instructions written by Kathleen Gomoll [34].)

Description of the observation:

- You are helping me by trying out a new game building tool.
- I am testing the interface; I am not testing you.

- I am looking for places where the tool may be difficult to use.
- If you have trouble with some of the tasks, it is the tool's fault, not yours. Do not feel bad; that is exactly what I am looking for.
- If I can locate trouble spots, then I can build a better system for people to use.
- This is totally voluntary. Although I do not know of any reason for this to happen, if you become uncomfortable or find this objectionable in any way, feel free to quit at any time.
- I have found that I get a great deal of information from these informal observations if I ask people to think-aloud as they work through the exercises.
- It may be a bit awkward at first, but it's really very easy once you get used to it.
- If you forget to think aloud, I'll remind you to keep talking.
- In the event that the computer crashes or some problem occurs, I may have to restart the test. This system is only a prototype so finding a problem is not unlikely.
- If you feel that a bug or problem with the system is preventing you from completing a task, just tell me what the problem is and move on to another item.

D.5 Post-Experiment Statement

When each experiment session was complete, the experimenter would debrief the participant using this statement. It is identical to the statement used in the paper prototype experiment (see Appendix B.5).

Thank you for participating in our experiment. The purpose of the experiment was to see whether nonprogrammers could use Gamut's new programming techniques to build games. We wanted to see which parts of the interface are confusing and difficult to use. We want to see how a person wanting to make a game uses and understands Gamut's techniques and see if they can be composed to build useful behavior. Your help will enable us to improve our techniques and build better systems in the future.

D.6 Tutorial Section

Unlike the paper prototype experiment, the final usability study had a written tutorial that the participant would first complete before attempting the tasks. The complete text of the tutorial follows.

Gamut Tutorial

Welcome to Gamut. With this tool, you can make games simply by showing the computer how the game works. To make a game, you need to first draw what the game looks like, and then you *demonstrate* how each part of the game works by directly modifying the game's graphics. In this study, we will provide most of the graphics for creating a new game. It's your job to make the game work.

This tutorial will show you how to use part of Gamut. First, we will show how to draw graphics and widgets on the screen. Then, we show the basics for demonstrating a new game behavior. Finally, we will show some techniques that will make demonstrating some behaviors much easier.

How To Use This Tutorial

First, read the introductory text to each section. This will tell you the basic concepts that you will be taught in that part. At the end of each section, there is a set of bulleted instructions that you must do. The results of these instructions feed into the instructions of following sections; so, please make sure you finish the instructions of one section before moving to the following section.

- **This is an instruction. Relax, have fun, and make sure you ask if you have any questions.**

Drawing Stuff

To draw an object, select the object from the palette. Then click and drag the mouse to create the object where you want it. After you draw the object, the palette will revert back to selection mode.

Note: The big blue frame in the middle of the drawing area is the window for your game. Everything you draw inside the blue frame will be visible in your game and everything drawn outside is hidden when the game is played.

In order to change an object's color or edit an object in other ways, you must select the object. To select an object click on it with the mouse. To select

multiple objects, hold down the *shift* key and click on the other objects in the set. You can also “lasso” multiple objects by clicking down the mouse in a vacant part of the window and dragging the dotted rectangle around all the selected objects.

- **Draw a rectangle and a circle. (Make their size about an inch across.)**
- **Select them both and duplicate them. (The “Duplicate” command is in the “Edit” menu.)**
- **Delete one of the rectangles and both of the two circles using “Cut”.**

To change an object’s color use the “color selector” that is in the toolbar region of the window. It is located near the top of the window and looks like the image in Figure 1. There are two things that look like the color selector in

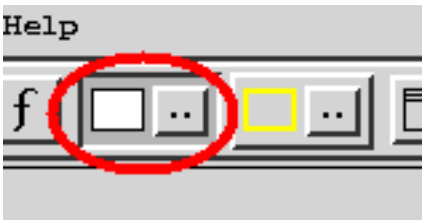


Figure 1: The color selector. This shows the color Gamut uses to draw new objects. Push the “..” button to change the color.

the toolbar. The one you want is on the left. The color selector has two parts. The rectangle in the middle shows what color Gamut will use when new objects are created. There is also a button that brings up a dialog box which you can use to change the currently selected color.

- **Change the rectangle’s fill color to red.**

When you are building a game, there are often a lot of features in the game that affect the game’s behavior but should not be visible to the player when the game is running. You can build these “invisible” objects by making them “guide objects.” Guide objects are just like regular objects except they use a color from the “guide object color selector” instead of the regular selector. The guide object color selector is on the right of the regular color selector

and works the same way. All guide objects can be made invisible using the "Hide Guide Objects" command in the "View" menu.

Note: When Gamut creates a new object it can only use one of the two kinds of color. You can tell whether Gamut will use the regular color or the guide object color because the region around the indicator's box will look pressed. You can change which color is used by clicking on the indicator.

- **Make a path out of five arrow lines. Connect the start of one line to the tail of the next as in Figure 2. (Draw the arrows inside the blue window frame because you will need to have it this way later.)**

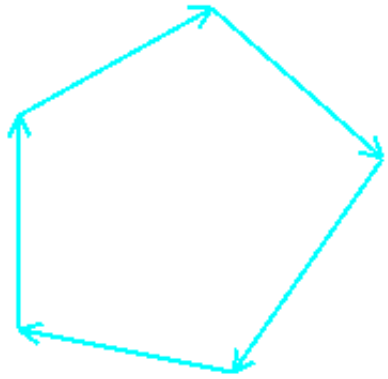


Figure 2: Draw a path of arrow lines like this.

- **Set the color of the arrow lines to be cyan "guide objects."**
- **Move the rectangle so that its center is positioned at the end of an arrow line. You may need to resize the rectangle so that you can place its center exactly at the end of an arrow.**
- **Use the "Hide Guide Objects" command in the "View" menu to make the guides invisible.**
- **Use "Show Guide Objects" to make them visible again.**

Finally, Gamut has a number of "widget" objects like buttons and check-boxes. Some widgets, like buttons, have a default size so no matter how big you draw the region, they will always be the same size when you create one.

You cannot set a widget's color so you cannot turn one into a "guide object." However, many widgets have properties that you can change. To change a widget's properties you can double-click the mouse over the object. You can also select the object and use the "Edit Properties" command in the "Edit" menu.

Selecting a widget can be a little tricky. For instance, a button can be pushed as well as be selected. Clicking in the center of a button pushes it, clicking on the edge of a button selects it. You can also "lasso" the button in order to select it. You will have to unselect the button in order to push it again.

- **Create a button. (Its palette icon looks like**  **.) Make the label of the button read "Move."**

Demonstrating Stuff

You can teach Gamut new behavior by "demonstrating" your game's responses in the editor. In order to teach a new behavior, though, you have to get Gamut's attention. Gamut normally does not watch what you are doing while you draw on the screen. To get Gamut's attention, you use the "Do Something" or "Stop That" buttons at the bottom right corner of the window which look like the image in Figure 3. As your first example, you will

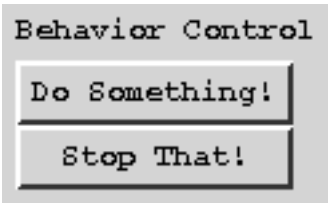


Figure 3: The Do Something and Stop That buttons. Use these to demonstrate new behavior.

make the rectangle that you created previously follow the arrow lines when you push the "Move" button.

The first thing you need to do is perform the action that causes the intended behavior to activate. In this case, you would push the "Move" button because that is what tells the rectangle to move. Then, when you see that Gamut

does not perform the right behavior, you press the “Do Something” or “Stop That” button to tell Gamut what it should have done.

- **Push your “Move” button. (That’s the button you created above.) If the “Move” button happens to be selected, you will have to unselect it in order to push it.**
- **Push “Do Something.”**
- **Move the rectangle to the next link in the path. Make sure it is centered correctly because Gamut can get confused if the picture is messy.**

We haven’t quite finished telling Gamut what to do. First, it is a good idea to show Gamut what objects are important to the intended behavior. In this case, we want Gamut to notice the path that the rectangle is following. Gamut can be a little dense sometimes and not see very obvious relationships (like when two objects are connected). You help Gamut see these relationships by *highlighting* important objects. You highlight an object by clicking on it with the rightmost mouse button.

- **Highlight the line connecting the rectangle to its original position. The screen should now look something like Figure 4.**
- **Push the “Done” button.**

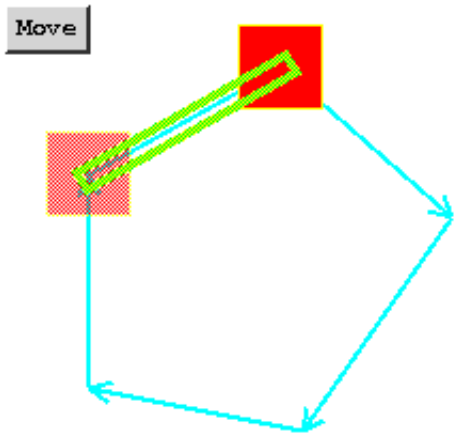


Figure 4: This is what the area near the rectangle should look like right before you press the “Done” button and finished demonstrating.

Now, when you push the “Move” button, the rectangle should follow the path just like you intended. Note: Gamut is not smart enough to learn to move the rectangle smoothly, yet. Do not be worried that objects tend to jump from one location to the next.

You can make more behaviors by continuing to show Gamut more examples. Let’s make another behavior that changes the color of the rectangle back and forth between red and blue.

- **Make a new button and label it “Color.”**
- **Push the “Color” button.**
- **Push “Do Something.”**
- **Change the color of the rectangle to blue (keeping the line style the same).**
- **Push “Done.”**

At this point, we have a behavior that changes the color of the rectangle to blue, but that’s all. Now when you push the “Color” button you’d want the color to change back to red.

