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 participant. 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 innovation. 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.

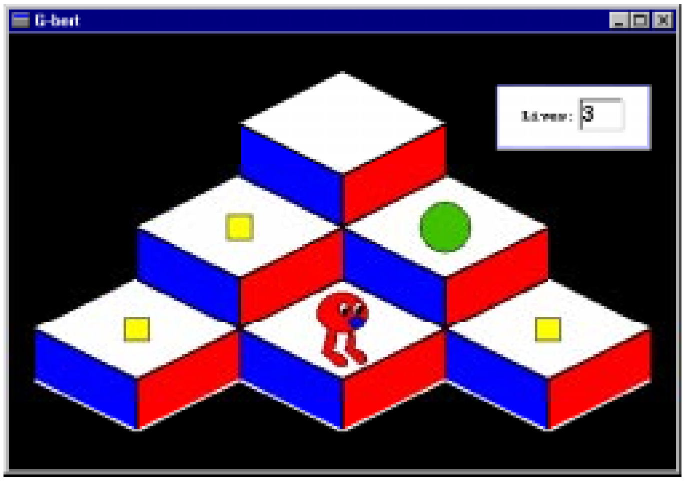


Figure 1.1: An example game created with Gamut. The board consists of a pyramid of cubes (drawn with bitmaps). The player's character jumps from cube to cube collecting markers and trying to avoid the balls falling down from above.

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 syntax 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

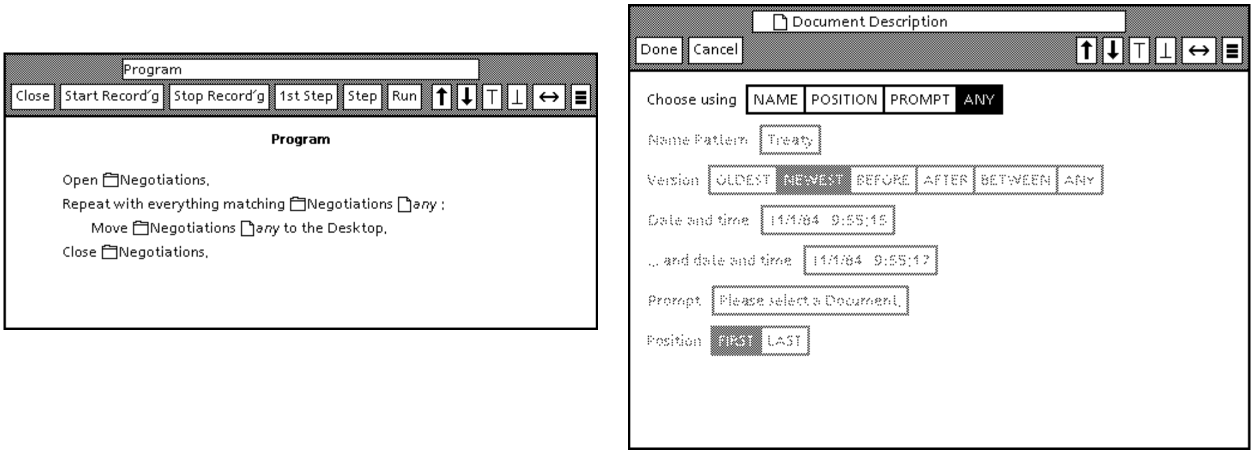
The past work in programming-by-demonstration spans about 25 years of research. The first efforts such as Smith's Pygmalion [91] involved using examples to aid in writing small programs. The field has since branched out from there. Recent research includes domains such as creating charts and graphs [73], 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.



Program

Open Negotiations.
Repeat with everything matching Negotiations *any* :
    Move Negotiations *any* to the Desktop.
Close Negotiations.

Choose using NAME POSITION PROMPT [✓ ANY]

Name Pattern Treaty

Version OLDEST [✓ NEWEST] BEFORE AFTER BETWEEN ANY

Date and time 11/1/84 9:55:15

... and date and time 11/1/84 9:55:17

Prompt Please select a Document.

Position [✓ FIRST] LAST

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.

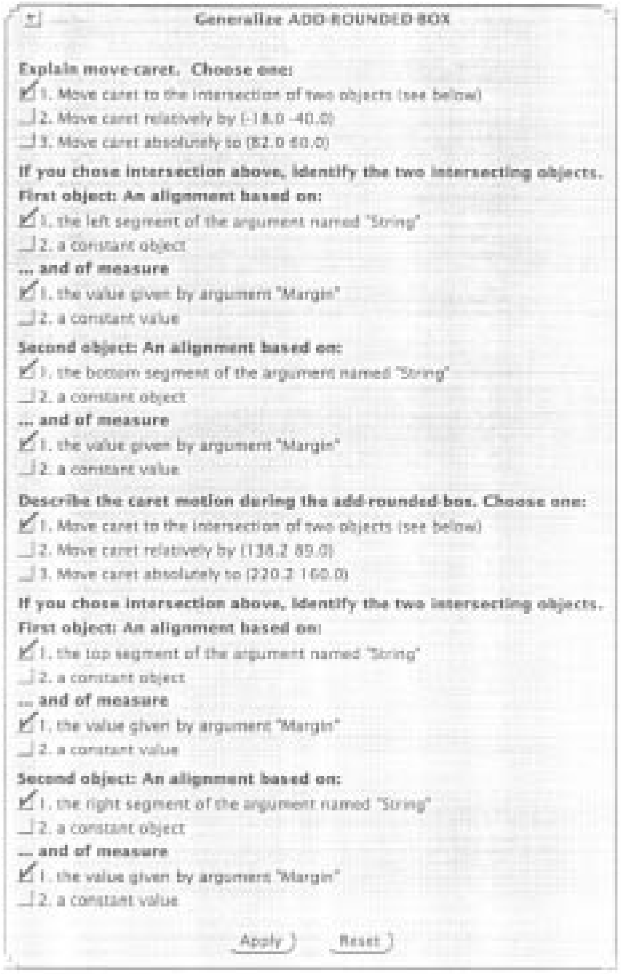
2.1.1.2 Kinds of Feedback

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.



Generalize ADD-ROUNDED-BOX

Explain move-caret. Choose one:

☑ 1. Move caret to the intersection of two objects (see below)
□ 2. Move caret relatively by (-18.0 -40.0)
□ 3. Move caret absolutely to (82.0 60.0)

If you chose intersection above, identify the two intersecting objects.

First object: An alignment based on:

☑ 1. the left segment of the argument named "String"
□ 2. a constant object

... and of measure

☑ 1. the value given by argument "Margin"
□ 2. a constant value

Second object: An alignment based on:

☑ 1. the bottom segment of the argument named "String"
□ 2. a constant object

... and of measure

☑ 1. the value given by argument "Margin"
□ 2. a constant value

Describe the caret motion during the add-rounded-box. Choose one:

☑ 1. Move caret to the intersection of two objects (see below)
□ 2. Move caret relatively by (138.2 89.0)
□ 3. Move caret absolutely to (220.2 160.0)

If you chose intersection above, identify the two intersecting objects.

First object: An alignment based on:

☑ 1. the top segment of the argument named "String"
□ 2. a constant object

... and of measure

☑ 1. the value given by argument "Margin"
□ 2. a constant value

Second object: An alignment based on:

☑ 1. the right segment of the argument named "String"
□ 2. a constant object

... and of measure

☑ 1. the value given by argument "Margin"
□ 2. a constant value

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 of the list so the pictures form a sequence of each action the behavior performs (see Figure 2.3). Liebermann's Mondrian system [50] uses a similar technique to display its behaviors. Mondrian converts the first and last picture of a behavior into a "domino" that the developer uses to cause the behavior to execute. Modugno also used the comic strip metaphor in her system called PURSUIT [66] which is shown in Figure 2.4. PURSUIT was used to infer operations that one might perform in a file system. PURSUIT also included annotations to represent conditional operations and parameters that Chimera did not. PURSUIT's interface was actually implemented in two styles. One has a pictorial representation of all the commands while the other used a more textual representation similar to the design in SmallStar. Modugno found that users preferred the pictorial representation and could understand it more easily.

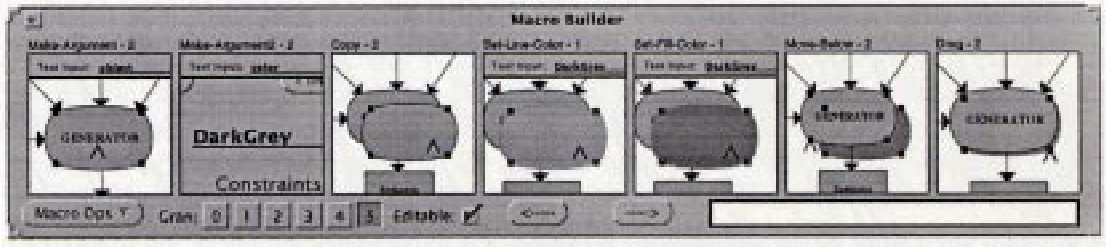


Figure 2.3: Chimera used a "comic strip" metaphor to display its code. Graphical operations were summarized in pictures that show how the graphics were transformed [47]. (Reprinted with permission.)

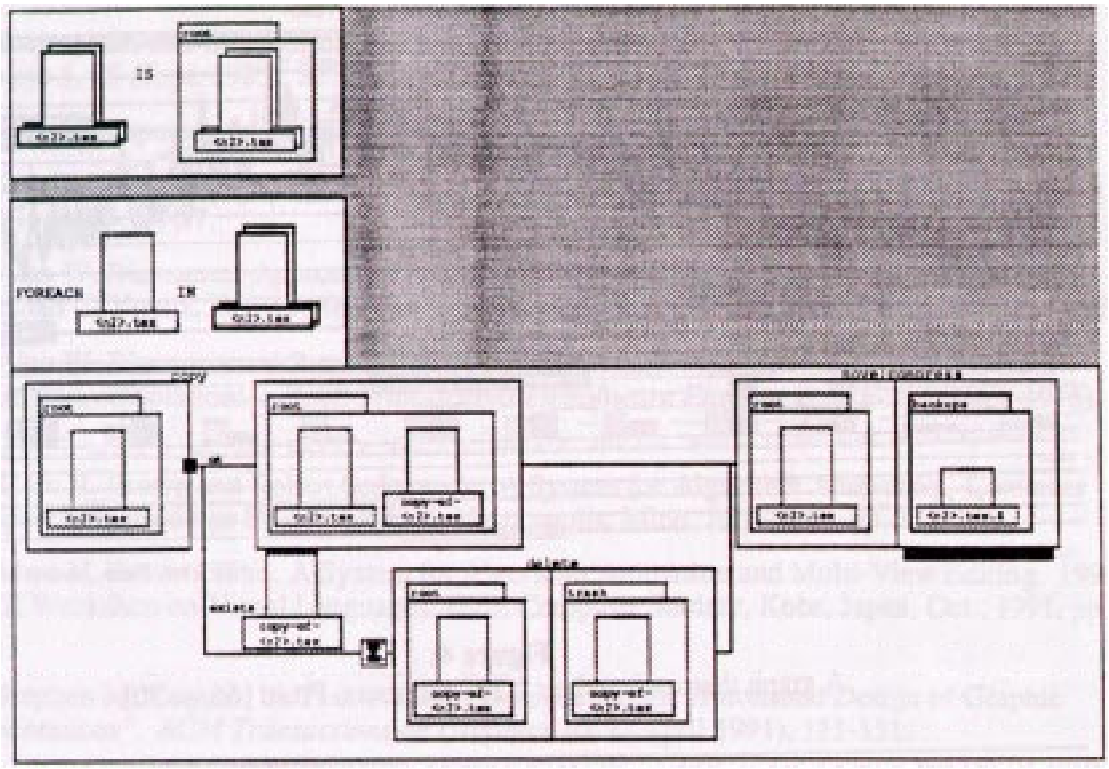


Figure 2.4: PURSUIT's comic strip interface included other annotations that represent variables and conditional branches [66]. (Reprinted with permission.)

Gamut does not possess a display for its internal code. Thus, in some ways it is like other systems which have failed to provide adequate feedback. Gamut does provide other feedback such as when it asks the developer for hints (see Section 4.2.3.2). When Gamut asks for a hint, it uses parts of the code to describe the problem it is trying to resolve, thus Gamut's display is more text oriented currently. In the future, some of the graphical methods for displaying code mentioned above may prove useful as a way for displaying Gamut's internal language.

2.1.1.3 Using Hints and Annotations

Gamut relies on the developer to provide hints and annotations in places where the system would have trouble making inferences (see Section 4.2.2.1). Most prior PBD systems did not provide the ability to make annotations and instead relied on plain unannotated examples. There were some exceptions, though. DEMO II [26] provided a "guidewire" annotation which it borrowed from Maulsby's Metamouse [54]. DEMO II and Metamouse used geometric graphics such as lines and rectangles to show relationships between other objects. Non-PBD systems like the NeXTStep [78] interface builder used objects to space out widgets and constrain how objects react to window resizing. Rehearsal World [36], which is also not a PBD system, used a technique called "off-stage actors" that is a metaphor to represent offscreen objects (see Section 4.1.2.2).

Maulsby's PBD text editing systems Turvy, Moctec, and Cima [55] all incorporated some form of hint-giving ability. Turvy and Moctec were not working systems but instead were "Wizard of Oz" experiments to see how users reacted to PBD. These included extensive hint-giving mechanisms through verbal commands. In Cima, the findings of those experiments were applied to a working tool. However, Cima was not quite powerful enough to perform real work. Cima's task was to recognize features of English text, but the capabilities of the language that Cima inferred were not structured to use hints effectively. As such, Maulsby concluded that hint-giving ability was not as useful as it may seem. However, Gamut's data descriptions and internal structure are specifically implemented to accommodate hints (see Section 6.5.2.2). Thus, hints are especially important in Gamut and allow Gamut to infer complex behaviors.

2.1.1.3.1 Hints in AI Systems

It is difficult to categorize AI systems that use hints because, in principal, a hint is just a part of the example data. It is difficult to tease apart which portion of an example is the hint and which part is the "true" example. For instance, a hint might be represented as extra attributes in the data set. For example, say that one is trying to infer whether a given web page is someone's home page by showing the system examples of home pages. The standard set of data would be the contents of the pages to be learned, but the system designer might find that the pages to which the tested pages are linked provide the system with more information that better distinguishes which ones are home pages [18]. The system designer could add this linking information to the data set to improve the system's learning. To the inferencing algorithm, though, this extra information would be seen not be considered differently from the rest of the example data. As a result, hints are more of a consequence of one's method for gathering examples than a fundamental principle of AI.