- **Push the “Color” button. (Nothing seems to happen.)**
- **Push “Do Something.”**
- **Change the rectangle’s color back to red.**
- **Push “Done.”**

Now, Gamut has seen you do the same thing twice, you pushed the “Color” button each time but the behavior had different results. Now, Gamut wants

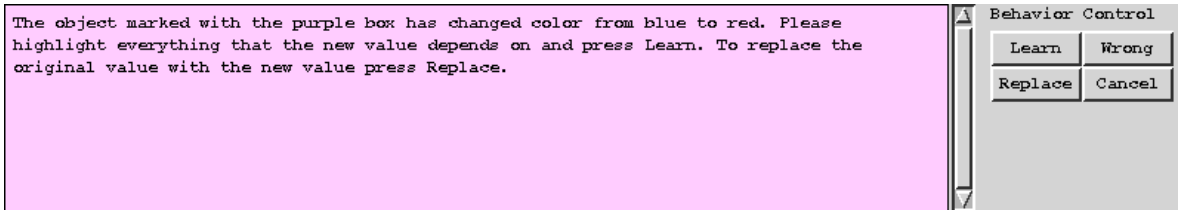


Figure 5: The dialog that Gamut uses to ask you questions. The buttons you use to answer the questions are at the right.

to know what the different colors depend on and it asks you with the dialog box at the bottom of the screen which looks like Figure 5. Alongside the text are four buttons labelled “Learn,” “Replace,” “Wrong,” and “Cancel.” **Understanding these buttons is extremely important.** The buttons are used to respond to Gamut’s questions so it is important that you know what you are telling the system. Here is what these buttons are used for:

Learn: This is the typical button you use. First you highlight the object(s) in the game that answer Gamut’s question and then you press this button to make Gamut “Learn” how to use whatever you highlighted.

Replace: This button is used to fix mistakes. You may have accidentally changed the game in the wrong way while Gamut was watching. When you push this button, Gamut will “Replace” whatever behavior it used to do with the new behavior.

Wrong: This button is used to skip dialogs that do not apply to the behavior you want to create. Make sure you know what the dialog text is asking before you dismiss it, though.

Cancel: Sometimes you will realize that you are not ready to demonstrate the behavior you just started. You can press “Cancel” to take you back to the normal editing mode without creating a new behavior.

In this case, we have not made a mistake and Gamut has not asked an unreasonable question, so we must highlight an object in the scene that tells Gamut when to make the color red or blue. The object you want to highlight is the faded “ghost object” of the rectangle which shows what the rectangle looked like before you changed its color. The ghost object looks like the image in Figure 6.

- **Highlight the “ghost object” of the rectangle.**
- **Press “Learn”**

Now that Gamut knows that the original properties of the rectangle are important, it creates the correct behavior. When you push the “Color” button, Gamut will switch the color back and forth as you intended. Of course, sometimes Gamut will not know which properties of the highlighted objects you want the behavior to use. In these cases, you will have to give Gamut more examples to sort out which properties are more important than others.

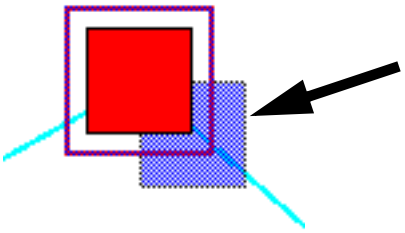


Figure 6: This shows the rectangle along with its “ghost object” which is the blue faded rectangle that is offset to the lower right from the original.

Knowing which object to highlight can sometimes be tricky. Let’s augment the “Move” button’s behavior so that it will move multiple rectangles.

- **Create a duplicate of the rectangle and place its center on the end of one of the arrows.**
- **Press the “Move” button. Here the original rectangle should have moved but not the new one.**
- **Use “Do Something” to demonstrate that the other rectangle should move as well. Press “Done” when you finish.**

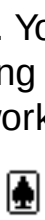
At this point, Gamut will want to know what the two rectangles depend on. Essentially, you want to move all the rectangles in the window, so it is the window object that describes both of the rectangles. You might not have thought that the window could be highlighted. You can highlight the blue window frame by clicking with the right mouse button near the inside edge of the frame.

- **Highlight the window frame.**
- **Press “Learn”**

At this point, you might be wondering what the “Stop That” button does. We’ll use it in the next example, but first we will create some new objects that we can use to demonstrate more behaviors.

Cards and Decks

Just like you can draw graphics and widgets in the main window, you can also draw stuff into “cards.” Cards are like extra drawing areas where you can put stuff that relates to an object in your game. You can get at the objects inside a card by double-clicking on it or using the “Edit Properties” command. This will bring up a special editor that works just for that card.

- **Create a card. (Its palette icon looks like  .)**
- **Open up the contents of the card using the card editor.**
- **Draw a text object inside the card (the icon looks like ) and make it read “Color” using a large, bold font.**

You’ll notice that there is a raised region in the card editor. This is the card’s “visible area.” The visible area is like a window: everything you draw in the visible area will be seen in the card object itself. You can select and move the visible region around if you want.

Stuff that you draw inside a card is difficult to get at when you are demonstrating behavior, so Gamut gives you a shortcut that lets you pre-highlight objects inside a card.

- **Pre-highlight the “Color” label by selecting it and using the “Highlight Item” command in the card editor’s “Edit” menu.**

Your card should look like the image in Figure 7. Now, whenever you highlight the card in the main window, the label in the card will also be highlighted.

- **Close the card editor for your new card. (Use the “Close” command in the card editor’s “Card” menu.)**
- **Select the card in the main window and make a duplicate of it.**
- **Open the new card’s editor and change its label to read “Move.”**
- **Close the card editor for the “Move” card.**

You now have two cards: one labelled “Color” and the other “Move.” Now, let’s make a deck out of these cards. The deck object holds a list of objects.

Figure 7: The “Color” label is placed in the visible area to make it visible on the card and it has a mark around it to show it is “pre-highlighted.”

To put an object in a deck, select the object and drag it over the top of the deck. When the deck accepts the object, it will show a purple outline around itself. To take an object out of the deck, select it and drag it away.

- **Create a deck. (Its palette icon looks like .)**
- **Drag your two cards into the deck.**

You will notice there are four buttons along the bottom of the deck object. The first button, , is the deck’s “lid.” When the lid is closed, the deck is locked and you cannot put objects in or take objects out of the deck. The second button labelled “shuffle” will shuffle the contents of the deck to randomize their order. The other two buttons, , are used to look at objects in the deck without changing their order.

- **Close the lid on the deck and move it to the side of the window frame’s region. The main window should now look something like the image in Figure 8.**

Okay, now we are ready to demonstrate more behaviors. In this next behavior, we will use the behaviors we demonstrated for the two buttons and let the deck choose which one to apply at random. When the deck shows the “Move” card, the rectangles should move. When the deck shows “Color” the one rectangle’s color will switch.

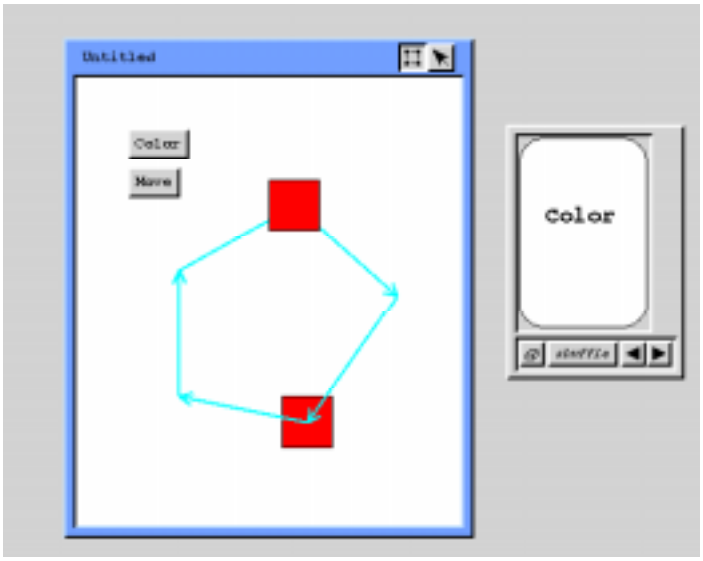


Figure 8: The main window with the assembled deck.

- **Create a new button, label it “Combined.”**
- **Push “Combined.”**
- **Push “Do Something.”**
- **Push the “shuffle” button on the deck.**

This shows that you want to deck to shuffle each time you push the “Combined” button. At this point, what you do depends on what card comes up on top.

- **If the “Move” card is on top, push the “Move” button. Otherwise push the “Color” button.**
- **Push “Done.”**

Now that your new behavior is partially defined, pressing the “Combined” button will keep moving or recoloring the rectangles depending on the random shuffle. Since you never demonstrated the other action, though, Gamut will not know what to do when the other card shows up.

- **Keep pushing the “Combined” button until the opposite card shows up.**

Finally, here is a case where you need to use “Stop That” instead of “Do Something.” You use the “Stop That” button when Gamut does something

wrong like choosing the wrong behavior for the rectangles. Use “Stop That” to tell Gamut to stop performing whatever behavior it did to one or more objects. In this case, you select the rectangles and push “Stop That.” Gamut will undo whatever behavior it performed on those objects. Sometimes people will forget to select the object they want to have stopped. A handy way to remember is to say “You, Stop That” and when you say “You,” you select the misbehaving object.

- **Select the rectangle(s) that you want stopped and push the “Stop That” button.**
- **Push the button for the behavior you really wanted to have happen and push “Done.”**

Gamut will want to know why the behavior has changed. In this case, the value of the top card on the deck is what is important.

- **Highlight the deck.**

Because you already pre-highlighted the label inside each card in the deck, Gamut will automatically know that it is the label that is important. Otherwise, you would have to open up the card editor and highlight the label manually. (Gamut is not smart enough to dig around inside your cards to see what is interesting.)

- **Press “Learn.”**

It turns out, Gamut will pick a reasonable rule out of three objects you highlighted when you highlighted the deck (the top card and the text object are also highlighted). When you push the “Combined” button, the behavior should work as you intended. Gamut is not usually so easily persuaded to perform the right behavior. You should test anything you demonstrate until Gamut either makes a mistake or you are satisfied that it works. Extra examples will help Gamut sort out the properties of the highlighted objects. While Gamut sorts out different possibilities, it will accept new examples without asking you to highlight things.

Controlling History

There is one more interaction you should learn how to use. In the toolbar, the four buttons furthest to the right are the “time control.” They look like the image in Figure 9, and they are used to step back and forth in your game’s

Figure 9: The time control buttons. The buttons are: back, stop, forward, and pause.

history and are also used to control timers. You will not need to use a timer in your current tasks so we will skip the “Stop” and “Pause” buttons (labelled ■ and ❚❚) for now. Just so you know, they are used as global stop and pause buttons that would affect all timers in your application.