Besides annotating the example data, hints can also be used to direct the progress of the system's search engine. For instance, Instructo-SOAR [41] provides a language for the user to direct the system's search engine by pointing out which concepts it has learned are appropriate. The techniques used to provide hints in several AI systems have been oriented toward having the user directly modify the data structures that the AI system uses. For instance, if a system is performing a searching task to find a path to some goal, the user might direct the system by selecting nodes in the path that seem appropriate. These kinds of manipulation tend to be low-level and require the user to understand the task the system performs in intimate detail. Gamut, on the other hand, does not require the user to know how the system builds code so that the user may concentrate on what is being built.

2.1.2 Applications

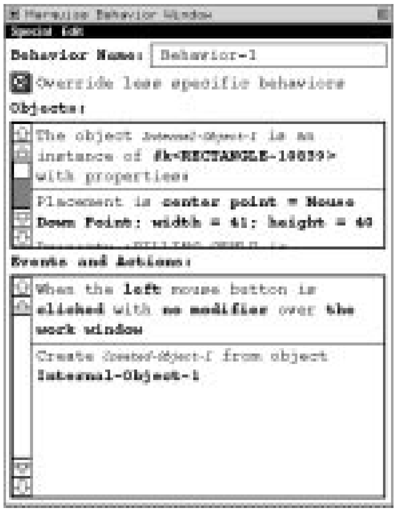
Though Gamut has drawn inspiration from many sources, a few PBD applications have provided significant influence. These were systems that have been applied to domains similar to Gamut's board game domain. The strongest influence is from Marquise because it was a system with which I had been personally involved. The next biggest influence was the DEMO II system from which many of Gamut's guide object concepts were derived. Another recent PBD system that supported a similar domain as Gamut and has some similar capabilities was Frank's Grizzly Bear.

2.1.2.1 Marquise

Gamut most resembles Marquise [76], a system that I helped build with Brad Myers and David Kosbie. Just like Gamut, Marquise's task was to develop whole applications, in this case graphical editors. Marquise provided a set of built-in tools for distinguishing common graphical editor characteristics. For instance, Marquise used a menu command to convert a group of objects into a palette. Gamut shares two aspects with Marquise. First, both use a similar icon arrangement to represent player mouse events. Second, Gamut uses a similar internal code representation as Marquise to distinguish events, actions, and descriptions.

However Gamut goes beyond Marquise's abilities and has both stronger inferencing and better interaction techniques. Marquise's heuristics only allowed the developer to demonstrate a single example. This rigidity made Marquise inferencing ability too low to be applied to the board game domain. Marquise would often construct unnecessary relationships such as graphically connecting two or more objects that were not related. The only way to correct these mistakes was to hand-edit Marquise's code using the editor shown in Figure 2.5. The editor was difficult to use because it created a separate dialog for each parameter in Marquise's code. Figure 2.6 shows how four levels of dialog were required in order to change the numeric offset in a description for moving an object. The fact that this dialog was so unwieldy was a major incentive for making Gamut's code revisable by demonstration.

Another failing in Marquise was how it used four different modes to demonstrate behaviors. Marquise defined Build, Run, Train, and Show modes. Run and Build were defined similarly to Run and Build modes in standard application builders. Train and Show were the demonstrational equivalent of Run and Build. In Build mode, the developer draws the static portion of the user interface. In Train mode, the developer performs the event portion of a behavior. In Show mode, the developer demonstrates the response for a behavior. The developer used control keys to switch between the modes. Having four distinct modes made the interface very complicated. Even expert users found the excess of modes irritating. Furthermore, switching between modes in unconventional orders was often undefined. For instance, what happened when the developer switched from Train to Build mode or from Build to Show mode? The mode problem in Marquise inspired the simpler design of the nudges technique in Gamut (see Section 4.2.1).



Marquise Behavior Window

Special Edit

Behavior Name: Behavior-1

☒ Override less specific behaviors

Objects:

The object Internal-Object-1 is an instance of **#k<RECTANGLE-19839>** with properties:

Placement is **center point = Mouse Down Point: width = 41; height = 40**

Events and Actions:

When the **left** mouse button is **clicked** with **no modifier** over **the work window**

Create Created-Object-1 from object **Internal-Object-1**

Figure 2.5: Marquise's dialog for editing code. The items in bold text could be selected to provide dialogs and menus that would let the developer choose different code fragments. [76]

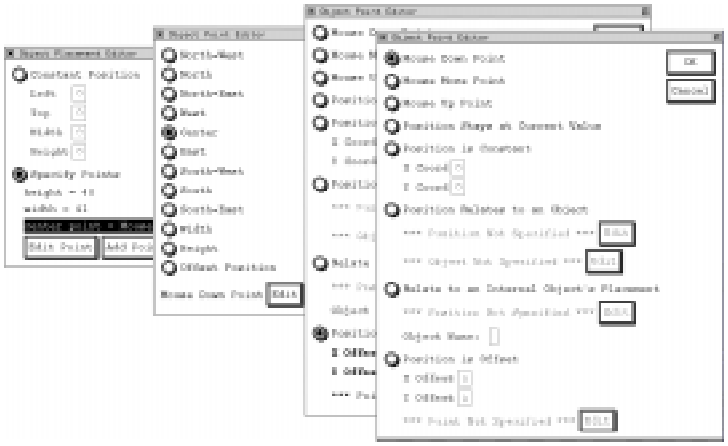
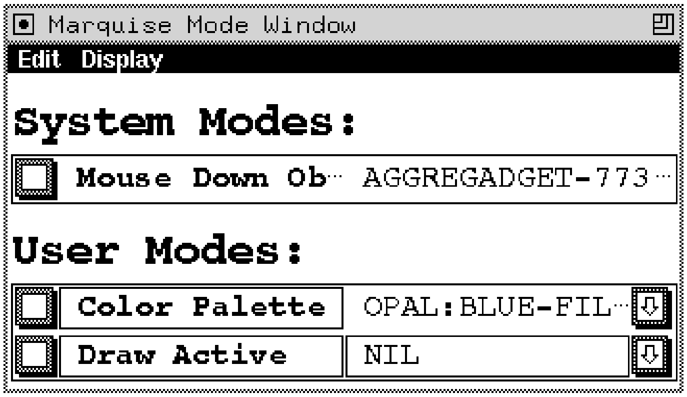


Figure 2.6: Here a portion of Marquise code is being edited by four levels of dialog. (It is adding an offset to the location description shown in the "Objects:" window of Figure 2.5.) It was difficult for a developer to edit the code because it was hard to reach the part that needed to be changed.

In Train mode, Marquise generated mouse cursors that are also used in Gamut (see Section 4.3.5). In Marquise, the mouse cursors would appear as the developer clicked and dragged the mouse on the screen. Since the real mouse caused the mouse cursors to appear, the developer had to use a control key in order to switch into Show mode to demonstrate the response of the event. This mechanism was fairly annoying which is why Gamut's technique does not require the developer to emulate the actual mouse events. The developer simply drops the desired icon on the screen and the computer interprets that icon as though it were the player using the mouse.

One final feature of Marquise worth mentioning was its Mode dialog window (shown in Figure 2.7). Marquise used a spreadsheet-like interface to store alpha-numeric data for the application. Each line of the dialog contained a string name and a value. The purpose of the dialog was to create global system modes for the created application. For instance, if the developer wanted to create a palette with three states, the developer would select the three objects that would become the images in the palette and use a menu command to tell the system that they were a palette. The system would then automatically create an entry in the Mode window that would represent the state of the palette. It contains a list of the names of the three objects from which the developer could select. The object selected in the entry would represent the mode of the palette that in turn could be used to represent a mode in the system. Beside the entry's name was a checkbox. If the checkbox was checked, the system would try to incorporate that mode into whatever behavior the developer was showing. This was the precursor to Gamut's hint highlighting (see Section 4.2.2.1).



System Modes:

□ Mouse Down Ob··· AGGREGADGET-773···

User Modes:

□ Color Palette OPAL:BLUE-FIL··· ⇩

□ Draw Active NIL ⇩

Figure 2.7: Marquise's Mode window. The developer created variables in this window to represent the various modes in the application. [76]

2.1.2.2 DEMO, DEMO II, and Pavlov

The Stimulus/Response demonstration style was first used by Wolber in the DEMO [96] system and was later applied to its successors DEMO II [26], and Pavlov [97]. DEMO was also the first system to allow the developer to demonstrate creating objects. The basis for each of these systems involved inferring linear constraints for each object's graphical properties with special consideration for dynamically created objects. The interaction techniques of these systems varied but generally involved selecting between Stimulus and Response modes from a menu.

Marquise's Train and Show modes were based on DEMO's Stimulus and Response modes and they function in virtually the same way. In Stimulus mode, the developer demonstrates the behavior's event, and in Response mode, the developer shows what the behavior performs in response to the event. Unlike Marquise, though, DEMO focused on using object manipulation and not the mouse to generate events (though mouse events were supported, too). For instance, moving an object would be a typical event in DEMO. In the thermometer example shown in Figure 2.8, the developer would move the needle on one thermometer as the event that would move the needle on the other. DEMO did not have Marquise's mouse input icons so demonstrating mouse events was more difficult. Since the developer had to push a button to switch from Stimulus to Response mode, the developer also could not hold down the mouse for a button down event and simultaneous switch the mode. DEMO used a dialog to disambiguate these situations.

Conditions in DEMO were not inferred by the system as they are in Gamut; instead, the developer was required to choose conditions using a complicated process. Rather than using normal Response mode, the developer selected "Demonstrate Conditional Response" from the menu.

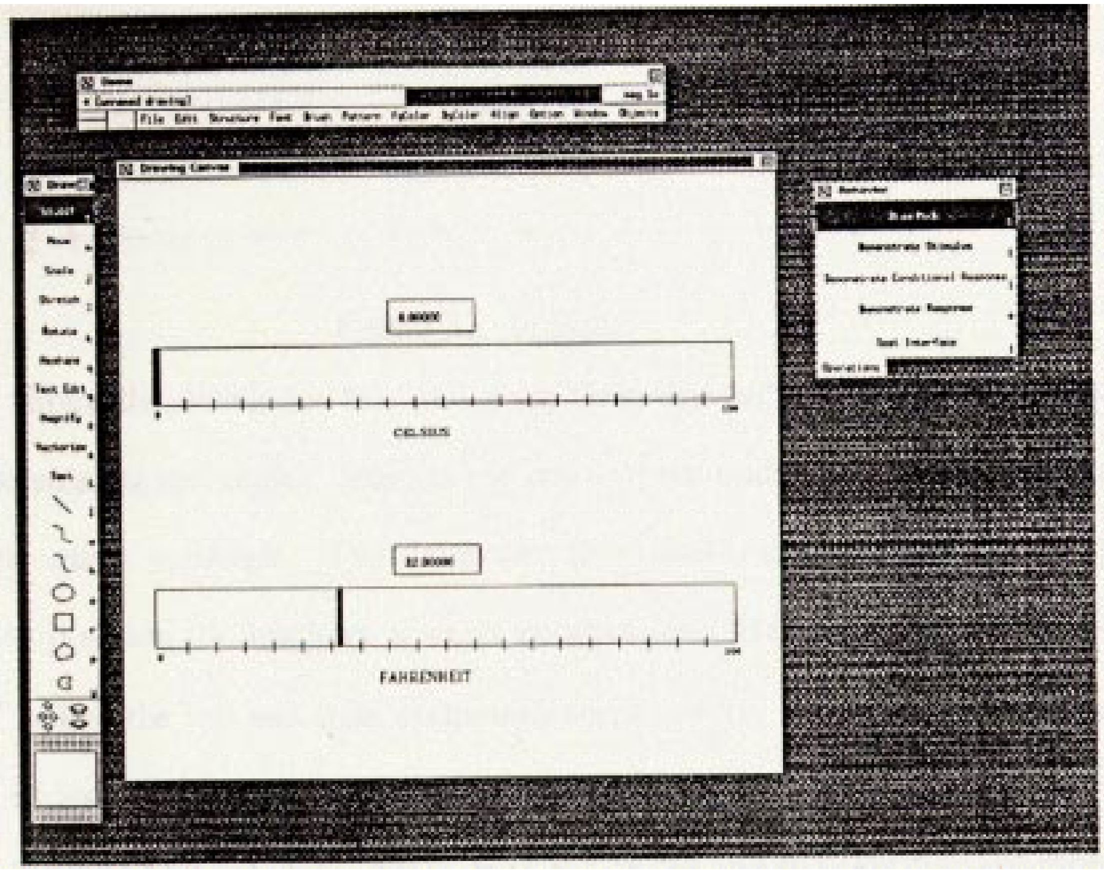


Figure 2.8: Thermometer example in DEMO. The developer could demonstrate that moving the needle in one thermometer would move the other needle a corresponding amount [96]. (Reprinted with permission.)

This invoked DEMO’s rule-based constraint detection algorithm. Like Peridot, DEMO used a set of rule-based heuristics and iterated through them to find appropriate questions to ask the developer. The dialog would ask the developer whether to apply the condition to the response or not and whether or not to negate the condition. The system would also ask the developer whether the condition should be applied before or after the behavior’s response. The latest system, Pavlov, uses a similar mechanism for assigning conditions but is much better at summarizing the developer’s choices. The system automatically invokes the condition generator after a stimulus event is demonstrated.

DEMO II did not use this dialog mechanism to ask the developer about conditions. Instead, it stored all the possible constraints in a list. On subsequent examples, the number of constraints in the list would be trimmed, removing those constraints that were no longer true. The remaining constraints would be applied in a constraint solving algorithm to make the application function. Gamut, on the other hand, does not use a constraint solving algorithm, but instead forms small pieces of code for its behaviors. Gamut then refines this code with information from subsequent examples.

One technique that Gamut borrowed from DEMO II directly was its “guidewire” concept. In Gamut, these are called onscreen guide objects (see Section 4.1.2.1). Figure 2.9 shows a screenshot from DEMO II where the developer created a game called “xeyes.” The xeyes program consisted of a pair of eyes whose pupils followed the mouse. The developer used several guidewires to construct the application, the most important being the line that connected the center of each eye to the mouse pointer. The guidewires could be made invisible when the program was running. By using guidewires, the system could learn the necessary behavior by inferring connections between objects and did not have to infer complex trigonometric equations. Gamut generalizes the guidewires concept to support arbitrary guide objects.

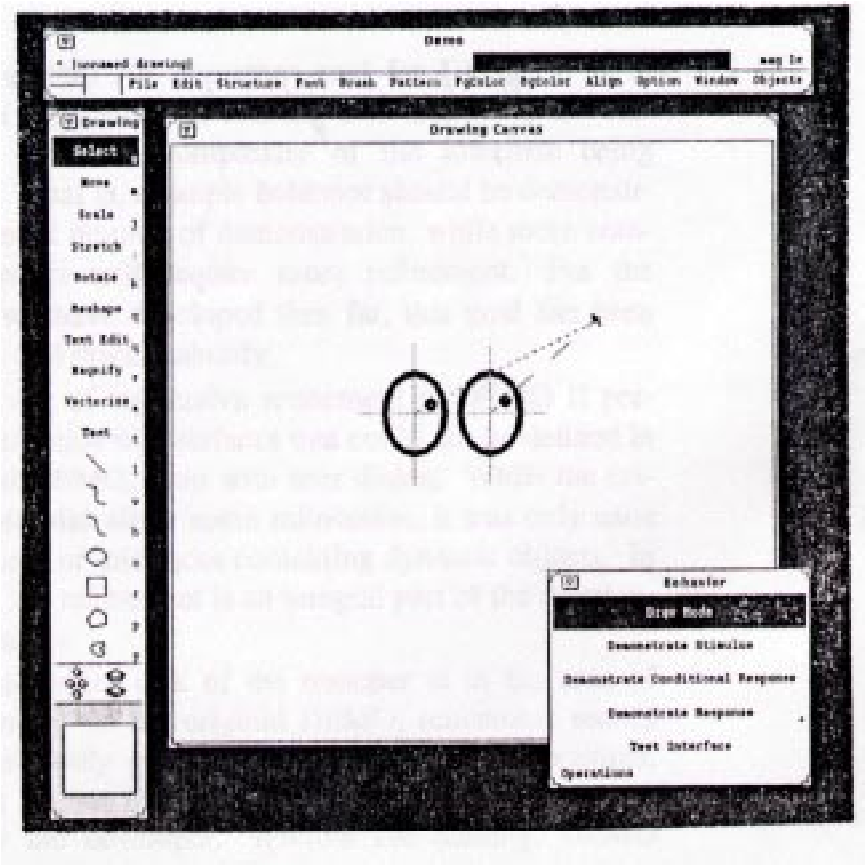


Figure 2.9: Creating the game xeyes in DEMO II uses guidewires to connect the pupils with the position of the mouse [26]. (Reprinted with permission.)

Pavlov does not possess DEMO II’s guidewire mechanism but it does have a special purpose object that uses an arrow to represent velocity. By re-orienting the arrow, the speed and direction of a graphical object can be made to change. The velocity arrow is not a true guidewire object as in DEMO II since it is part of the object and the developer does not draw it. The main thrust in the Pavlov research has been displaying complex animated behaviors. Pavlov allows the developer to specify times for behaviors and it uses a scoresheet metaphor like Macromedia’s Director to represent behaviors with multiple parts. Gamut does not handle timed behavior this way, but does allow the developer to include timers in the application.

2.1.2.3 Grizzly Bear

Frank’s Grizzly Bear system could create graphical applications like Gamut, though in several ways, it was more restricted. Grizzly Bear’s inferencing could generate some forms of conditional logic and object descriptions. However its object descriptions were limited to two forms neither of which supported nested descriptions. Grizzly Bear's inferencing algorithm was designed to be domain independent, so it lacked the custom rules that systems like Peridot, DEMO, or DEMO II possessed. The domain of the system was determined primarily by the objects and widgets that the system supported and the manner in which the properties of those objects were arranged. The system divided property values into different types and used different rules to describe each type.

Grizzly Bear's internal language was based on a list of assignment statements grouped within an event descriptor. Grizzly Bear events were methods attached to the objects within the interface. For instance, the background window had an event named "pressed" which would be called when the user clicked the mouse on the window. The assignment statements assigned values to the interface's objects by reading and setting the objects' properties through object descriptions. The system used a matrix technique to determine which properties affected others. Grizzly Bear could use this matrix to infer linear constraints between numerical properties and equality constraints between other values. The matrix technique allowed Grizzly Bear to create formulas using variables other than the parameters in the stimulus event. However, Grizzly Bear had no mechanism to distinguish which variables it should prefer over others and it was also limited to inferring relatively simple expressions.

Grizzly Bear's object descriptions came in two forms. The basic form referred to an object as a constant value. For instance, the description, `Box.color`, would refer to the color property of the constant Box object. The other form was called a "wildcard" form where the values of an object's properties determined which object was selected. The description, `(*.selected == "true").color`, refers to the color property of all objects whose selected property equals true. Grizzly Bear used parenthesized arguments like this to represent conditional expressions as well. The description, `(*.name == "Box" && Checkbox.value == 1)`, returns the object whose name is "Box" but only when the Checkbox object's value equals one. Though conceptually Grizzly Bear's object description system is complete and could be used to represent complicated descriptions, in practice, the inferencing system could not generate indirect descriptions based on chains of objects. In other words, the inferencing algorithm could not describe an object that was based on the description of another object. For example, Grizzly Bear could not describe that the path that an object follows is based on an arrow line that is connected to the object's original position.

Grizzly Bear's interface was based on a camera metaphor. Developers would take before and after pictures of the desired behavior. The system allowed the developer to take as many pictures as was necessary, but did not provide any convenient way for the developer to know whether the examples were successful. The snap-shot interface also did not allow the developer to revise a behavior after it had been created.

Instead of guide objects, Grizzly Bear required developers to add extra properties to objects in the interface manually to represent various modes and states. For instance, in order to show which object was selected, the developer might add a "selected" property to all objects that could be drawn by the user. The developer might define that when the selected property was set to true, the object was selected. In that way, a short wildcard description like `(*.selected == "true")` could be inferred by the system to describe the set of selected objects. The developer would demonstrate modifying these extra slots as well as show the interface's external behavior as a response. The developer could also create normal objects onscreen and set their "visibility" property to false that would make the objects invisible. The developer would use invisible objects like this as the prototypes for demonstrating the creation of new objects.

The developer could not give the system hints. Instead, the system attempted to infer which properties were involved with each behavior change. Since the developer was not running the interface to create a new example, the developer could be expected to reduce the amount of variation in the interface to only show the system what mattered. The system would not run partially inferred behaviors while the developer demonstrated, so that it would not interfere with what the example showed. The developer was expected only to modify objects on which the behavior actually depended.

2.2 Tools Used to Build Applications

Programming-by-demonstration systems are still in the research stage and are not the common way that people build software today. However, there are software tools that people use to reduce the amount of programming required for parts of an application. These tools are essentially special purpose languages and editors that describe programs in their domain more concisely. In the graphical interface domain, these tools naturally gravitate toward graphical representations that are edited with direct manipulation. There are several kinds of tools for creating graphical software [71]. There are “interface builders” that allow the developer to draw an interface in an editor and then write code to make it work. There are also programming environments for higher-level languages that often include user interface editors as well. These include prototyping tools that are used to quickly program an application for interface testing purposes. Languages designed for children are also available. These often include novel interaction techniques and are designed to be simple so that a child might understand them. Finally, “application builders” are tools tailored to a specific domain that allow a developer to create applications often without writing code at all.

2.2.1 Toolkits and Application Frameworks

Most software is created using programming languages like C++. These languages do not supply direct support for creating graphical user interfaces. Instead, that support is found in libraries called *toolkits*. A toolkit is the set of widgets and procedures used to create standard graphical interfaces. A toolkit is often part a broader user interface management system (UIMS) that determines the structure of the code used to write graphical interfaces. The UIMS can also be part of an application framework that is used to structure an entire application. Gamut is built using the Amulet UIMS [74]. An example of a toolkit would be Motif [24] which defines the look and feel of buttons, scrollbars, and other kinds of widgets. The Microsoft Foundation Classes (MFC) [59] is an example of an application framework which also includes interfaces for writing graphical interfaces as well as a toolkit. Writing software using these sorts of systems requires extensive programming experience.

2.2.1.1 Amulet

Amulet is a research system designed by Myers and his associates including myself. It was based on an earlier system called Garnet [72]. Both use a prototype-instance-based object system [77] in which new object types can be created dynamically at runtime. Most object systems are class-instance based where new types of objects can only be created at compile time. Amulet contains a number of innovative features that makes creating user interfaces easier. First, basic graphical operations such as refreshing the screen, controlling z-order, and maintaining display lists is all handled automatically. Also, Amulet provides a powerful model for user input called interactors [68]. An interactor is a direct application of Smalltalk's Model-View-Controller design [42]. An interactor is a separate object that represents a specific kind of user input. The interactor may be attached to graphical objects and the graphics will behave using that interactor's input mechanism. Another feature in Amulet is constraints. Amulet's objects are tied together using constraints so that when one object changes, other objects will change automatically using Amulet's constraint propagation mechanisms. Gamut uses the constraint mechanism for its own interface, but it is not used to describe Gamut's inferred behaviors.

One feature of Amulet that Gamut uses heavily is its command object architecture [75]. Amulet represents its various operations using command objects. These objects, when invoked, are copied and placed into the application's event history list. Command objects contain methods for executing the basic operation of the command as well as the methods for undo, redo, and other operations. The command object also contains the application state that the command modified. Having the command objects available in an accessible form allows other parts of the system to display and modify the event history. This can be used to create macro recorders and similar facilities [69]. Gamut uses the command object architecture to record the developer's examples. The architecture is also used in Gamut's internal language (see Section 5.3). Gamut's event and action objects are all based on command objects.

2.2.2 User Interface Builders

A user interface builder is essentially a direct manipulation drawing tool that uses interface widgets like menus and scrollbars in addition to graphics like rectangles and circles. UI builders like the ones included with NeXTStep [78] and Visual C++ [61] allow the interface designer to arrange widgets on a prototype window, choose colors and styles, and experiment with different configurations. When the designer finds the interface acceptable, the layout is converted into code. The code will contain all the information necessary to draw the interface, but the parts that perform the application's behaviors are left blank. A programmer then fills out the stub procedures or writes callbacks to connect the interface to the actual application code. UI builders focus on widgets and typically give no assistance to the designer for specifying behavior beyond the widgets. Gamut is designed to provide full assistance for specifying novel behaviors without writing code.

2.2.3 Higher-Level Programming Environments and Prototyping Tools

When a user interface builder is tightly coupled with a higher-level-language, the result is often an environment that is easier to program than a UI builder alone. Languages like Smalltalk [42] and Visual Basic [60] have graphical primitives built-in to make creating the interface easier. Like traditional UI builders, these tools lack the ability to specify behavior without programming. One must still add code (though in the higher-level language) to the various widgets and events in order to make them perform any action.

One of the early programming environments was Gould and Finzer's Rehearsal World [36]. Rehearsal World based its widgets on an acting metaphor: widgets were called "actors" and events and behavior were called "cues." By connecting cues to actors' attributes, one could generate programs. Looking beyond Rehearsal World's acting metaphor, one can see many of the attributes of modern programming environments such as Visual Basic. The developer created widgets and placed them onto the drawing surface.

Prototyping tools are UI building environments that allow a limited set of behaviors to be specified through direct manipulation. Prototyping tools are purposely kept simple to enable a designer to implement different interface ideas quickly. When the intended behavior becomes too complex to be specified using the direct manipulation methods, the developer uses the tool's scripting language just as in higher-level language programming environments. For example, HyperCard [4] uses an "index card" metaphor where cards represent windows and transitions between system states is handled by having HyperCard "go to" the appropriate card. Note that HyperCard's index card metaphor is not the same as Gamut's playing-card metaphor (see Section 4.3.1). Gamut's deck of playing cards is a separate entity from any given window. The graphics on a given card need not even be shown to the player. Furthermore, one does not "go to" a playing card; playing cards are shuffled, dealt, and played.

Macromedia Director [53] is another modern prototyping tool. It uses a score sheet abstraction to lay out events as one would a musical composition. This allows the developer to build a movie-like simulation of an interface. However, if developers want to add interactive behavior to the interface, they must resort to Director's scripting language called Lingo.

2.2.4 Languages for Children

A number of programming environments are aimed at making programming fun for children. These systems do not try to hide programming details but instead they try make the environment entertaining and attractive. These systems also must be easy to use so as not to frustrate their young audience. Among these systems are Repenning's Agentsheets [84], Smith's Cocoa [22], and Kahn's Toontalk [45]. Each of these languages is simple. That is, they do not possess higher-level abstractions such as hierarchical types, program libraries, or structured programming. The purpose of these language is to introduce children to programming without burdening them with unneeded complications. Although none of these systems use inferencing or techniques that are like Gamut's, these systems are often compared to Gamut because they can be used to construct games and simulations.

Agentsheets' language is composed of graphical code segments. Instead of typing in code using a keyboard, the developer drags code segments out of a palette and snaps them together (see Figure 2.10). Though the basic code structures the developer must create are not much different from other languages, the process of dragging and snapping together code is entertaining and it helps eliminate syntax and type-mismatch errors. Agentsheets is built on a tile metaphor. The world is broken into square segments that are filled with graphical icons.

Cocoa uses the same tile-based graphics technique as Agentsheets. However, Cocoa does not use Agentsheets' code editor to construct behavior. Instead, Cocoa uses before-and-after pairs of pictures. The before picture shows the configuration that Cocoa must recognize, the after picture shows how Cocoa should change the picture to make the behavior occur. Figure 2.11a shows how these pictures can be used to move a train object around the screen. The developer can add conditions to the before-and-after pairs using a special dialog shown in Figure 2.11b. The developer selects a condition from a set of predefined types and then parameterizes it to perform the needed operation.