Starting from the left, the first button is “back” which is labelled with ◀◀ . This button sends your game backward one step in time. The other button that you will want to use is labelled ▶▶ , and is the button for moving forward in time. You might think of the back and forward buttons as acting similar to undo and redo except that if you have demonstrated a new behavior during one of the steps, the behavior will still be there after you have stepped back in time over it. Using the back button is a good way to go back to a previous state in order to test what you just demonstrated or to demonstrate a slightly different example.

- **Practice using the “forward” and “back” buttons to see how it affects your game’s objects.**

That completes the tutorial example. Before you go, make sure that you save your work.

- **Use the “Save” command in the “File” menu to save your work. Name the file “path.gam”.**

When You Have Trouble

Unfortunately, Gamut is just a prototype system. You can make a lot of good games using this system, but sometimes Gamut can fail. For some reason, you might run into a bug in Gamut which won't let you finish demonstrating a behavior. You can use these techniques to get yourself out of a mess.

Undo: First, you should realize that undo exists in Gamut. The "Undo" command is in the "Edit" menu. For instance, if you press the "Done" button but you forgot to highlight something, you can perform undo, highlight what you want, and hit "Done" again. Note that this is not the same as using the back button in the time control which doesn't remove behaviors.

Delete Behavior: At some point, you may completely lose track of what you were demonstrating. Sometimes it can be easier to start over than to try to correct a behavior to make it work. To delete a behavior, select the object that causes the behavior to occur. This object will likely be the button or timer that you would activate to make the behavior happen. Then select the "Delete Behavior" command in the "Behavior" menu. Gamut will erase the behavior from that object and you can start over again.

Starting Over: If Gamut crashes or you are really having trouble building one of the trial games, you can always start over. The first part of each experiment will ask you to load a file as your starting point. You can restart by reloading that file. You can also save your progress if you get the game partially working and start over from that point. Please do not feel as if you have to keep going if you get lost or Gamut starts acting very poorly. Sometimes a fresh start will clear away the cobwebs and you can make the game much more easily the second time. We do ask that if you start a trial over again that you save the work that you are abandoning. This information will help us fix the problems you encountered and might help make Gamut a better tool.

Let's Begin!

Thanks for helping us with this study. When you feel comfortable with the material in this tutorial, we will be ready to begin.

D.7 Review Questions

After finishing the tutorial, the participant would be asked the following review questions. Answers were given orally.

Review Questions

1. **How do you demonstrate a new behavior?**

2. **How do you fix a behavior that's not doing the right thing?**

3. **How do you add new actions to a behavior that already exists?**

4. **What is the difference between highlighting an object and selecting an object?**

5. **What is the difference between using the Learn and Replace buttons when Gamut asks you a question?**

6. **What should you select before you push the Stop That button?**

7. **When Gamut asks a question, what is the most likely way it should be answered?**

D.8 Extra Task Tutorial

One participant was asked to perform the third task that required her to use some techniques that were not discussed in the main tutorial. The following "Extra Tutorial" discussed these techniques that included the player mouse icons and timers.

Extra Task Tutorial

In order to implement the last task, you must know about a couple more of Gamut's features. These features are not at all difficult, but they aren't really needed to complete the first two tasks, so we saved introducing them until now. The last task has two elements that are not found in the previous two. First, it contains a "monster"-like object that moves around on its own. You'll need to learn about timers to make that work. Also, the player uses the mouse to click on the screen to tell the game which way to move. This requires that you use the player input icons. This part of the tutorial will discuss these two techniques.

Timers

The timer is used to create behavior that automatically repeats at a fixed interval.

- **Open the file "path.gam". This is the file you saved earlier during the first tutorial.**
- **Create a timer widget. (The palette item looks like: )**
- **Set the timer to tick every 500 milliseconds. (Hint: it's a property.)**

The timer widget has a set of three buttons below the picture of a clock. The first button is the "step" button which has this icon  . When you push this button, the timer ticks exactly one time. You use this button to demonstrate what the timer does for your game. The other two buttons which look like  are the "stop" and "play" button which turn the timer off and on.

Let's transfer the behavior of the "Combined" button you demonstrated earlier into the timer.

- **Press the "step" button on the timer.**
- **Press "Do Something."**

- **Press the “Combined” button.**
- **Press “Done.”**

Now, when you press the timer’s “step” button, it will shuffle the deck and do all the stuff you demonstrated for the “Combined” button. If you press the “play” button everything will happen automatically. Just for fun, while the timer is running, you can create more rectangles and add them to the path and they will move like the others.

Stop and Pause

In the previous tutorial, we skipped discussing the stop and pause buttons because they only affect timers. Now that you know about timers, you should also know about the rest of the history controls. Recall that the history control buttons are in the toolbar and look like the image in Figure 9.

Figure 1: The time control buttons. The buttons are: back, stop, forward, and pause.

They are used to step back and forth in your game’s history as well as pause and stop timers. We mentioned the “back” and “forward” buttons (and) in the last tutorial. The “stop” button is labelled . This is a global stop button that will stop all timers. You may want to turn on the timer you demonstrated above and then stop it by pushing the global stop button. The last button is “pause” which is labelled with . The pause button is like stop in that it turns off timers except that the timers remain active. In other words, they are still “on” but they do not tick. If you turn off pause, the timers that were previously active will continue to run. Stop will just turn the timers off outright.

When you use the back button and go to a previous state, Gamut will automatically turn on pause. This is necessary because if a timer is running when you back up, it will mess up the state of your game and you would not

be able to look at the history. Because Gamut sometimes activates pause on its own, it is easy to forget to turn off pause. Gamut will flash a box around the pause button whenever you push the play button on a timer while pause is active. If you do not mean for the game to be paused, you can deactivate it by pushing the pause button again.

The previous example is no longer needed and its graphics will probably be in the way for the next example. Since you already saved this example during the last tutorial we can just save into the same file. Then we'll clear some space for the new example.

- **Use the "Save" command in the "File" menu to save your modifications in the original file.**
- **Use the "New" command in the "File" menu to begin a new game.**

Player Input

Some games allow the player to use the mouse to manipulate the game. To demonstrate player input, you use the "mouse icon palette" (see Figure 2) that sits right below the main object palette. Mouse icons are created similarly to the way you would create graphical objects. First, you select the icon you want to drop and then click on the position in the window where you want to drop it. Gamut interprets a dropped mouse icon as though the player has just performed the corresponding event. If you drop a mouse click icon, Gamut reacts as though the player has clicked the mouse. There are also icons for performing mouse down, drag, and up events, as well as double down and double click, but we will only use the mouse click icon which looks like this  . You can only drop mouse icons into the blue frame window and not into the surrounding areas. That's because the player can only click on the window and cannot see any of the offscreen area.

Let's make a behavior that creates a yellow circle wherever the player clicks the mouse. Let's also make it so that when the player clicks on a circle that already exists, Gamut will delete that circle (instead of creating one).

Figure 2: This palette contains all of the player input icons. The “click” icons have an arrow head pointing both up and down.

- **Drop a click icon into the window. (Its palette icon looks like .)**
- **Press “Do Something.”**
- **Create a yellow circle and place its center at the tip of the click icon.**
- **Highlight the click icon (to indicate that its position is important).**
- **Press “Done.”**

This is the first time we’ve demonstrated creating a new object. Gamut marks created objects with a big “C” in the center. The C will remain attached to the object until you perform another event. You can test that your new behavior works by dropping more click icons in the window. You will see a C inside each new object as it is created.

Now, let’s demonstrate the deleting part of this behavior.

- **Drop a click icon onto an existing circle. (Try not to put it in the center because Gamut might think that the alignment is intentional.)**
- **Select the newly created circle and press “Stop That.”**
- **Delete the circle that the click icon is over.**
- **Press “Done.”**

When you delete an object, Gamut draws a “D” in the place where the deleted object used to be. The D is important because this is what you would select if you need to use Stop That on a deleted object.

At this point, Gamut will want to know why it has changed from creating an object to deleting one. The reason is because the player clicked on a circle.

- **Highlight the click icon and the ghost of the deleted object.**
- **Press “Learn.”**

Here is where the time control really helps. We can use the time control to bring back the deleted object so that we can see if Gamut really knows which object it is supposed to delete.

- **Press the history controller’s back button. (Its image is  .)**
- **Drop a click icon onto a circle but not the one you deleted earlier.**

As you can see from the big D and the fact that the wrong circle is gone, Gamut currently thinks it’s always supposed to delete that specific object. We can fix this using Stop That.

- **Select the “D” of the deleted circle and press “Stop That.”**
- **Delete the circle that was supposed to be deleted and press “Done.”**

Gamut will want to know how to pick the right object to delete.

- **Highlight the click icon and press “Learn.”**

Now your clicking behavior should work correctly. Test it by dropping click icons around the window. You can also generate player input directly through the regular mouse as well as with the icons. To activate “direct input mode” press the  button on the frame of the blue window. Of course, you can’t select objects in the window frame while the direct input mode is active so to go back to “selection mode” press the  button.

Save your work as “click.gam”.

Okay, that’s it. You are now ready for the final task.

D.9 Wall Poster

A poster was placed on the wall in front of the participants to remind them how to demonstrate things in Gamut. The poster was made from a single sheet of 8.5 x 11 inch paper and had the following contents.

To Create New Behavior

- **Perform the event that causes the behavior to occur.**
- **Push "Do Something."**
- **Modify the objects to look the way they were supposed to.**
- **Press "Done."**
- **Answer Gamut's questions by highlighting appropriate objects and pressing "Learn."**

To Modify a Behavior

- **Perform the event that causes the behavior to occur.**
- **Select the objects that did not do the right thing.**
- **Push "Stop That."**
- **Modify the objects to look the way they were supposed to.**
- **Press "Done."**
- **Answer Gamut's questions by highlighting appropriate objects and pressing "Learn."**

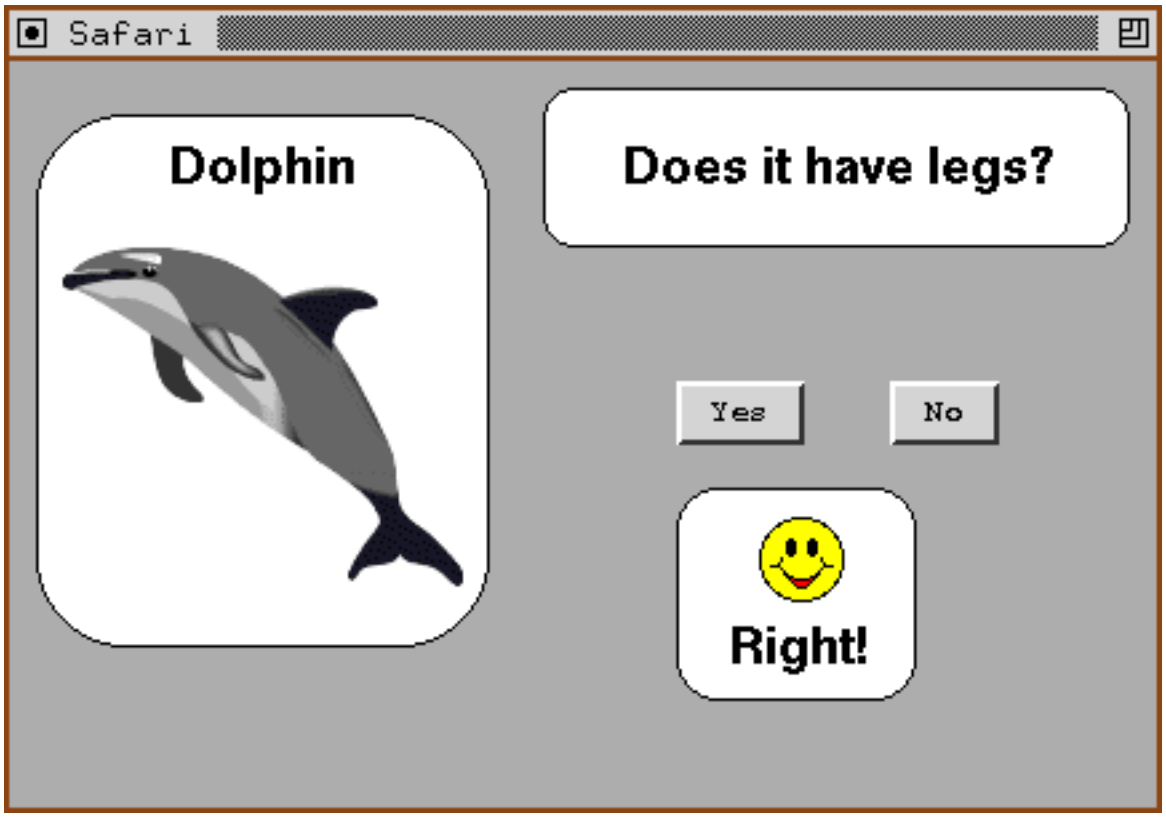
D.10 Final Usability Tasks

There were three potential tasks that could be given to the participants. All participants received the first two tasks. One participant received all three tasks as well as the Extra Tutorial. This section lists the instructions for each task.

D.10.1 Task One: Safari

Safari

Safari is a small educational program to quiz children about animals. The game consists of two decks of cards: one contains a set of five animals, the other is a list of questions like “Does it have stripes?” The child is supposed to answer the question with the yes and no buttons below the question deck. If the player is right, the smiley face appears to the right. If not, the player gets the dreaded frowny face.



To get you started, we have begun preparing all the cards for you. The animal deck contains four cards. Each has a label with the animal’s name and a picture of the animal. The question deck has four different question cards. There is also a “response” deck which contains the smiley and frowny face. Note that you don’t have to shuffle the response deck. You can use the arrow keys to pick the right response directly. The main thing you will have to

do for this task is augment the animal and question cards so that Gamut doesn't have to learn everything for itself.

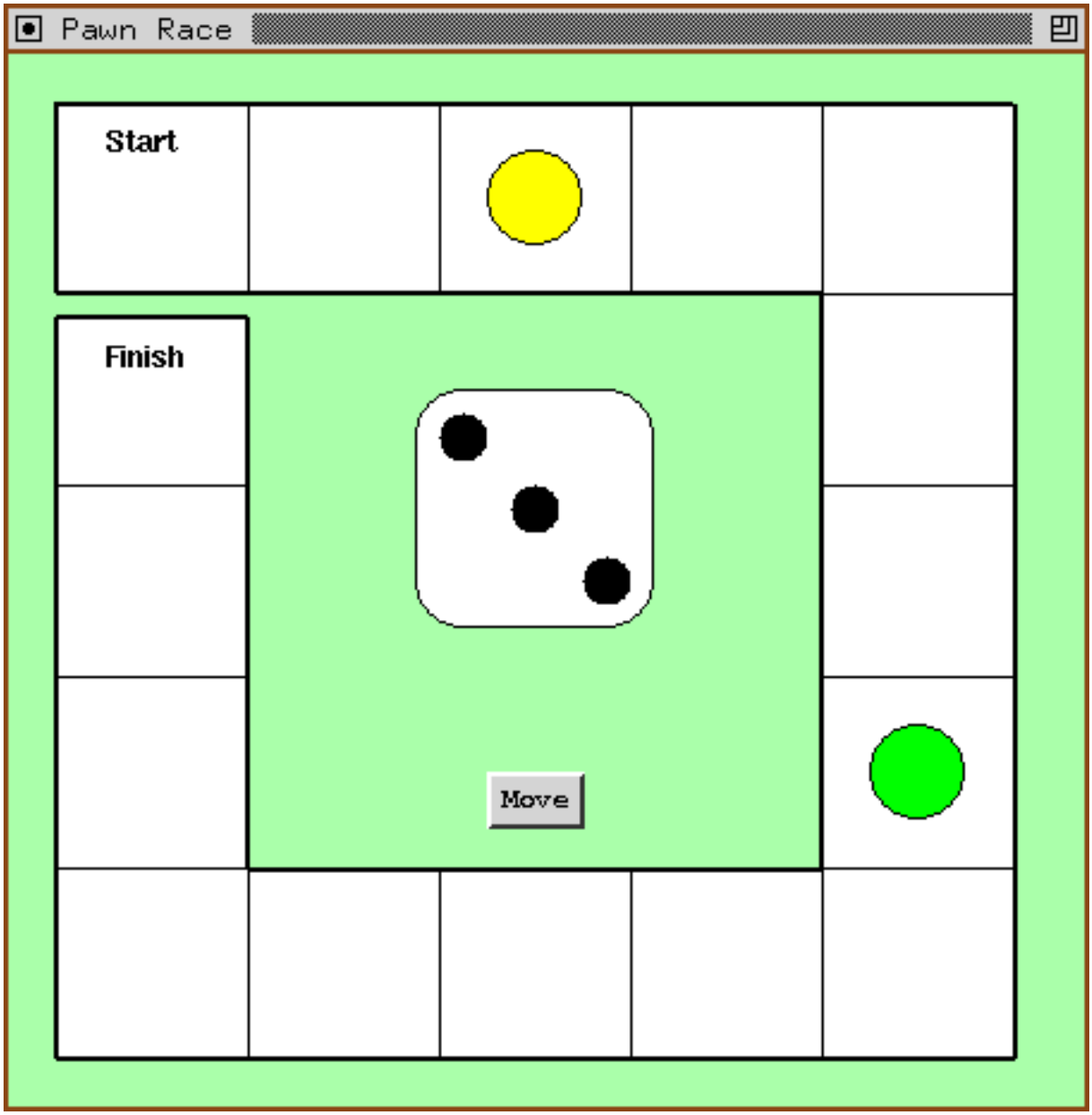
Gamut is just a prototype system, so it only has a limited range of properties that it can detect. It doesn't know anything about animals and it certainly cannot read. Also, Gamut can only match the whole text in a text object and cannot pick out words or phrases. For instance, Gamut cannot see the word "legs" in the sentence, "Does it have legs?" Similarly, the case and spacing of words matter: "Spider" does not match "spider." Thus, you must be careful not to overestimate what Gamut can see when you put objects into the cards. You may want to ask the experimenter if you feel that Gamut might have trouble detecting the rule you want it to learn.

- **Open the file "safari.gam". This file contains the starting set of cards and buttons. You may save your work in this file, too.**
- **Add objects to the cards in the animal and question deck so that Gamut has enough information to know what the right response should be for each question. Hint: whatever you draw in one deck, make sure you make a matching object in the other deck.**
- **"Pre-highlight" the objects you created in the previous step. You should be pre-highlighting objects in both decks.**
- **Train the "Yes" button to shuffle the animal and question decks and show the frowny face when the player answers a question wrong and show the smiley face when right.**
- **Train the "No" button in the same way as the previous step (except the answers are reversed).**
- **When you finish, save your work in the file, "safari.gam".**

D.10.2 Task Two: Pawn Race

Pawn Race

The Pawn Race is a simple game for two players. Each player takes turns and rolls a die to move his or her piece that number of spaces. The first player to reach the end is the winner. To finish the game, the player must roll exactly the number of spaces to the end. If a player lands his or her piece on the other player's piece, the other player must start over from the beginning.