Toontalk translates video game-like behaviors into a programming language. Programming Toontalk is a considerably different experience from programming in a normal language. The lan-

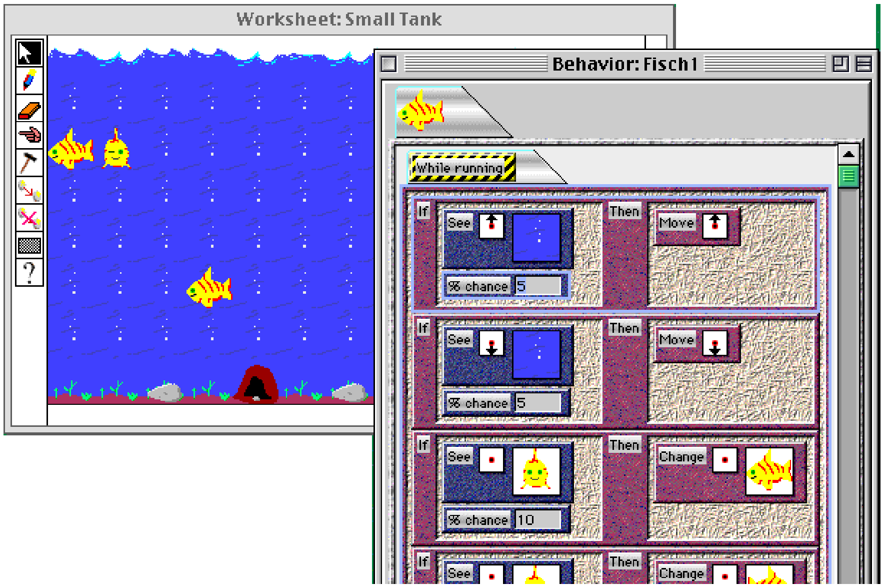


Figure 2.10: Writing code in Agentsheets involves snapping together graphical code segments. The segments are selected from a large palette of choices [84]. (Reprinted with permission.)

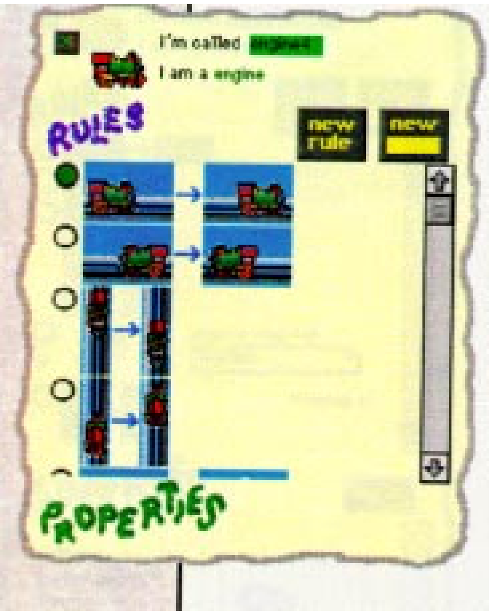
guage's "code" is stored in objects like houses where birds and rodents run around and cause various computations to occur (see Figure 2.12). The developer is given a video persona who walks around the world and adds and modifies objects as desired. Programming in this language is a visceral experience. One blows up the houses to delete variables and rides around in a helicopter to reach different parts of the application.

Though Agentsheets, Cocoa, and Toontalk are not PBD systems, one could possibly use similar techniques to display what the system has inferred. These systems show the kinds of feedback from which a PBD system might benefit.

2.2.5 Application Builders

An application builder is tailored toward building a specific kind of application usually without programming. For example, Pinball Construction Set [11] lets the author draw the parts of a pinball game in a window. Components such as ball and levers have behavior built-in and other connections are made using menus. A game built with this tool can be executed immediately and with little effort. Another domain is the class of financial applications that can be built with similar ease using spreadsheets. A spreadsheet's cells hold the requisite numbers and equations in a mostly free-form style that is easy to create and use. Likewise, form generators can be used to build data entry systems [71]. Like a UI builder, the designer can lay out data widgets onto forms. Since most data base entries possess no intrinsic behavior (it is the database system that stores, reads, and catalogues the entries, not the data), standard widgets are sufficient to build the whole interface.

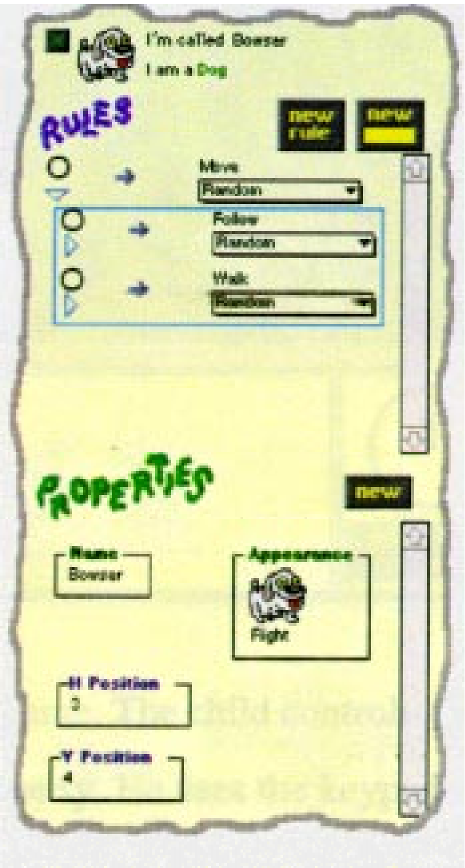
A modern application builder for making games is Corel's Click & Create [17]. Click & Create is entirely menu-based. One lays out the objects in the game using the editor and then assigns game behavior to each component using menus. Each kind of object has a number of predefined behaviors such as bouncing or moving with acceleration. The developer selects the behavior and sets the



I'm called engine4

I am a engine

Figure 2.11a: Cocoa uses this dialog to display before-and-after tile transformations. The before picture is copied from the background scene and the after picture is created by editing the before picture [22]. (Reprinted with permission.)



I'm called Bowser

I am a Dog

Move: Random

Follow: Random

Walk: Random

Name: Bowser

Appearance: Right

H Position: 3

Y Position: 4

Figure 2.11b: This dialog is used in Cocoa to add conditional guard statements to the tile transformations [22]. The pull-down menus in the "rules" portion of the dialog contains a list of possible conditions such as random, selected by a keypress, and others. (Reprinted with permission.)

parameters to the desired value. The developer is limited to the behaviors in the menu and cannot create new ones. The developer can create conditions using a spreadsheet-like dialog where the potential interactions between objects are listed. For instance, the developer can trigger a behavior when two kinds of objects collide.

Though application builders can be used to build complete applications, they cannot be used to create new forms of behavior without programming. The domains of these tools have been selected to minimize the amount of novel behaviors that the developer may require. Thus, building applications in that domain require only a small set of specific behaviors which can be mapped to a small set of techniques.

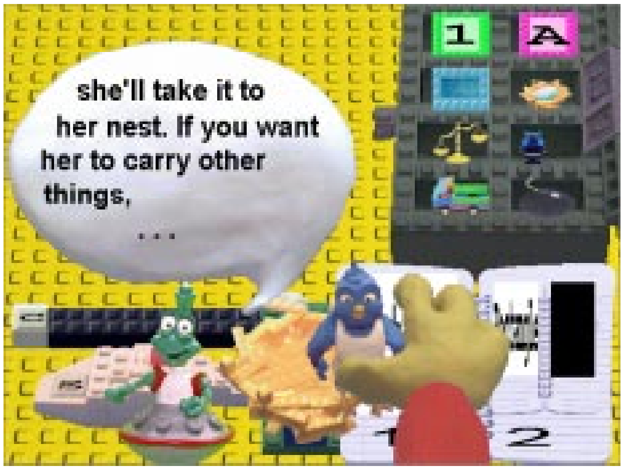

she'll take it to her nest. If you want her to carry other things, . . .
1
A
1
2

Figure 2.12: A picture of the Toontalk's code. Video game characters carry out the program's activities [45]. (Reprinted with permission.)

2.3 Machine Learning

The inferencing performed by PBD systems is a form of inductive learning. In inductive learning, a set of examples is presented to the system which uses them to generate an internal model that represents the learned concepts. The results of the system can later be used to apply the concepts to some task. There are many variations to inductive learning. The differences between techniques depend both on the kinds of examples the system can use and the kind of internal language the system produces to describe the examples.

2.3.1 A Few Properties of Machine Learning Domains

Machine learning is a broad field and its techniques have been applied to many domains. When machine learning is applied to areas such as speech recognition and machine vision there are different criteria to consider but several basic properties have emerged that most AI induction algorithms share. Induction algorithms are based on reducing a set of training examples into some form such that the system might later recognize and properly classify examples in the desired domain. The structure and the purpose of the examples can differ widely as can the method by which the learning algorithm applies the examples to its own structures.

Winston's classic system that learned to recognize the properties of arches was one of the first inductive systems [95]. Winston used this system to discuss various techniques through which an AI system might be trained. In Winston's system, training examples fell into two categories: positive examples and negative examples. A positive example is one where the learned concept holds whereas in a negative example, the concept is not true. In domains where the system must learn more than one concept, the examples are labelled to list the concepts for which the example is positive. The example would be negative for all other concepts. Typically, the concepts are selected so that a given example will only be positive for one concept.

Another property of domains is the amount of *noise* found in the examples. A noisy domain is one in which some of the examples are mislabelled or somehow wrong. For instance, one of the attributes of an example might be set to the wrong value, or an example might be improperly labelled positive when in fact it is negative. Noise arises from the method in which examples are gathered. The examples might be gathered using physical equipment which will always possess some measurement errors. Example data entered by hand can also have errors if the person recording the data makes mistakes. *Overfitting* results when a learning algorithm attempts to learn the noise in the examples as if it were valid data and does not try to generalize [64]. Overfitting is only problematic when the number of examples is large compared to the number of concepts. However, when the number of examples is small, overfitting may actually be desirable since the lack of examples heightens the importance of each nuance that each example possesses.

2.3.2 Classifiers

The most basic form of inductive learning and a task that has been heavily researched is classifying examples. In this task, an example consists of a set of attributes or features and a symbol that represents the concept that the example represents. The attributes typically consist of fixed values with boolean or ordinal types. The goal of the learning algorithm is to predict what concept symbol corresponds to a given set of attribute values. For instance, the decision tree algorithm ID3 [82] builds a tree structure using the attributes as choice points. Gamut uses a version of ID3 to build its decision trees (see Section 6.6). There are many classifier algorithms including Mitchell’s concept space [65], first-order logic learners such as FOIL [83] and PRISM [13], neural networks [63], etc.

Each algorithm has different advantages that depend on the domain in which the algorithm is applied. For instance, neural networks are best applied in situations where one possesses a large number of noisy examples. On the other hand, a concept space could be used when there are fewer examples and the format of the solution has a restricted form.

Classifier algorithms are used in Gamut to build conditional statements. In a condition, the system must construct a predicate that chooses which way the code branches. A classification algorithm can combine attributes based on the properties of objects involved in the branch to decide whether the code takes a branch or not. In most AI systems, determining the attributes for the domain is performed ahead of time by hand. That is, the algorithm designer selects the information that will be stored in the examples and the algorithm finds the pattern within that information. Gamut, however, must not only determine the pattern in the set of attributes, but it must also create the attributes in the first place. Gamut’s algorithm for creating attributes is described in Section 6.6.3.

2.3.3 Learning Procedures

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

Commonly, plan recognition is used to model some aspect of human cognition. For instance, speech recognition or dialog modeling use plan recognition principles to understand spoken language. These systems tend to be specialized to their domain to such a high degree as to difficult to apply to Gamut's domain. On the other hand, simpler systems, such as Van Lehn's SIERRA [93] which models how a student learns to perform subtraction, are more useful. Van Lehn argues that a student learns multiplication by learning and using the special annotations that show where work is performed. By crossing out digits and writing carries, a teacher shows the student annotations that make the problem tractable. Van Lehn call this the "show work" principle. Gamut uses the same concepts for its guide objects and hint highlighting (see Section 4.1.2 and Section 4.2.2.1). The developer uses these annotations to demonstrate the work that the system must perform.

Lutz's Pascal tutoring system, PROUST [52], uses plan recognition to help a student examine an incorrect Pascal program and determine what is missing or mistaken. Lutz's system could disassemble a program into a network of control structures whose nodes represent the conditions that determine which path is followed. By analyzing the structure and comparing it to known programming idioms, PROUST could determine what student programs are meant to do and find errors. Gamut has a similar ability to structure programs so that code fragments can be easily compared and analyzed. One advantage Gamut has over PROUST is that the system is writing its own code so the code can be made more amenable to computation.

Gamut uses plan recognition differently than these other systems. Instead of trying to model the developer's game, Gamut uses plan recognition techniques to compare the developer's latest example with the previously demonstrated behavior. Gamut might conceivably use plan recognition to learn higher-level game behaviors. For instance, it might be able to learn that the application is a two-player game. However, inferring higher-level behaviors was not explored in this research.

2.4 Summary

This section discussed three areas that are related to the work in this thesis. First, it discussed previous programming-by-demonstration systems such as Marquise and DEMO II. Gamut borrows some of the techniques from these systems such as Marquise's mouse input icons and DEMO II's guide object techniques. Next, it discussed common tools used to build applications. Gamut's graphical editor uses the direct manipulation techniques found in user interface builders and prototyping tools. Finally, the last section discussed machine learning techniques from Artificial Intelligence. Gamut's inferencing algorithms are based on some of these machine learning techniques. In particular, Gamut uses a plan recognition algorithm to revise behaviors and it uses a decision tree algorithm to automatically infer conditions.

Chapter 3: An Illustrative Example

In this chapter, a small game will be assembled in order to provide a better picture of what sorts of applications Gamut can be used to build. This game will contain several behaviors to control a player-directed character and an autonomous “monster” object. These objects will have interactions with one another and with objects on the game board.

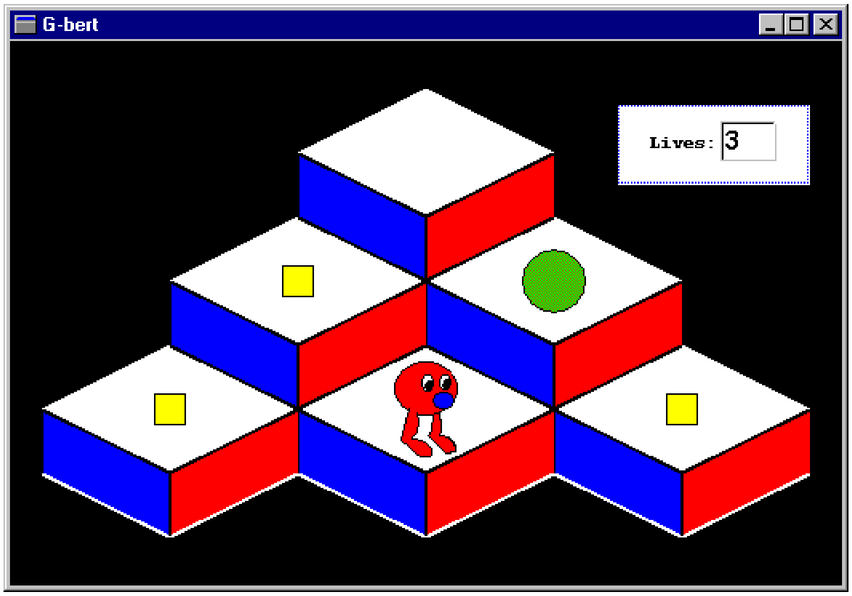


Figure 3.1: A screen shot of the complete G-bert game. The character jumps around on the cubes under player control picking up the square markers. The ball object bounces down from the top and can hit G-bert.

3.1 The Game

The game is called “G-bert” and it is derived from the game Q*bert that was created by Gottlieb in the early 1980’s. The game is shown in Figure 3.1. The player controls the character (named G-bert) who bounces around on the tops of a pyramid of blocks. There are six blocks. G-bert starts on the topmost block and is only allowed to jump to adjacent ones. The player moves G-bert by clicking on the block where G-bert should go. If the player clicks on a block that is not adjacent or clicks somewhere else on the screen, G-bert will not move. Yellow markers are located on all blocks except the top. When G-bert lands on a marker, he collects it. When G-bert collects all five markers, he wins the game, but there is also danger in the land of G-bert. A green ball falls down from the top of the pyramid and moves randomly down the blocks and off the bottom of the screen. Every so often, another green ball appears and if it hits G-bert, G-bert will lose a life and begin again at the top of the pyramid. If G-bert loses three lives, the game is over and the player loses.

3.2 The Background

The first thing to do is draw the background pyramid of cubes as well as a bitmap for G-bert. Gamut does not currently possess a bitmap editor so the developer needs to use an external program. The pyramid consists of eight kinds of tiles. Four tiles define the interior of a cube and the other four define the outside edges (see Figure 3.2). The tiles are sized to be multiples of ten pixels to coincide with Gamut's default gridding size. Similarly, G-bert's size is also drawn as an even multiple of ten pixels so that the developer can align objects on G-bert's center. The developer loads each of the eight bitmaps using Gamut's "Load Image" menu command and arranges them on the screen making copies of the bitmaps as needed. At this time, the developer also adjusts the size of the window frame and uses the property editor to change the window's title to read "G-bert."

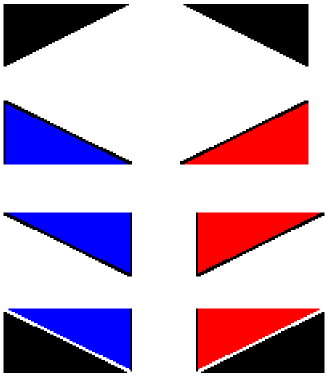


Figure 3.2: The eight tiles used to create the pyramid background. The tiles are repeated to create the visual effect of multiple cubes.

3.3 The Monster

At first, the developer will concentrate on making the bouncing green ball work and worry about G-bert later. The developer wants a green ball to start from the top and bounce its way down the pyramid. The developer creates a green ball using a circle and places it on the topmost cube. The system will have to be able to detect where the tops of all the blocks are and it will need a way to decide which direction the ball should fall. First, the developer creates objects to represent the various things the ball will be able to do, then the developer demonstrates how the ball behaves.

3.3.1 Creating Guide Objects For the Ball

The developer uses arrow lines as onscreen guide objects (see Section 4.1.2.1) to represent the path the ball can take. The developer draws six arrow lines on the board using guide object colors. To distinguish directions, the arrows pointing left are drawn cyan and the ones pointing right are drawn magenta. The player will not see these arrows because guide objects can be made invisible. The system will still know that the lines are there but the player does not see them. The ball also

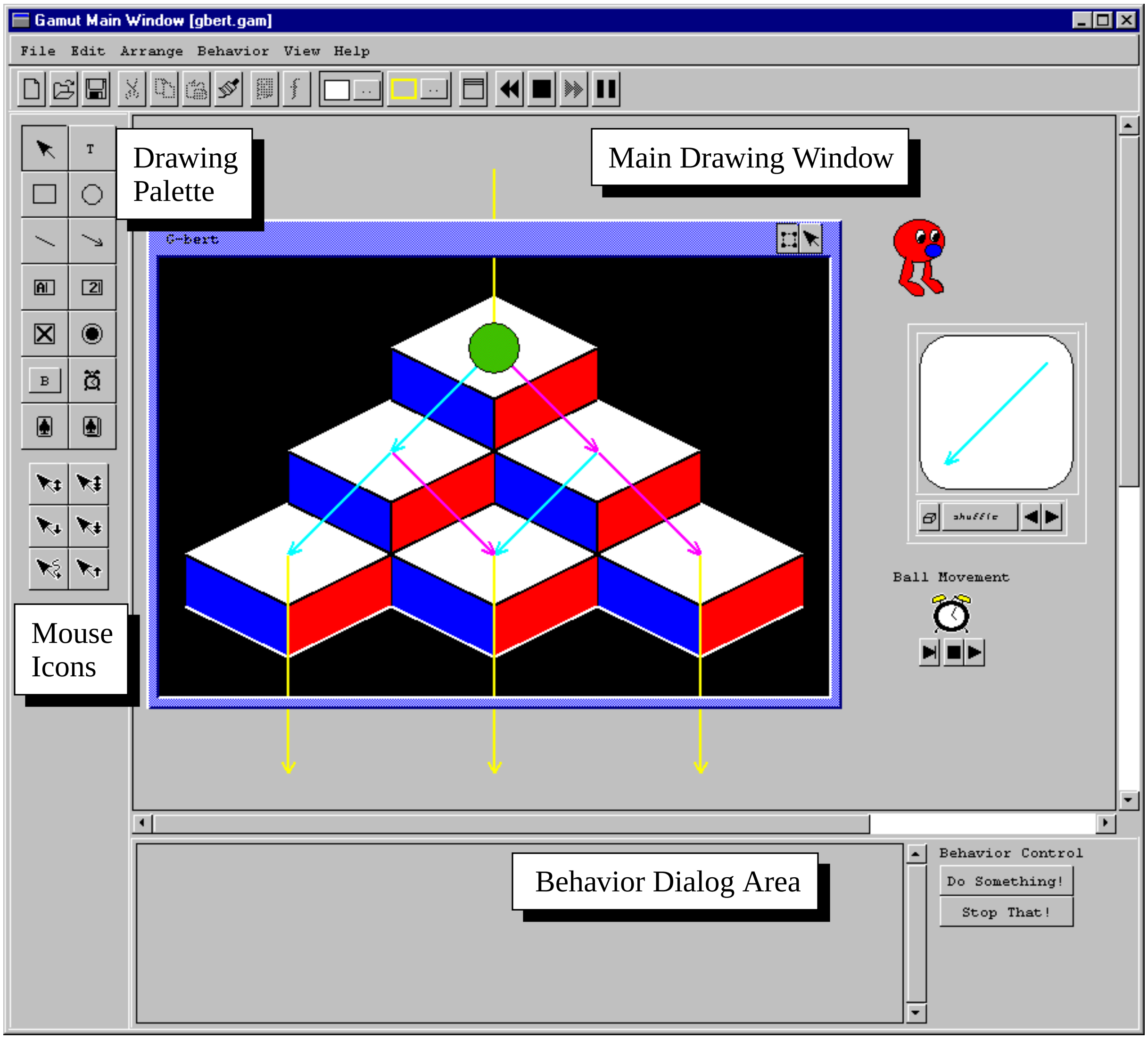


Figure 3.3: Initial arrangement of the background showing the guide objects that the ball follows as well as the deck and timer widgets.

needs to be able to fall in from the top of the screen and fall off the bottom of the screen. So, the developer draws four more arrow-line guide objects: one coming in from the top and three dropping down off the bottom of the pyramid. To distinguish the new arrows from the ones in the pyramid, the developer makes the entry/exit arrows yellow. The arrows can be seen in Figure 3.3.

The developer has to create an object that will select which way the ball falls at any given time. The ball falls randomly so the developer creates a deck widget and places it offscreen beside the window. The ball has two options: it can fall left or right, so the developer creates two card widgets: one for each direction. On one card, the developer draws a cyan arrow and on the other is drawn a magenta arrow. Figure 3.4 shows the deck editor displaying the cards. The developer has also “pre-highlighted” the arrow in each card which is indicated by an orange outline. Pre-highlighting an object in a card lets the system know that a particular object is important. To pre-highlight the arrow, the developer selects the arrow and uses a command in the card editor’s menu.

Later, whenever the developer highlights a card to give the system a hint, pre-highlighted objects inside the card will also be highlighted as hints (see Section 4.3.1.1). Highlighting objects as hints will occur later while the developer is demonstrating behavior. This is a shortcut that allows objects inside a card to become highlighted without having to open up the card editor.

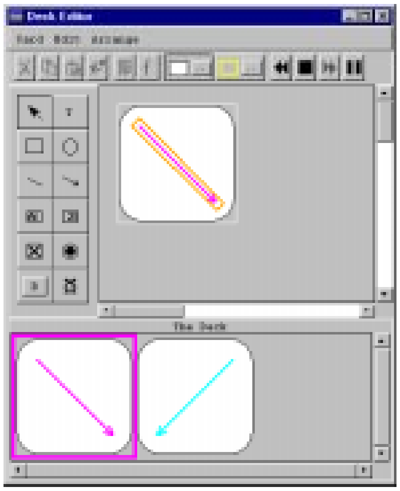
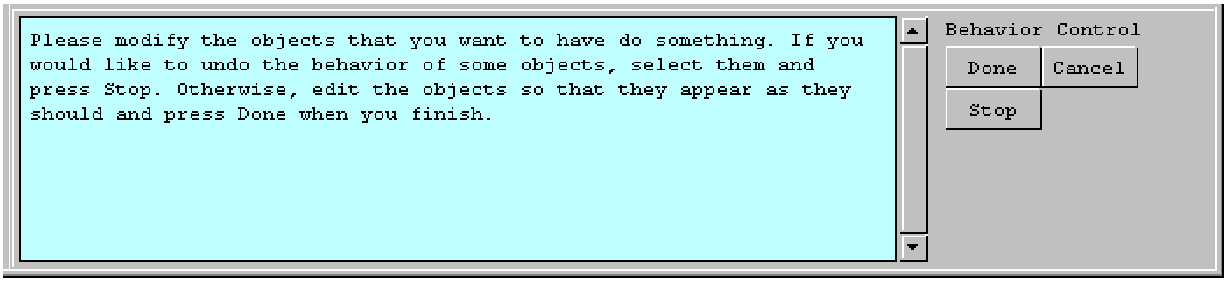


Figure 3.4: The developer makes a deck of two cards to represent the directions the ball can move. The line on one card is cyan to indicate moving to the right and the other is magenta to indicate moving to the left. The box around arrow line on the card shows that it is "pre-highlighted." The look of pre-highlighted objects is designed to look similar to normally highlighted objects as in Figure 3.6.

The game needs something that will cause the ball to move automatically. The developer creates a timer widget and places it below the deck. (The label above the timer in Figure 3.3 is actually just a text object that is created like any other object. The developer is free to add comments to his or her offscreen guide objects just as one can comment code.) The board now looks like Figure 3.3. The arrow lines show which direction the ball is allowed to move, the deck shows which direction the ball will take and the timer causes the ball to move.

3.3.2 Demonstrating The Ball

Once the pieces are in place, the developer begins demonstrating the ball's behavior. Gamut's technique for demonstrating behavior is called "nudges" (see Section 4.2.1). First, the developer produces the event that is supposed to make the behavior occur. In this case, the timer should cause the ball to move, so the developer pushes the timer's "tick" button that is labeled with a triangle pointing into a line. Producing the event should cause the system to respond with the appropriate behavior. When the system does not perform the right behavior, the developer "nudges" the system in order to correct it. In this case, the system has not yet been taught any behavior so it does nothing when the tick button is pushed. There are two kinds of nudges. The first, called Do Something, tells the system to perform some new action in response to the event. There is also the Stop That nudge that tells the system not to perform an action. The developer wants the ball to move, so Do Something is pushed. Pushing a nudge button switches Gamut into Response mode (see Section 4.2.2) where it becomes ready to accept a new example. The system produces the dialog shown in Figure 3.5 asking the developer what to do.



Please modify the objects that you want to have do something. If you would like to undo the behavior of some objects, select them and press Stop. Otherwise, edit the objects so that they appear as they should and press Done when you finish.

Figure 3.5: Initial system dialog after the developer has pushed Do Something. The dialog asks the developer to finish the example and push Done when complete.

The developer wants several things to happen. First, the deck must be shuffled to determine which path the ball should take. The developer pushes the “shuffle” button on the deck. The deck randomly picks the cyan card, so the developer moves the ball one step along the cyan path. When the developer makes these changes, Gamut draws “temporal ghost” objects on the screen to show what has happened (see Figure 3.6). A ghost object of the ball appears on the top cube showing where the ball used to be and a ghost of the deck appears behind the real deck to show that its has been shuffled.

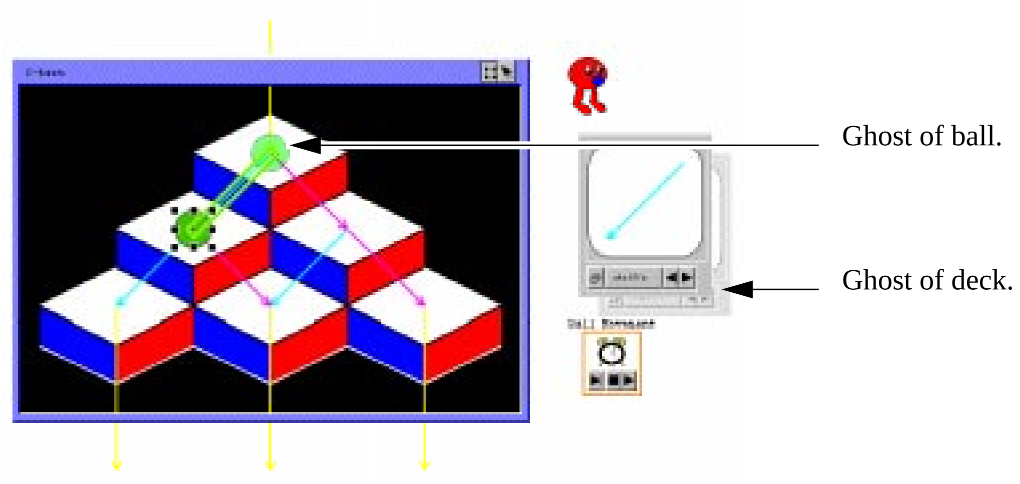


Figure 3.6: When the developer shuffles the deck and moves the ball, Gamut places ghost objects into the scene to show how the interface has changed. The developer has also highlighted an arrow line as a hint. This is indicated by a box around the line.