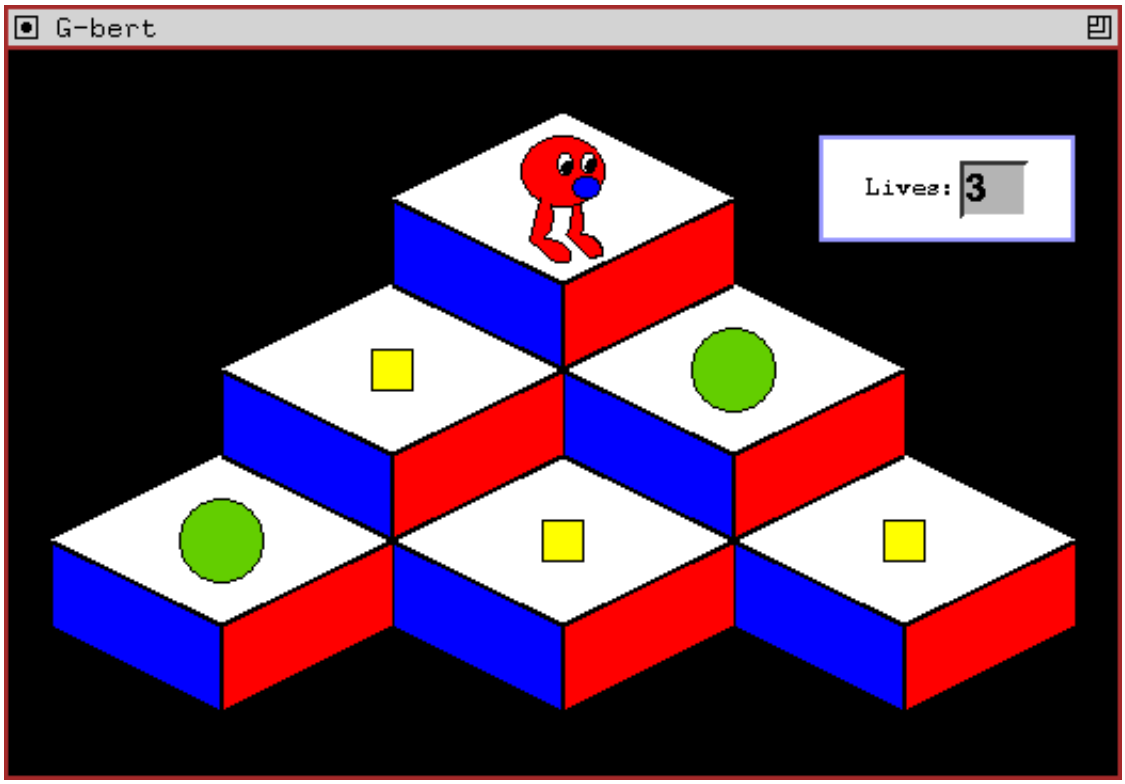
To start you off, we have already assembled the game board for you. You will have to make the pieces yourself, as well as demonstrate the game's entire behavior. Please note that although the board looks like a circle of squares, it is actually composed of lots of little pieces and Gamut cannot see the individual squares unless you do something to point them out. You will need to draw something on the board to show Gamut how the pieces move. You will also need to create objects offscreen to keep track of the game's state. The die in the center is actually only "three sided." That is, it has numbers one, two, and three and that's it. This is intentional so don't worry about it.

- **Open the file "pawn.gam". This has the start-up components for your task. Please feel free to save your work in this file.**
- **Create the pieces for the players. Make a green piece and a yellow piece using circles. Place both pieces in the center of the starting square directly one on top of the other.**
- **Draw something on the board to show the path that the pieces will follow.**
- **Make it so that pushing the "Move" button causes the die to roll (shuffle the deck) and moves a piece the corresponding number of places.**
- **Create an object to keep track of the current player's turn and make the "Move" button alternate between players.**
- **Demonstrate that when either player's piece lands on the other, the other player's piece goes to the beginning.**
- **Create an text object labelled "Winner!" and mark a position on the board where you would want it to go. Place the Winner! object offscreen.**
- **Make the Winner! label move to the marker when one of the pawns reaches the last position.**
- **When you finish, save your work in the file "pawn.gam".**

D.10.3 Task Three: G-bert

G-bert

In this task, you will make a simplified game similar in some ways to the game “Q-bert.” The background consists of a pyramid of blocks. On top of each block is a marker. The player moves G-bert to each cube and collects the markers. Balls drop down from the top of the pyramid and randomly fall down the blocks. If a ball touches G-bert, the character begins again at the top block and loses a life. The player moves by clicking on the top of the block where G-bert should move. G-bert only moves to a block if it begins on an adjacent cube and can move both upward and down. The player wins when all of the markers are collected.



We provide the background and a bitmap image for G-bert. You will be demonstrating G-bert’s behavior as well as the behavior of the bouncing balls. When you demonstrate objects that move, remember that it’s okay if the movement is jumpy. Gamut isn’t set up to let you demonstrate smooth motion. The characters will just jump from one point to the next. Also, the background we provide for you is not complete. You will have to augment it

so that Gamut will know where the player is allowed to click the mouse as well as how G-bert and the balls move.

The player is supposed to click on the top of a cube to show where G-bert moves. Since Gamut doesn't support a polygon object, you can't draw a guide object that exactly fits the top of the cube. It is okay to use rectangles, instead.

- **Open the file "gbert.gam". This file includes all the needed starting components. This is a longer task so make sure you save your work in phases. If you want to save multiple files, just make sure they all begin with the word "gbert" (gbert1.gam, gbert2.gam, etc).**
- **Demonstrate how G-bert moves. Use the click icon to show how the player clicks on the top of a cube. You will need to draw something to show where the cubes are located and how the board is connected.**
- **Create yellow rectangles to serve as markers as put them on the board.**
- **Demonstrate how the markers are collected when G-bert lands on them.**
- **Create a green circle to be the bouncing ball.**
- **Demonstrate how the ball travels randomly down the pyramid. You will need to use several widgets and draw guides on the board to support this. To make sure that the ball keeps coming down from the top make the ball's path loop around to the top.**
- **Demonstrate that when the ball hits G-bert, G-bert loses a life and goes back to the top of the pyramid.**
- **Demonstrate that the game is won when G-bert collects all the markers. A big sign saying "You Win!" would be fine.**
- **Demonstrate the losing condition when G-bert runs out of lives. A sign saying "You Lose..." would work.**
- **Save your game in the "gbert.gam" file.**

Appendix E: Usability Experiment Results

This appendix lists the results of the final usability experiment. The results consist of the answers to the surveys and the drawings that the participants made in each task. Participants also drew objects into the cards in task one. There were four participants in the experiment number one through four. The second participant answered some forms twice. These are labelled as 2a and 2b.

E.1 Form Results

The final usability experiment used two forms: the pre-test survey and the post-test questionnaire. In this section, each participant's responses are listed.

E.1.1 Survey

Subject No.: _____ Age: _____ Sex: _____ Male _____ Female
Age: (1) 47, (2) 21, (3) 19, (4) 21
Sex: (1) Male, (2) Female, (3) Male, (4) Female

Do you own a computer? _____ YES _____ NO
(1) Yes, (2) Yes, (3) Yes, (4) Yes

What do you use computers for?:
(1) Work - Word Processing
Spread Sheet
Database
Calendar

(2) checking my email
wasting large quantities of time on zephyr
wasting small quantities of time playing games

(3) Word processing, HW, games, internet surfing

(4) Work
Word Processing, statistics, e-mail, internet etc.

Do you use a computer in your work? _____ YES _____ NO
(1) Yes, (2) Yes, (3) Yes, (4) Yes

Can you program a computer? _____ YES _____ NO
(1) Yes, (2) No, (3) Yes, (4) No

If so, please explain (Where do you program computers? What language(s) do you use?):
(1) Prolog - very rusty

(2) --

(3) took programming class 125

(4) --

Do you use a computer to play games? _____ YES _____ NO
(1) No, (2) Yes, (3) Yes, (4) Yes

If so, what games do you like to play?
(1) --

(2) Diablo
Lords of Magic
Tetris
Myst

(3) RPGs & action games and thinking games (ex. Myst)

(4) Solitaire

Do you play non-computer games like board games or puzzles? _____ YES _____ NO
(1) No, (2) Yes, (3) Yes, (4) Yes

If so, what non-computer games do you like to play?
(1) --

(2) Mancala
Spades
Most card games

(3) chess; and just about anything else

(4) Puzzles, monopoly, life, scrabble

If it were easy enough to do, would you write your own computer games?
_____ YES _____ NO
(1) Yes, (2) Yes, (3) Yes, (4) Yes

If so, would you want to (check as many as you like):
_____ Design original games.
(1) Yes, (2) Yes, (3) Yes, (4) Yes

_____ Port non-computer games to the computer.
(1) No, (2) Yes, (3) Yes, (4) No

_____ Other. Please specify:
(1) No, (2) No, (3) No, (4) No

E.1.2 Questionnaire

Participant two filled out the questionnaire twice, once for each session. The questionnaire for the first session (tasks one and two) is labelled 2a and the other session is labelled 2b.

Which task did you find the easiest to do? What made the task easy?
(1) Using the aspects of the interface that reminds me most of other things I have already used.

(2a) 2nd one (moving dots pawn) -> It was more straightforward

(2b) Stupid card game - fewer things to manipulate

(3) Creating buttons and arrows; circles, squares, etc. Button in the toolbar

(4) The card task because it was fairly repetitive.

Which task did you find the most difficult? Why was it difficult?
(1) Getting items to perform together. I didn't pay enough attention to what was already caused in the manual.

(2a) 1st one (cards) -> The answer [{smiley face} or {frowny face}] was based on something you couldn't see @ the same time as the question.

(2b) G.Bert - most complicated
- couldn't get Gamut to grok deleting + decrementing blocks

(3) Training the pieces to start from the beginning when on same square, Computer just wouldn't learn it!

(4) Picking a tracker for pawn because I didn't know all available options & limited my thinking.

Did you experience problems with any of the following:
1. Understanding how to carry out the tasks (check one):

_____ no problems _____ minor problems _____ major problems
(1) Major, (2a) Minor, (2b) Minor, (3) Minor, (4) Minor

Please explain:
(1) Some people are more visual than others. I think a visual "walk-tru" would have been more helpful for me.

(2a) on some things I drew a blank after reading the instructions. I think it's because I wasn't quite used to what each thing was called.

(2b) I'm not horribly familiar w/ this (obviously) so it wasn't always clear what to do to use for what step/job/etc.

(3) Just needed more time to practice & play around w/ it; bugs; the interface was useful & user friendly

(4) I doubted myself & automatically assumed I couldn't do it but once started it went well

2. Knowing what to do next (check one):

_____ no problems _____ minor problems _____ major problems
(1) Major, (2a) No problems, (2b) No problems, (3) Minor, (4) Minor

Please explain:
(1) The examples didn't reinforce well-enough what I was supposed to do to get a good response. I forgot very quickly.

(2a) --

(2b) It was generally clear when to do what. what to do was the fuzzy part.

(3) Same as previous question

(4) Deciding between replace & learn

What are the best aspects of Gamut for the user?
(1) The tasks that build ups what a user already is failing with.

(2a) It's ability to learn based on selecting what it should notice

(2b) It's visual, so you can tell what you have + haven't done (in general) because it's either there or not there.

(3) can be fun when had time to practice on; can be educational and a good way for children to get into programming

(4) easy to follow dialogue box & menus.