The developer can highlight objects on the screen to give the system hints (see Section 4.2.2.1). To highlight an object, the developer clicks the rightmost mouse button on the desired object. Gamut will try to use highlighted objects when it generates a description for the behavior. The developer highlights the arrow that the ball followed and the deck. Gamut is able to match the color of the arrow on the board with the color of the arrow in the top card of the deck and creates a behavior where the ball moves to follow the arrow whose color matches the color of the object on the top card. The developer pushes the “Done” button because this part of the behavior is finished.

This first inferred behavior is fairly complex and the system creates a behavior that is close to what is desired for the ball’s movement. It was made possible by the developer’s highlighting which allowed the system to incorporate objects that it would not have known otherwise. The behavior that Gamut forms is comprised of Move/Grow action that moves the ball and a Shuffle action which shuffles the deck. The position where the ball moves is described by a Connect description which tells the system how to find the arrow line that the ball is following. The Connect description is embedded in a Select Object and an Align description which further refine how the proper arrow is selected and chooses the end point of the arrow as the position for the ball. This behavior is pictured in Figure 3.7. This diagram is fairly complicated even though it is not complete. The boxes represent the parts of Gamut's code: the cross-hatched box (1) is an event and the thick-lined boxes (2, 3) represent actions. All the other boxes (4, 5, 6, 7) are descriptions. The parts of Gamut's code will be described in more detail in Chapter 5 where the inferencing system is introduced. At this point, it is not necessary to know how Gamut works in detail, but it is worthwhile to see the format in which Gamut is recording the developer's examples.

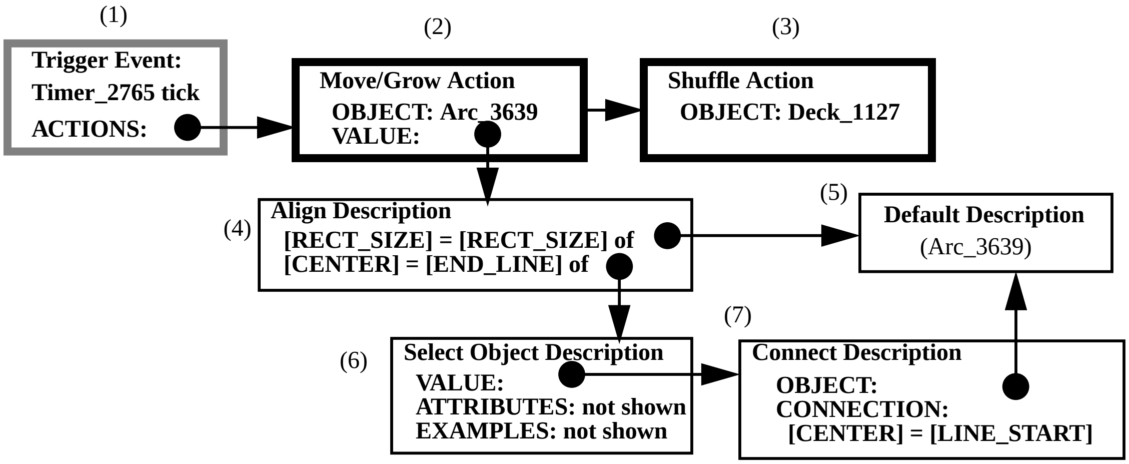


Figure 3.7: Diagram of the behavior Gamut created on the first example. Gamut was able to create a behavior this detailed because the developer highlighted objects in the scene.

The generated behavior defines that when the tick button (1) is pushed (or the timer is turned on), the system will evaluate the descriptions (4, 5, 6, 7) and perform the Move/Grow action (2) to move the ball and perform the Shuffle action (3) to shuffle the deck. The Connect description (7) finds appropriate arrow lines by picking objects whose start points connect to the center of the ball (5). This code was created based on the developer highlighting the arrow line. The Select Object description (6) contains an attribute (that is not shown) that matches the color of a the arrow lines picked by the Connect description (7) with the color of the arrow in the top card of the deck. This code was created because the developer highlighted the deck which also highlighted the top card of the deck which in turn highlighted the pre-highlighted item in the top card. So, by using highlighted objects as hints, Gamut can make quite powerful inferences with only a single example.

The system still needs to be taught about the yellow lines that the ball should follow regardless of the card showing in the deck. The developer places the ball at the top of the upper yellow arrow line as in Figure 3.8 and pushes the tick button. The ball does nothing. Since the arrow is neither magenta or cyan, the Select Object description (6) will fail. This in turn causes the Move/Grow action (2) to fail. The developer pushes Do Something and moves the ball manually. Gamut will accept this example without question because it finds that it can create a new predicate for the Select Object description (6) that adds a test for the color yellow to the set of predicates (see Section 6.6.5). However, this new example will affect the Select Object's behavior for other objects, too. Gamut does not know what the developer would want when the deck picks the magenta card. Also, the top card no longer matches the color of the line so Gamut will temporarily suspend the attribute that matches the color of the arrows on the screen with the arrows in the deck. Suspending and reusing attributes is handled automatically by the decision tree learning algorithm which must select which attributes among the Select Object's set it should use (see Section 6.6.1).

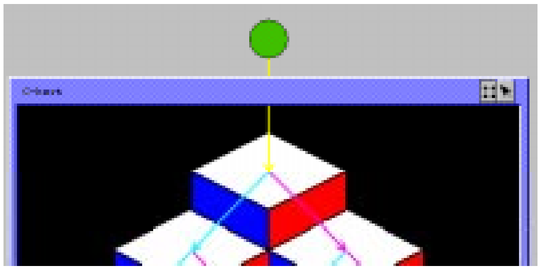


Figure 3.8: In the second example, the developer places the ball at the starting point of a yellow arrow line.

When the ball is in the pyramid again and the deck shows magenta, the ball incorrectly follows the cyan line. This is because the previous demonstration using a yellow line has disrupted the decision tree and now more examples are required to fix it. The developer selects the ball and pushes "Stop That." The Stop That nudge is used to show negative examples. It will automatically undo the actions performed on objects selected by the developer. In this case, the developer selected the ball so the system will undo the Move/Grow action that it performed on the ball and will place it back in the position where it started (see Figure 3.9). The developer wants the ball to follow the magenta line so the ball is moved accordingly. When the developer presses Done, the system will also accept this example without question. It simply adds a new entry to Select Object's decision tree database (see Section 6.6.6).

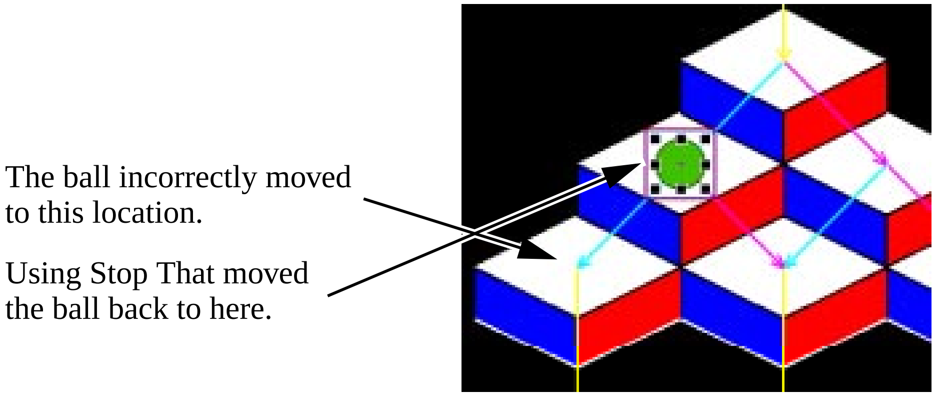

The ball incorrectly moved to this location.
Using Stop That moved the ball back to here.

Figure 3.9: When the ball moves the wrong way, the developer selects it and presses Stop That. This will undo the actions the behavior performed on the selected objects bringing the ball back to its initial point.

One more example will complete this part of the behavior. When the deck selects cyan, the ball will not follow cyan lines. This occurs because the original example in the decision tree database did not contain an entry for the new predicate that referred to the color yellow. For this example, the developer pushes Do Something, moves the ball to follow a cyan line, and presses Done. The system fills in the last bit of knowledge into the database (see Section 6.6.7.3) and the ball will move as it should.

3.4 G-bert

After working on the ball's behavior for a while, the developer switches attention to the G-bert character. G-bert does not move like the green ball, so the developer will need to demonstrate its behavior separately. The player moves G-bert by clicking the mouse on the top of an adjacent cube whereas the ball moves randomly down a fixed path. The developer will need to add guide objects to the game to denote where the tops of the cubes are located that will show where the player can click. G-bert also clears rectangular markers as he moves. Thus, the developer will need to add the markers as well as a means to track them.

3.4.1 Moving G-bert

Though the pyramid is already composed of graphical objects, the top of each cube consists of four tiles (see Figure 3.2). It would be easier for Gamut to recognize the tops of the cubes if these positions were more concisely represented. The developer draws an ellipse around the top of each cube as shown in Figure 3.10. Currently, Gamut does not distinguish a difference between ellipsoid and rectangular objects, but the ellipse is more visually appealing to the developer because it more closely matches the shape of the cube's top. If Gamut supported a polygon object, the developer could make guide objects that matched the tops of the cubes exactly. Denoting which cube is connected to which is already handled by the arrow lines that the developer used for moving the ball objects. The cyan and magenta lines connect all adjacent destinations.

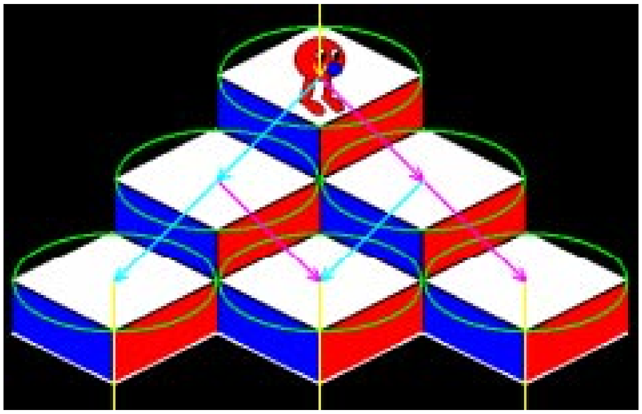


Figure 3.10: The developer marks the top of each cube by drawing an ellipse around each. This will give the player something to click on that covers the entire top of a cube.

The developer positions G-bert at the top of the topmost cube. The ball has been moved off the interface for convenience. Gamut will not be confused that the ball is moved off the board because the timer event that controls the ball's movement will not be used. To generate a player mouse event, the developer uses the mouse icons (see Section 4.3.5) that are located below the drawing palette in Figure 3.3 (see Figure 3.11 for a close-up view). The developer first selects the single click icon (which is an arrow with two heads, one pointing down and one pointing up, as well as a big arrow that represents the mouse). Then, the developer clicks on one of the ellipses below G-bert as shown in Figure 3.12. The system shows a click icon at the position where the developer dropped it. Dropping the icon onto the window acts the same way as if the player had just used the mouse to generate that event. The developer shows the response by using Do Something and moving G-bert to the new position. Let us assume that the developer is not highlighting objects for a while so that Gamut will eventually have to ask a question. Gamut will accept this first example without question and assume that the developer wants G-bert to move to a constant location.

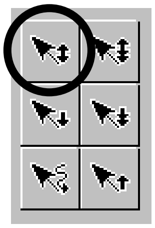


Figure 3.11: The player input icon palette. These buttons are used to create mouse icons within the application interface. The click icon is circled.

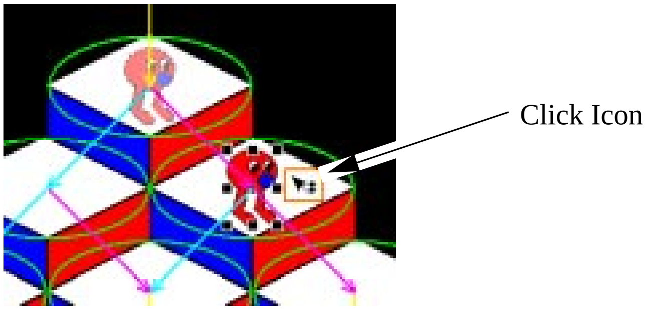


Figure 3.12: The developer drops a click icon onto the place where G-bert is supposed to move and moves G-bert accordingly.

The developer drops another click icon and nothing happens. The developer pushes Do Something to create a new example. In Response mode, the developer moves G-bert to the bottom of the pyramid (see Figure 3.13). Gamut will accept the second example without question, too. Gamut is able to see that only the x,y position of the G-bert object is changing (and not its size), so it can search for objects that would describe the x,y position. The ellipses, the background tiles, and the arrow lines all would be candidates to describe G-bert's new position. The system also automatically highlights G-bert's ghost position. The Move/Grow action that affects G-bert automatically highlights the ghost of the object it affects when it tries to discover a description for its value. This feature was added because we found in testing that developers were surprisingly reluctant to highlight ghost objects (see Section 7.2.3.1). Since the ghost of G-bert coincides with the starting point of one of the arrow lines, Gamut will infer that G-bert is moving to the end point of an arrow that is connected to G-bert's initial location.

On the third example, the developer moves G-bert against the grain, so to speak, by moving G-bert in the opposite direction of the arrow that connects the two ellipses (from position 2 to 3 in

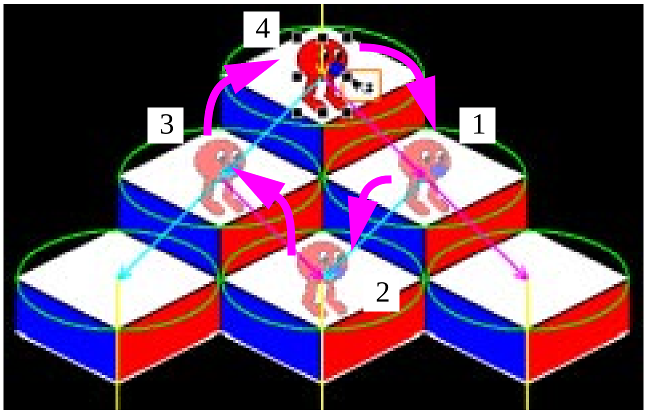


Figure 3.13: The developer keeps jumping G-bert around the screen. Sometimes G-bert does not move at all and sometimes it moves the wrong way. For each example, the developer nudges the system with Do Something or Stop That and corrects G-bert's position.

Figure 3.13). Gamut will accept this example without question as well by concluding that it needs to ignore the direction of the arrow lines. If the developer had used undirected lines as guide objects, the system would have inferred this on the second example. Of course, with undirected lines, demonstrating the ball's movement would be much harder.

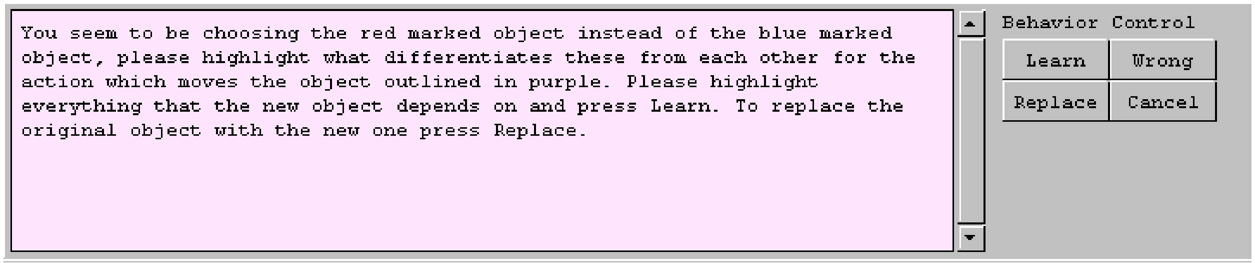
On the fourth example, when the developer moves G-bert back to the top of the pyramid, Gamut finally cracks and asks the developer the question shown in Figure 3.14. The system wants to know which line the G-bert object is following. Gamut's question dialogs give the developer three options (see Section 4.2.3.2). The first option is called "Learn" where the developer wants the system to learn a new concept. This is accomplished by highlighting the objects that are related to the concept and pushing the Learn button. The second option is "Replace" which is used to update old concepts with new ones. Pushing the Replace button causes the system to replace the description where the conflict was discovered with a new one. The last option is called "Wrong" and it is used when Gamut's heuristics produce an improper question. Since Gamut uses heuristics to match new examples with the original behavior, it can sometimes make mistakes. The Wrong button causes the system to skip the erroneous question in favor of another.

The developer highlights the ellipse that G-bert lands on as well as the mouse icon to show how the ellipse is selected. This is enough information for Gamut to build predicates for a Select Object description that describes the correct arrow line. By highlighting the appropriate objects in advance, this behavior could have been demonstrated with one less example.

3.4.2 Picking Up the Markers

The developer draws five yellow rectangles in each square that G-bert has to visit as shown in Figure 3.15. The developer also wants the system to be able to count how many of these markers remain so that the winning condition of the game may be demonstrated. To represent the number of markers, the developer adds a number box and labels it Markers Remaining and sets it to five.

The developer begins demonstrating by dropping a click icon onto a location with a marker. The system will move G-bert as it should. The developer then uses Do Something and shows that the marker should be deleted. The developer also sets the number box to four and highlights the G-



You seem to be choosing the red marked object instead of the blue marked object, please highlight what differentiates these from each other for the action which moves the object outlined in purple. Please highlight everything that the new object depends on and press Learn. To replace the original object with the new one press Replace.

Figure 3.14: The question that Gamut asks about G-bert's position. It is selecting between two arrow line guide objects.

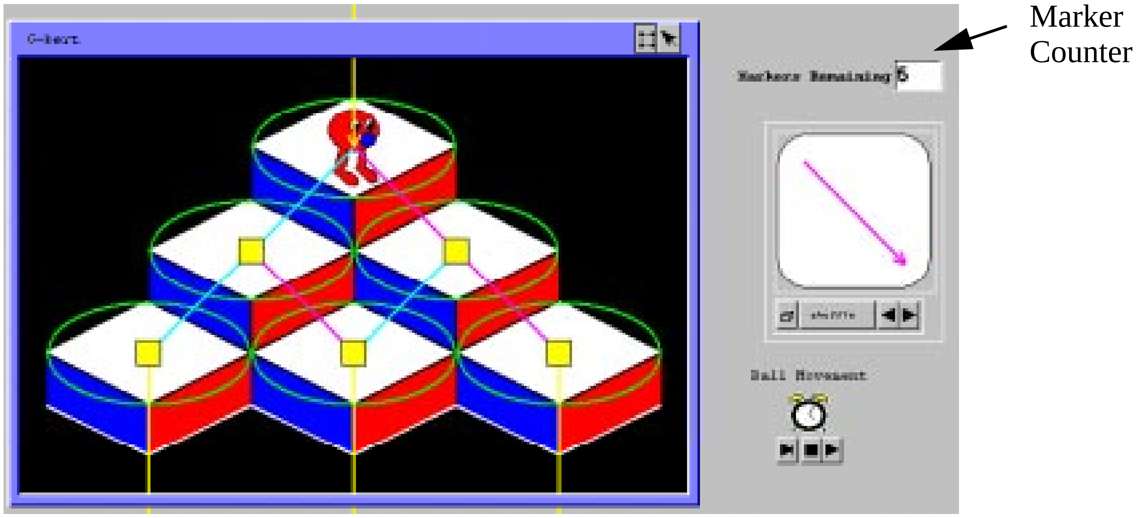
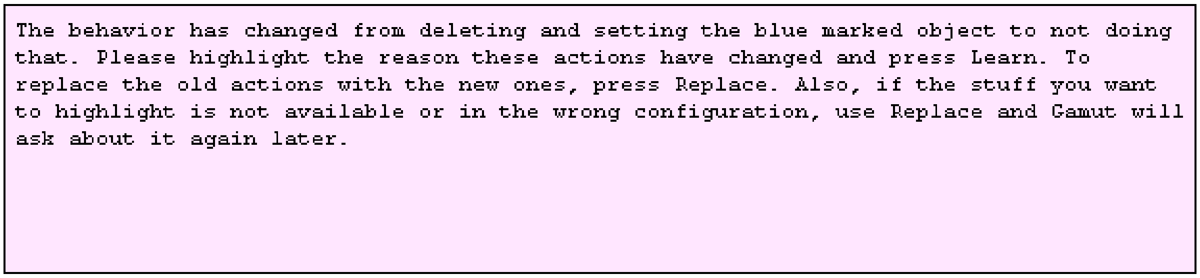


Figure 3.15: The developer adds the yellow marker squares to the interface and a counter to count how many markers remain.



The behavior has changed from deleting and setting the blue marked object to not doing that. Please highlight the reason these actions have changed and press Learn. To replace the old actions with the new ones, press Replace. Also, if the stuff you want to highlight is not available or in the wrong configuration, use Replace and Gamut will ask about it again later.

Figure 3.16: A question Gamut generates to ask about a conditional statement. One of the ways to answer this question is to use Replace when the objects that need to be highlighted are not in the correct configuration.

bert object to show that it mattered. The system will correctly describe which marker is to be deleted and it will make a Set Property action for the number operation that sets the number box's value to be one lower than it was previously.

When the developer clicks on another position with a marker, the marker disappears and the number changes correctly, but when the developer moves G-bert to a position where the marker is already gone, the counter decreases even though no marker was deleted. The developer selects the counter and pushes Stop That. This restores the counter’s value. When pushing Done, the system asks why the counter did not change. The developer realizes that the counter changes when G-bert lands on a marker, but this is a case where there is no marker; thus, there is no relationship to highlight. At the bottom of Gamut’s question shown in Figure 3.16, Gamut gives the option to use Replace in order to defer the question for later. The developer decides that this is the proper response and pushes Replace. Though this action may seem confusing, in fact, volunteers who tested Gamut’s interface for a user study were able to use this technique correctly to build behaviors (see Section 7.2.4.1). Using Replace to defer a question allows Gamut to escape from situations where the needed relationship only exists in one branch of a conditional statement but the developer is currently demonstrating the other branch.

The next time the developer moves G-bert onto a marker, the system will correctly delete the marker but will leave the counter alone. This is because the developer deferred the inferencing of the conditional statement that affects when the counter changes. The developer uses Do Something to make the counter decrease and after pushing Done, the system asks the same question it did last time. This time, the developer has a good response and can highlight the ghost of the deleted marker and G-bert to show that the counter decreases when G-bert lands on a marker.

3.5 Extending the Monster’s Behavior

At this point, the developer has a monster that starts from offscreen, falls randomly down the pyramid and off the bottom of the screen. Once at the bottom, the ball just sits there. The developer would rather have new balls come in from the top at a fixed rate and have old balls that fall to the bottom go away. Gamut allows the developer to revise any behavior simply by demonstrating more examples. In this section, the developer will make the system create new balls and delete old ones.

3.5.1 Creating Balls at the Top

The developer creates a new timer that will cause the new balls to be created above the pyramid. To train the timer, the developer pushes the tick button, pushes Do Something, creates an appropriate ball at the top of the yellow arrow (see Figure 3.17), and pushes Done. When Gamut creates an object via a behavior, it draws a letter “C” in the center of the object (see Section 4.4.1). The mark distinguishes created objects from other objects in the scene. The C will go away when the developer performs an event which causes a new behavior to occur. For instance, the developer might drop a click icon and cause G-bert to move which would make the C disappear.

3.5.2 Making the Created Balls Move

The developer wants every ball on the screen to move when the original, movement timer ticks. Currently, the timer only knows to move the first ball and not the others. The developer places two balls on the screen in positions where they should both move. The developer also creates a large guide object rectangle in the background of the window which is large enough to enclose the whole board (see Figure 3.18). This new guide object will represent the space where balls move. Gamut needs this object in order to describe all of the balls. The system will look for objects inside the rectangle as its starting point to find the balls in the application. This large rectangle is only necessary because Gamut currently lacks a description that returns all the objects in the application window. Such a description would be fairly easy to add for future versions. In the

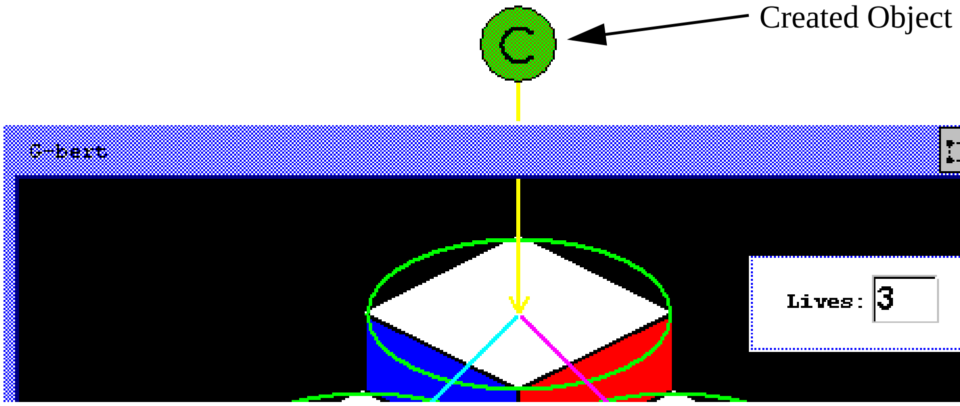


Figure 3.17: The developer trains a second timer to create a ball at the top of the pyramid. The system draws a “C” over the center of the new object to show that it has just been created.

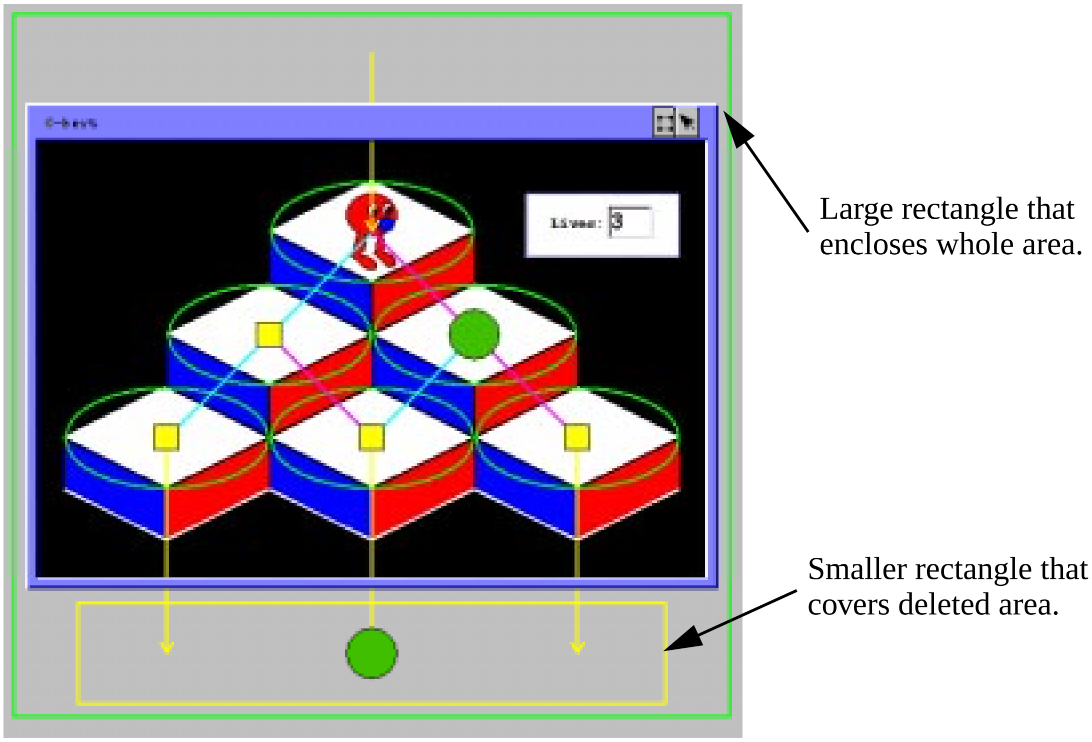
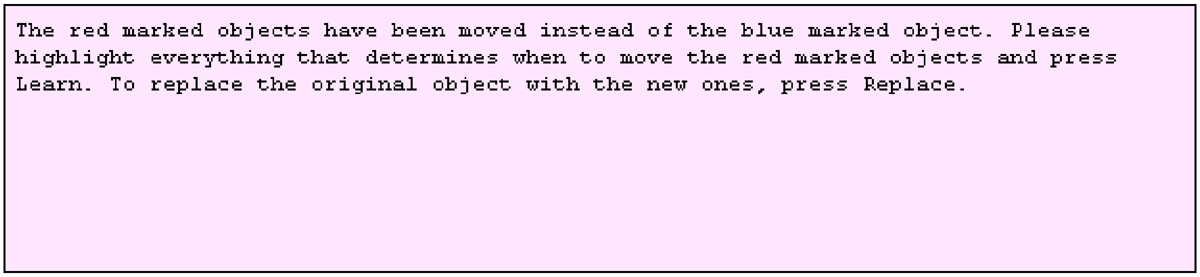

Q-bert
Lives: 3
Large rectangle that encloses whole area.
Smaller rectangle that covers deleted area.