What are the worst aspects of Gamut for the user?
(1) I haven't used any software that is supposed to "learn" what I'm trying to teach it. Maybe the software should anticipate what the user wants to do + make something that are visual rather than written.

(2a) It asks the same question twice.

(2b) trickiness of manipulating behaviors

(3) to many bugs; a little confusing to train; but experience w/ program & practice helped

(4) knowing you have to move the marker ie Winner back.

What changes should be made to improve Gamut so people can use it better?

(1) As above. Possibly, like, in a bowling ally, where some machines can show you the best approach to taking out the remaining pins, this software can help plan a series of action for the user to take. The software, as is, has too little intelligence for anticipating what the user wants + depends upon the user remembering a rules and sequences and stymuli to take.
I'm quite tired, and it was hard for me to remember, so I think the software should have relied less on my skill. The version of the software I tested doesn't seem to be for real beginners.

(2a) Make it easier to understand what it's asking

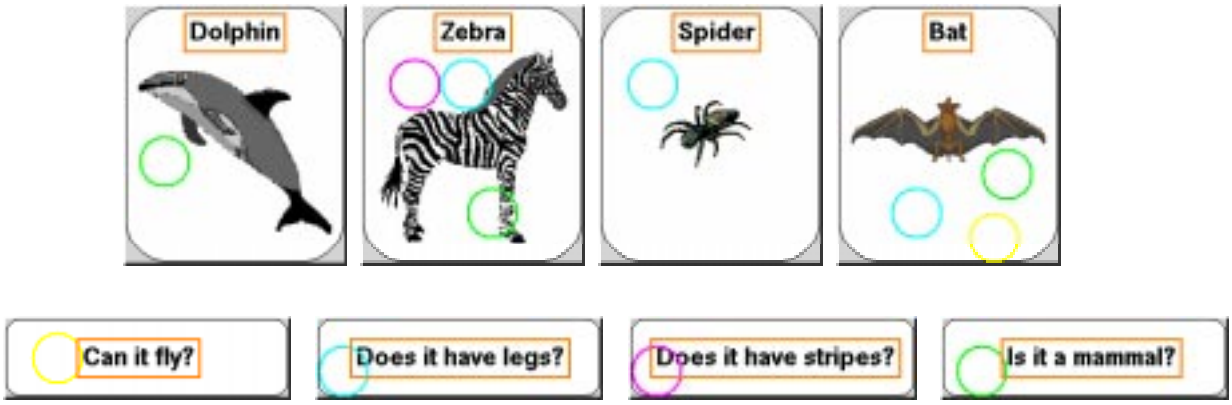
(2b) Make it more obvious what it thinks it's supposed to be doing

(3) pretty good, just give detailed manual and fix the bugs

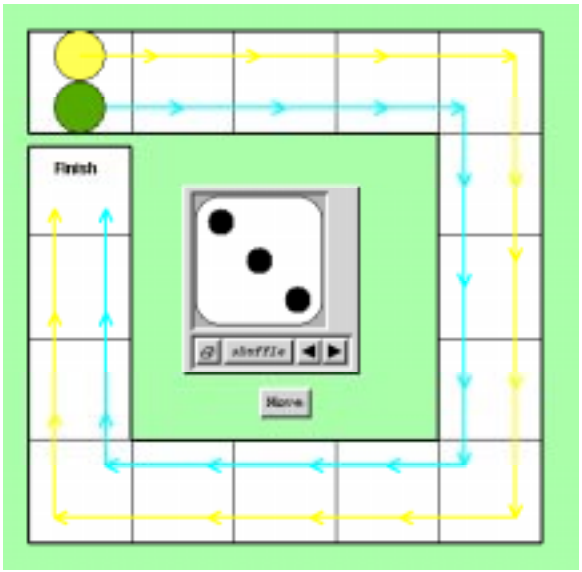
(4) Gamut Quick reference card for the palette.

E.2 Participant One's Saved Files

E.2.1 Task One: Safari

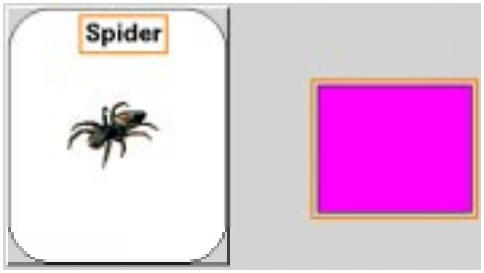


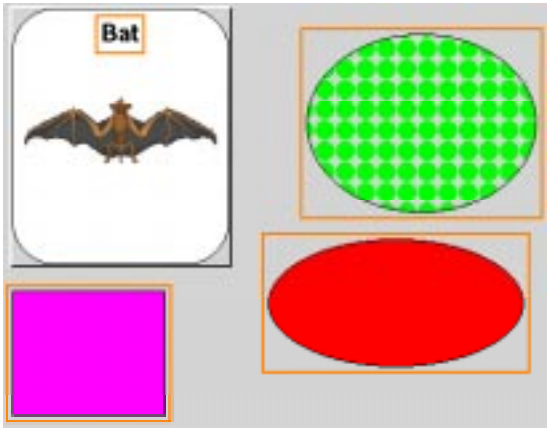
E.2.2 Task Two: Pawn Race

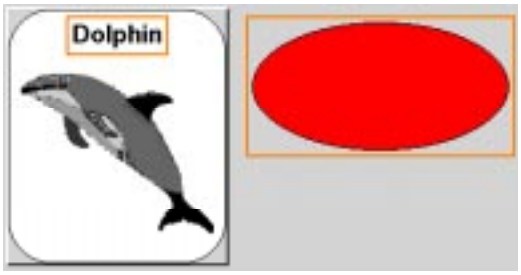


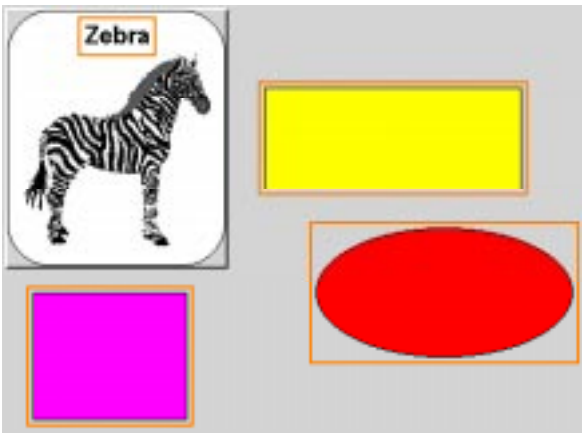
E.3 Participant Two's Saved Files

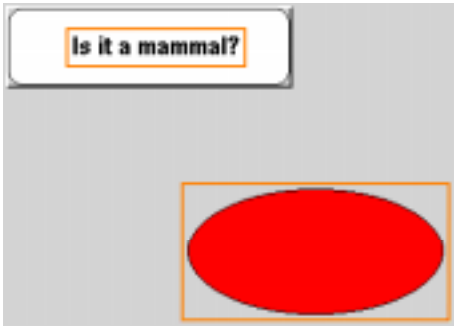
E.3.1 Task One: Safari

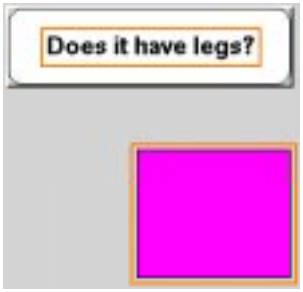


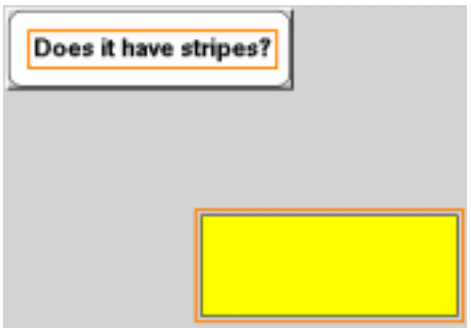


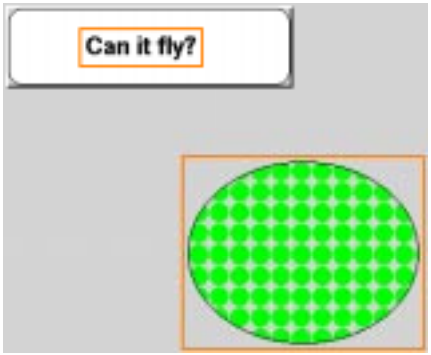




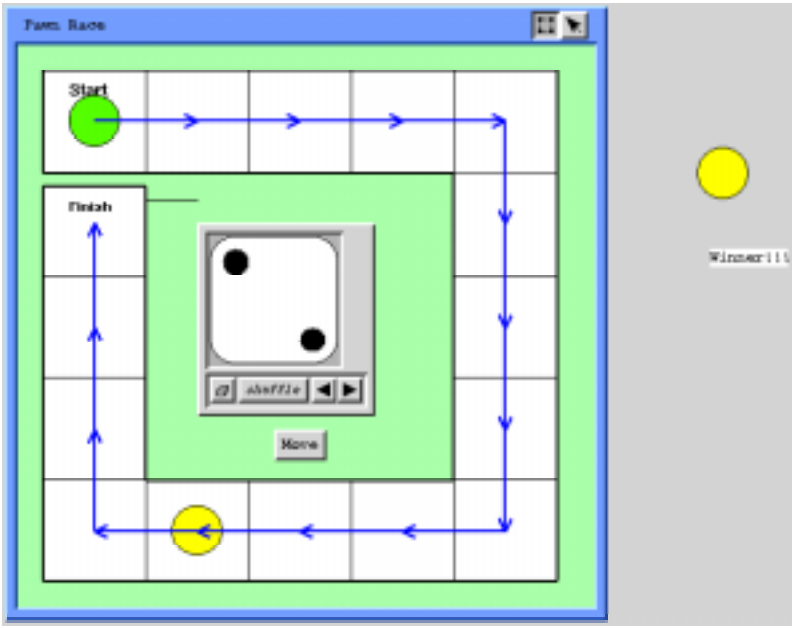




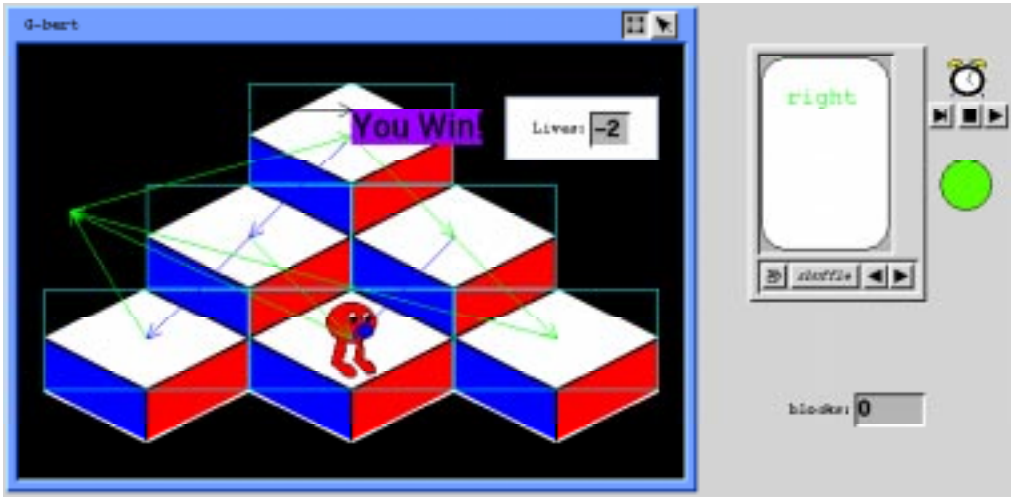




E.3.2 Task Two: Pawn Race

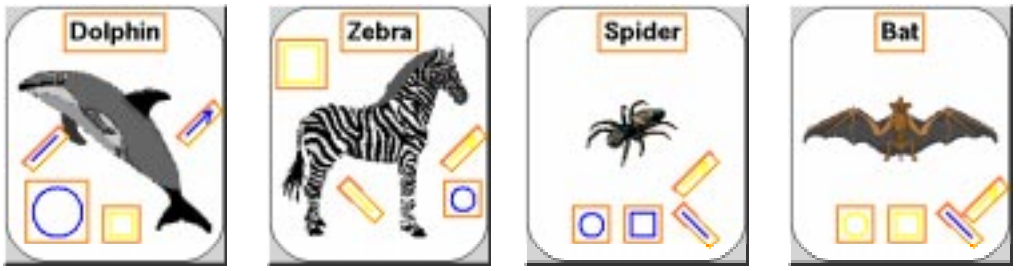


E.3.3 Task Three: G-bert



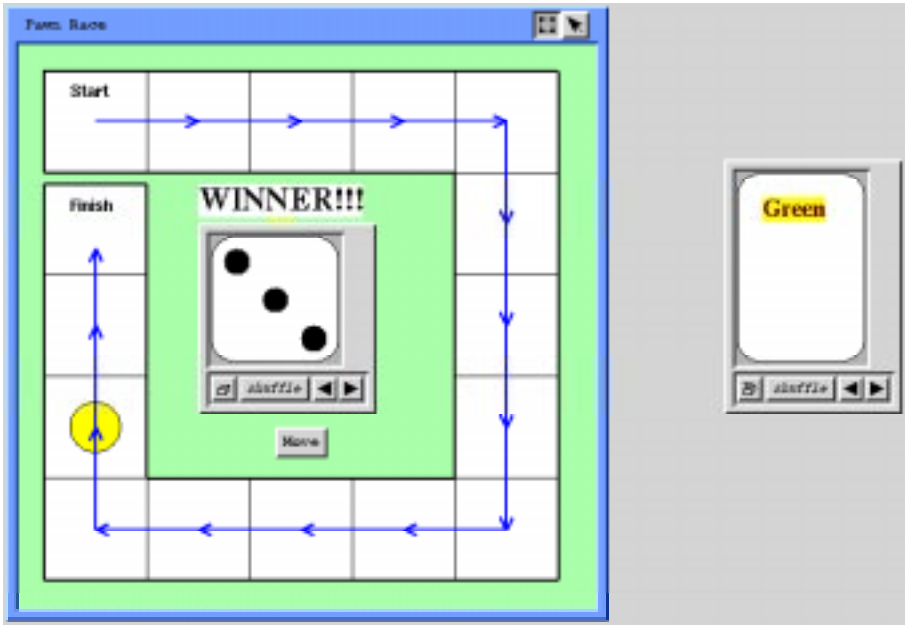
E.4 Participant Three's Saved Files

E.4.1 Task One: Safari



(participant 3 did not draw on the question cards)

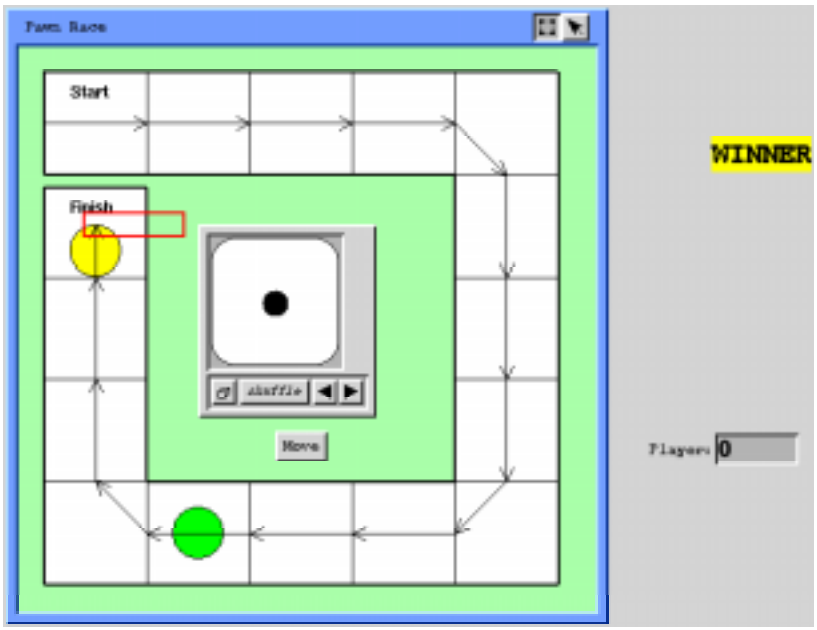
E.4.2 Task Two: Pawn Race



E.5 Participant Four's Saved Files

E.5.1 Task One: Safari

E.5.2 Task Two: Pawn Race



Appendix F: Gamut

> **gam'ut** (gam'ut), *n.* [*gam*ma, a name used formerly for the first note of the early scale + *ut.*] **1.** The series of recognized musical notes; sometimes, any recognized scale; specif., the major scale. **2.** An entire range or series.
>
> - *Webster's New Collegiate Dictionary, 2nd Edition*

The name of the project in this thesis is Gamut. It is not GAMUT. The word, Gamut, was selected because it gives a sense that the project's domain is all encompassing. The stuff you can make with this tool "runs the gamut." Also, Gamut kind of sounds like Amulet and Garnet which are names of the projects under which this project was created.

Is Gamut an acronym? Well, not really. However, the tools created for the Garnet and Amulet projects always seem to need silly acronyms. So, take your pick. Here is a bunch of acronyms any of which might be what Gamut stands for.

Games
Are
Made
Using
This

Great,
Another
Mind-numbing
User-interface
Tool

Generating
Applications
Made
Unusually
Trivial

Give me
Another
Min**UT**e
(it's almost done)

Totally
Understandable
Method to
Assemble
Games

Green
Apples
Make the
Ultimate
Treat

Gorilla
Aardvark
Mongoose
Unicorn
Tyranosaur

References

[1] Adobe Systems Incorporated. *FrameMaker.* 345 Park Avenue, San Jose, CA 95110-2704, 1986-1997.

[2] J. R. Anderson. *The Architecture of Cognition*. Harvard University Press, Cambridge, MA, 1983.

[3] J. R. Anderson, and R. Pelletier. “A Development System for Model-Tracing Tutors.” *Proceedings of the International Conference of the Learning Sciences*, Evanston, IL, January 8, 1991.

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

[5] Krishna Bharat and Marc H. Brown. “Building Distributed, Multi-User Applications by Direct Manipulation.” *Proceedings of the ACM Symposium on User Interface Software and Technology*, UIST’94, Marina del Rey, CA, November 2-4, 1994. pp. 71-80.

[6] Eric Allan Bier and Maureen C. Stone. “Snap-Dragging.” *Computer Graphics: Proceedings SIGGRAPH’86*, Vol. 20, No. 4, August 1986, Dallas, Texas. pp. 233-240.

[7] Alan W. Biermann. “Approaches to Automatic Programming.” *Advances in Computers*, Morris Rubinoff and Marshall C. Yovitz, eds. Volume 15. New York: Academic Press, 1976. pp. 1-63.

[8] Neil Bradley. *The XML Companion*. Addison-Wesley, Reading, Massachusetts, 1998.

[9] Ted Bridis. “Cyberspace is Driving America’s Economy.” *Pittsburgh Post Gazette*. Vol. 71, No. 259, April 16, 1998. p. A-1.

[10] Kraig Brockschmidt. *Inside OLE*, 2nd edition, Microsoft Press, Redmond, Washington, 1995.

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

[12] Robert C. Calfee. “Planning Psychological Experiments.” *Human Experimental Psychology*, Holt, Rinehart, and Winston, New York, 1975.

[13] Jadzia Cendrowska. “PRISM: An Algorithm for Inducing Modular Rules.” *International Journal of Man-Machine Studies*, Vol. 27, 1987. pp. 349-370.

[14] Craig Chambers, David Ungar, and Elgin Lee. “An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes.” *Sigplan Notices*, Vol. 24, No. 10, New Orleans, LA, October 1989. pp. 49-70.

[15] Peter Cheeseman, *et al*. “AutoClass: A Bayesian Classification System.” *Proceedings of the Fifth International Workshop on Machine Learning*, San Mateo, CA, 1988.

[16] *The Common Object Request Broker: Architecture and Specification*. OMG Document Number 93.12.43, December 1993.

[17] *Corel Click & Create*. Corel Corporation and Europress Software Ltd. 1996.

[18] Mark Craven, Seán Slattery, and Kamal Nigram. “First-Order Learning for Web Mining.” *Proceedings of the 10th European Conference on Machine Learning*. Chmnitz, Germany, 1998, Springer-Verlag. pp. 250-255.

[19] M. Cutter, B. Halpern, J. Spiegel. *MacDraw.* Apple Computer Inc., 1987.

[20] Allen Cypher. “Eager: Programming Repetitive Tasks by Example.” *Proceedings CHI’91: Human Factors in Computing Systems*, New Orleans, 1991, pp. 33-39.

[21] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, 1993.

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

[23] Gerald Dejong and Raymond Mooney. “Explanation-Based Learning: An Alternative View.” *Machine Learning*, Vol. 1, 1986. pp. 145-176.

[24] Paula Ferguson and David Brennan. *Motif Reference Manual*, Volume 6B. O’Reilly & Associates. 1993.

[25] R. E. Fikes, N. J. Nilsson. “STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving.” *Artificial Intelligence*, Volume 2, 1971. pp. 189-288.

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

[27] David Flanagan. *Java: Java In A Nutshell*. O’Reilly & Associates, 1997.

[28] David Flanagan. *Javascript: The Definitive Guide*. O’Reilly & Associates, 1996.

[29] James Foley, Andries van Dam, Steven Feiner, and John Hughes. “The Window-To-Viewport Transformation.” *Computer Graphics: Principles and Practice*, 2nd edition, Addison-Wesley Publishing Company, 1990. pp. 210-212.

[30] Martin Frank. *Model-Based User Interface Design by Demonstration and by Interview.* Ph.D. Thesis. Graphics Visualization and Usability Center, Georgia Institute of Technology, Atlanta, Georgia, 1995.

[31] Martin R. Frank. “Standardizing the Representation of User Tasks.” *Acquisition, Learning & Demonstration: Automating Tasks for Users*. Papers from the 1996 AAAI Symposium, Technical Report SS-96-02. pp. 20-22.

[32] Michael R. Garey and David S. Johnson. “The Theory of NP-Completeness.” *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Bell Laboratories, Murray Hill, New Jersey, 1979. pp. 17-44.

[33] Michael Gleicher and Andrew Witkin. “Drawing With Constraints.” *The Visual Computer*, Volume 11, Number 1, 1994. pp. 39-51.

[34] Kathleen Gomoll. “Some Techniques for Observing Users.” *The Art of Human-Computer Interface Design*, B. Laural, ed. Addison-Wesley, Reading, MA, 1990. pp. 85-90.

[35] Gottlieb. *Q*Bert*. 1983.

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

[37] Ralph Grabowski. *The Web Publisher’s Illustrated Quick Reference: Covers HTML 3.2 and VRML 2.0*. Springer, New York, 1997.

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

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

[40] Kristian J. Hammond. “CHEF.” *Inside Case-Based Reasoning*. C. Reisbeck and R. Shank, eds. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

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

[42] 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.

[43] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, Ken Doyle. “Fabrik: A Visual Programming Environment.” *Proceedings OOPSLA ’88*. pp. 176-190.

[44] B.E. John, A.H. Vera, and A. Newell. “Towards real-time GOMS.” *The Soar Papers: Research on Integrated Intelligence*. Paul S. Rosenbloom, John E. Laird, and Allen Newell eds., MIT Press, Cambridge, MA, 1993.

[45] Ken Kahn. “ToonTalk: An Animated Programming Environment for Children.” *Journal of Visual Languages and Computing*, June 1996.

[46] Solange Karsenty, James A. Landay, and Chris Weikart. “Inferring Graphical Constraints with Rockit.” *Human Computer Interaction: Proceedings of HCI’92*. York, United Kingdom, September 1992.

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

[48] The Learning Company. *Reader Rabbit*. 1987.

[49] Henry Lieberman. “Integrating User Interface Agents with Conventional Applications.” *Proceedings IUI’98: 1998 International Conference on Intelligent User Interfaces*. January 6-9, San Francisco, CA, 1998. pp. 39-46.

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

[51] Henry Lieberman and Christopher Fry. “Bridging the Gulf Between Code and Behavior in Programming.” *Proceedings CHI’95: Human Factors In Computing Systems.* Denver, Colorado, May 7-11, 1995. pp. 480-486.

[52] Rudi K. Lutz. *Toward 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.

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

[54] David Maulsby. *Inducing Procedures Interactively: Adventures with Metamouse*. M.Sc. Thesis. Department of Computer Science, University of Calgary, 1988.

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

[56] David Maulsby, Saul Greenberg, and Richard Mander. “Prototyping an Intelligent Agent Through Wizard of Oz.” *Proceedings of InterCHI’93: Human Factors in Computing Systems*, Amsterdam, 1993. pp. 277-285.

[57] Michael Meyer. “Fight to the Finish.” *Newsweek*, December 12, 1994. pp. 56-57.

[58] Ryszard S. Michalski. “A Theory and Methodology of Inductive Learning.” *Artificial Intelligence*, Vol. 20, 1983. pp. 111-116.

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

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

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

[62] Marvin Minsky. “A Framework for Representing Knowledge.” *Mind Design*, edited by J. Haugeland, Cambribge, MA, 1981. pp. 95-128.

[63] Tom M. Mitchell. “Artificial Neural Networks.” *Machine Learning*. McGraw-Hill, 1997. pp. 81-127.

[64] Tom M. Mitchell. “Decision Tree Learning.” *Machine Learning*. McGraw-Hill, 1997. pp. 52-80.

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

[66] Francesmary Modugno, Albert T. Corbett, and Brad A. Myers. “Graphical Representation of Programs in a Demonstrational Visual Shell -- An Empirical Evaluation.” *ACM Transactions on Computer-Human Interaction*. September 1997, Vol. 4, No. 3. pp. 276-308.

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

[68] Brad A. Myers. “A New Model for Handling Input.” *ACM Transactions on Information Systems*. Vol. 8, No. 3, July 1990. pp. 289-320.

[69] Brad A. Myers. “Scripting Graphical Applications by Demonstration.” *Proceedings CHI’98: Human Factors in Computing Systems*. Los Angeles, CA, April 18-23, 1998. pp. 534-541.

[70] Brad A. Myers. “Text Formatting by Demonstration.” *Proceedings ACM SIGCHI’91*, New Orleans, 1991. pp. 251-256.

[71] Brad A. Myers. “User Interface Software Tools.” *ACM Transactions on Computer Human Interactions*, Volume 2, Number 1, March 1995. pp. 64-103.

[72] Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, E Pervis, Anrew Mickish, and Philippe Marchal. “Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces.” *IEEE Computer*, Vol. 23, No. 11, November 1990. pp. 71-85.

[73] Brad A. Myers, Jade Goldstein, and Matthew A. Goldberg. “Creating Charts by Demonstration.” *Proceedings CHI’94: Human Factors in Computing Systems*. Boston, MA, April 24-28, 1994. pp. 106-111.

[74] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. “The Amulet Environment: New Models for Effective User Interface Software Development.” *IEEE Transactions on Software Engineering*, Vol. 23, No. 6, June 1997. pp. 347-365.

[75] Brad A. Myers and David S. Kosbie. “Reusable Hierarchical Command Objects.” *Human Factors in Computing Systems, Proceedings SIGCHI’96*, Denver, CO, April 1996, pp. 260-267.

[76] 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.

[77] Brad A. Myers, Richard McDaniel, Robert Miller, Brad Vander Zanden, Dario Giuse, David Kosbie, and Andrew Mickish. “The Prototype-Instance Object Systems in Amulet and Garnet.” *Prototype Based Programming*. James Nobel, Antero Taivalsaari, and Ivan Moore, eds. Springer-Verlag, 1999. To appear.

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

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

[80] S. Pangoli and F. Paterno. “Automatic Generation of Task-Oriented Help.” *Proceedings of the ACM Symposium on User Interface Software and Technology,* UIST’95, Pittsburgh, PA November 14-17, 1995. pp. 181-187.

[81] Tomi Pierce. “Creating a PC Game.” *Newsweek Special Issue: Computers and the Family.* Winter 1997. p. 22.

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

[83] J. R. Quinlan and R. M. Cameron-Jones. “FOIL: A midterm report.” *Proceedings of the European Conference on Machine Learning*, Vienna, Austria, 1993. pp. 3-20.

[84] Alex Repenning. *Agentsheets: A Tool for Building Domain-Oriented Dynamic, Visual Environments*. Dept. of Computer Science, University of Colorado at Boulder, 1993.

[85] M. Rettig. “Prototyping for tiny fingers.” *Communications of the ACM*, 37, 4 (April 1994). pp. 21-27.

[86] Stuart Russell and Peter Norvig. “Reinforcement Learning.” *Artificial Intelligence: A Modern Approach*, Prentice Hall, Upper Saddle River, New Jersey 07458. pp. 598-624.

[87] Stuart Russell and Peter Norvig. “Agents That Communicate.” *Artificial Intelligence: A Modern Approach*, Prentice Hall, Upper Saddle River, New Jersey 07458. p. 654.

[88] Eric Saund and Thomas P. Moran. “A Perceptually-Supported Sketch Editory.” *Proceedings of the ACM Symposium on User Interface Software and Technology,* UIST’94, Marina del Rey, CA, November 2-4, 1994. pp. 175-184.

[89] Robert W. Scheifler and Jim Gettys. “The X Window System.” *ACM Transactions on Graphics*, Volume 5, Number 2, April 1986. pp. 79-109.

[90] Ben Shneiderman. “Direct Manipulation: A Step Beyond Programming Languages.” IEEE Computer, Vol. 16, No. 8, August 1983. pp. 57-69.

[91] David C. Smith. *Pygmalion: A Creative Programming Environment*. Ph.D. Thesis, Stanford, 1975.

[92] Randall Smith, *et al*. “The Use-Mention Perspective on Programming for the Interface.” *Languages for Developing User Interfaces*. Myers, ed. Jones and Bartlett Publishers, 1992. pp. 79-90.

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

[94] Mary-Anne Williams. “Anytime Belief Revision.” *Fifteenth International Joint Conference on Artificial Intelligence*. August 1997. pp. 75-79.

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

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

[97] David Wolber. “Pavlov: Programming by Stimulus-Response Demonstration.” *Human Factors in Computing Systems, Proceedings SIGCHI’96*, Denver, CO, April 1996. pp. 252-259.

[98] William A. Woods. “What’s in a Link: Foundations for Semantic Networks.” *Representation and Understanding: Studies in Cognitive Science*, edited by D. G. Bobrow and A. M. Collins, Academic Press, New York, 1975. pp. 35-82.