Figure 3.18: The developer uses rectangles to enclose portions of the interface where special behaviors occur. The large rectangle shows the area in which the ball objects are allowed to move. The smaller rectangle along the bottom shows where balls become deleted.

future, it might also be possible to form descriptions that list all the objects created by the behavior that creates balls at the top.

The developer pushes the movement timer’s tick button. The system only moves the original ball object, and the developer uses Do Something to demonstrate that the other ball moves as well. In this case, both balls are following the same color of line. If the developer wanted the balls to move in independent directions, there would need to be a different deck associated with each ball. The developer pushes Done but this time does not highlight any objects. If the developer had highlighted the background rectangle, the system would use it to form the correct description, but by not highlighting, the system is forced to ask the developer a question.

The system's search process is able to match the movement of both circles so it knows that it is the description in the Move/Grow's object parameter that has changed. The system does not find a suitable description that can describe the set of both objects so it asks the question shown in Figure 3.19. The original ball object is the one marked in blue whereas both ball objects are marked in red. The developer responds to the question by highlighting the large rectangle and pressing Learn. This allows Gamut to form a description for the set of ball objects using the large rectangle as a point of reference. The description consists of a Connect within a Select Object like the description created in the very first example. The Connect description finds all objects that are inside the rectangle and the Select Object description picks the objects which are circles.



The red marked objects have been moved instead of the blue marked object. Please highlight everything that determines when to move the red marked objects and press Learn. To replace the original object with the new ones, press Replace.

Figure 3.19: The system asks the developer this question in order to describe the set of two balls that moved instead of the one ball that moved in the past.

3.5.3 Delete Balls at the Bottom

To simplify matters, the developer will reuse the timer that creates balls in order to delete balls from the bottom of the pyramid. The developer will use the same process used in the last example for the system to describe the set of balls that get deleted. First, the developer draws a guide object rectangle around the bottom of all three of the lower arrow lines. Then, the developer puts two ball objects at the ends of two of the lines. The two rectangles that the developer uses to divide the screen are shown in Figure 3.18.

When the developer pushes the tick button on the timer, the system will create a new ball at the top of the pyramid as it should. The developer pushes Do Something and deletes the two balls at the bottom of the pyramid. When the developer deletes the objects, the system draws a letter "D" in the place where the object used to be (see Section 4.4.1). The "D" is selectable so that the developer can use it to perform Stop That if necessary

Before pushing Done, the developer highlights the rectangle that surrounds the objects to be deleted. This will create the appropriate description so the new Delete action will delete all balls in the rectangle. The behavior for the timer that creates and deletes balls is now complete.

3.6 Winning and Losing the Game

So far the developer has demonstrated behavior to make all the needed objects mobile, but the game is still not quite ready to play. The developer must now show the winning and losing conditions and make the game announce when these conditions occur.

3.6.1 Adding The Lives Counter

G-bert, like many video games, keeps track of the number of “lives” the player has remaining. When the monsters successfully attack the player’s character, the number of lives is reduced by one. When the player runs out of lives the game is over. The developer decides to use a counter to represent the number of remaining lives. A counter object is placed in the upper right corner of the screen where it can be seen by the player. The developer also draws a rectangle behind the counter in order to give the label contrast. The counter can be seen in Figure 3.1.

3.6.2 Getting Bounced On

The developer needs to demonstrate that G-bert can be hit by the ball. The developer sets up a situation where a ball is positioned right above G-bert and pushes the ball movement timer’s tick button. The ball has a 50-50 shot at hitting G-bert. We have observed that most developers will let the deck shuffle and take what they get. Eventually the ball will hit G-bert and the developer can demonstrate the desired effect. Gamut actually allows the developer to set the deck manually, though. If the developer changes the value of the top deck item in Response mode after the deck is shuffled, Gamut will ignore the deck modification and assume that the deck actually produced that value during the shuffle. The developer can then rearrange the interface to show what happens with the deck in that arrangement. The developer makes the ball hit G-bert and demonstrates that G-bert moves to the top cube of the pyramid and that the “lives” counter is decreased by one.

The developer resets the interface to show another example by using “back” button on the history controls (see Section 4.4.2). Gamut provides a history capability that allows the developer to move back and forth through all the behaviors that have occurred in the application. Gamut treats a behavior that has just been demonstrated as though it occurred in the game normally. This allows the developer to be able to use the back button as though the last demonstration were any other event.

When the developer continues testing the game, it is found that when the ball misses G-bert, the G-bert object and lives counter are still being modified. The developer fixes this problem using Stop That. Gamut will want to know why the behavior has changed and the developer uses Replace to defer the question. In the third example, the developer gets the ball to strike G-bert again and redemonstrates the movement and lives counter effects. When Gamut asks a question this time, the developer highlights the ball and the ghost of G-bert that overlap. This will make Gamut produce the correct behavior.

3.6.3 Announcing Win or Lose

The developer has arranged the game so that counters detect whether the player has won or lost the game. If the Markers Remaining count reaches zero, the player should win. If the lives counter reaches zero, the player should lose. To indicate winning and losing, the developer creates two signs that are initially placed offscreen so the player cannot see them. One sign reads, “You Win!,” the other reads “Game Over.”

The developer decides to demonstrate winning first. The developer clears all the markers but one and verifies that the Markers Remaining counter reads one. Then the developer uses the player input icon to move G-bert and clear the last marker. The developer uses Do Something and moves the winning sign into place as in Figure 3.20. Gamut is set up to create new actions in the most specific condition that it can. In this case, it will store the new Move/Grow action into the same branch as the Set Property action that affects the counter. Using the history control's back button, the developer retraces back to a point where two markers remain. When G-bert eats the marker, the system puts out the You Win! sign. The developer stops this using Stop That and uses Replace to skip the question. Finally, the developer makes G-bert eat the last marker again, demonstrates moving the sign back out using Do Something and highlights the Lives Remaining counter to tell Gamut why it happened.

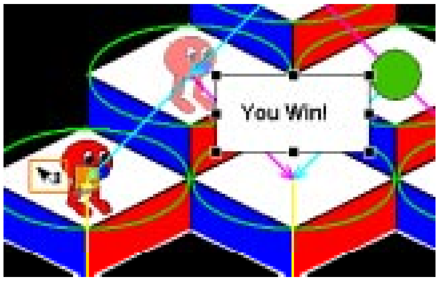


Figure 3.20: The developer moves the You Win! sign out over the playing field when G-bert collects the last marker. The same method is applied to move the Game Over sign when the player runs out of lives.

Demonstrating the Game Over sign is similar. Instead of moving G-bert, the developer gets the ball to hit G-bert the requisite number of times. On the last example, the developer highlights the lives counter to show Gamut the final dependency.

There are still other behaviors that the developer could demonstrate to make this game more complete. For instance, instead of putting out a You Win! sign, the developer could have the game reset the board with new markers and have the player play another round. Also, the developer should probably demonstrate having the two timers stop running when the player loses. This could all be demonstrated using the same techniques illustrated in this example.

3.7 G-bert in the Usability Study

The G-bert game was one of the applications that was tested in Gamut's usability study (see Section 7.2.2.3). The participant who attempted the task was able to create the whole application in about three hours. (The test version did not have the creation and deletion of new balls, but otherwise it was virtually identical.) The ability for an intelligent person with no previous Gamut experience to be able to use the system to make such a complicated game suggests that Gamut's techniques are worth exploring further in other domains.

An experienced Gamut developer can build G-bert in about an hour with the majority of the time spent drawing and arranging the graphics. It takes about twenty examples (with appropriate highlighting) to build this game's behaviors which shows the efficiency of Gamut's inferencing techniques.

3.8 Summary

This section showed how a developer can make a relatively complex game using Gamut. The game consisted of multiple behaviors including an autonomous monster object, a player controlled character, and their various interactions. The game primarily uses moving actions in its behaviors but it also contains some manipulation of counters along with the creation and deletion of objects.

The key aspect of Gamut's techniques is that developers can create new behaviors as they are needed and do not have to demonstrate an entire behavior all at once. For instance, demonstrating the ball object's behavior had at least three distinct phases where first the ball was made to move, then the ball's behavior was generalized to make all ball objects move, and finally the behavior was made to affect the G-bert object. Furthermore, the various guide objects needed by the behaviors were created as needed. Gamut was not confused when the developer created new objects even when previous behaviors had already been demonstrated.

Chapter 4: Interaction Techniques

Gamut's interface is based on a standard graphical editor model. However, it incorporates a number of new widgets and interactions that make applications easier to demonstrate. Gamut supplies new widgets to aid in creating a game's internal data and to demonstrate more complex behaviors. Gamut also allows the developer to give the system hints that make the inferencing of complicated behaviors possible. Also, Gamut produces feedback both within the developer's drawing region and in separate dialog areas to help the developer manipulate objects that would not otherwise be available and to help solve conflicts that Gamut finds while it is inferring behaviors.

Gamut supports a new interaction technique called "nudges" for demonstrating behavior. It is loosely based on the way that people may correct the behavior of a child. There are two interactions called "Do Something" and "Stop That." The developer uses the Do Something interaction to demonstrate a new behavior and to augment an existing behavior. Stop That is used to correct a behavior that performs unwanted actions. The developer selects objects to be "stopped" and actions performed on those objects are undone. If left stopped, that example is considered a negative example (further described in Section 4.2). The nudges technique is enhanced by allowing the developer to give the system "hints." With hints, the developer can point out objects that are important to the demonstrated behavior. Unlike other systems, Gamut relies heavily upon hints for its inferencing.

Gamut allows the developer to draw special objects that make demonstrating more complicated game behaviors possible. Based on work performed by Fisher *et al* [26] and Maulsby [54], Gamut incorporates the ability to draw "guide objects." Guide objects are graphical objects that can be seen by the developer, but are made invisible to the player. Guide objects are used to illustrate important relationships between a game's components and to affect the behavior of objects in constructive ways.

Gamut also supports new abstractions called the card and deck widgets. These new widgets are based on a playing card metaphor like the standard Poker deck or the Chance deck in Monopoly. Cards and decks are useful widgets for grouping and arranging related data and can serve as a basis for intricate behaviors and for randomness.

Gamut also uses special objects and markers for feedback. For instance, Gamut represents the prior state of objects with "temporal ghosts" (or "ghost objects"). The system draws dimmed images of modified objects the way they looked in the past. This allows the developer to refer to the past state of the objects to use in hints. Other kinds of markers are used for pointing out objects used in Gamut's query dialogs and to mark objects that have just been created or deleted.

This chapter describes all of Gamut's interaction techniques. It begins by describing the graphical editor capabilities including drawing graphics such as guide objects. Next, it explains the nudges method for demonstrating behaviors. Finally, it closes with sections discussing the advanced widgets such as the card and deck objects as well as some of Gamut's other features used for editing.

4.1 Drawing

A tool for programming by demonstration must have an editor that fits the domain of programs that it can build. Gamut is designed to build graphical applications like board games hence its editor is a structured graphical drawing tool similar to MacDraw [19] or Visual Basic [60]. Gamut uses a custom-built editor rather than using an existing editor to provide the system full control over its interface. Since Gamut provides feedback directly in the drawing windows, it would be difficult to implement within another program's interface that does not provide a sufficient degree of customization. Similarly, having a custom interface allows the developer's input to be gathered directly from the interface in a form amenable to Gamut's inferencing. It is unlikely that another drawing package would provide Gamut enough access to read and modify the history list.

Gamut's editor provides features not usually found in other graphical editors. For instance, Gamut supports two different color palettes: one for normal colors and another to create "guide objects" (see Section 4.1.2). All of Gamut's drawing areas are divided into onscreen and offscreen areas for the developer to create hidden objects and widgets to represent the application's data. Also, Gamut makes widgets such as buttons immediately active but still allows the developer to select them by clicking on their border. This eliminates many "Run/Build" mode errors that developers often commit in other tools. Gamut's editor also has other features, and these will be presented after the discussion about "nudges," Gamut's interaction for demonstrating behaviors.

4.1.1 The Drawing Area

Gamut's main window is configured as a standard graphical drawing editor (see Figure 4.1). The drawing area is bordered by palettes, menus, toolbars, and dialogs. The menubar and toolbar perform basic operations such as loading/saving, cut/copy/paste, as well as control the window layout, current mode, and popup dialog boxes. The tool palette on the left is used to create new graphics and widgets. Below the tool palette is the mouse input palette that is used to simulate player input events that use the mouse (see Section 4.3.5). Below the drawing area is the demonstration feedback area (see Section 4.2). This area has a large text window for the system to give the developer messages. The buttons alongside the text window are used by the developer to respond to the system as well as initiate demonstrating or revising behaviors.

Gamut's graphical editor is a "structured" editor as opposed to a "bitmap" editor. Sometimes such an editor is called a "drawing" tool versus a "painting" tool. In a drawing program, graphics are created via a fixed set of objects that can be manipulated using selection. In a painting program, objects the user creates from the palette are rendered onto the screen. After that, the object becomes mere pixels on the screen. A painting program usually allows fine-grained manipulation of the image as well as support for free-hand drawing. A drawing program, however, maintains the semantic structure of the rendered objects. Objects can be changed repeatedly by selecting

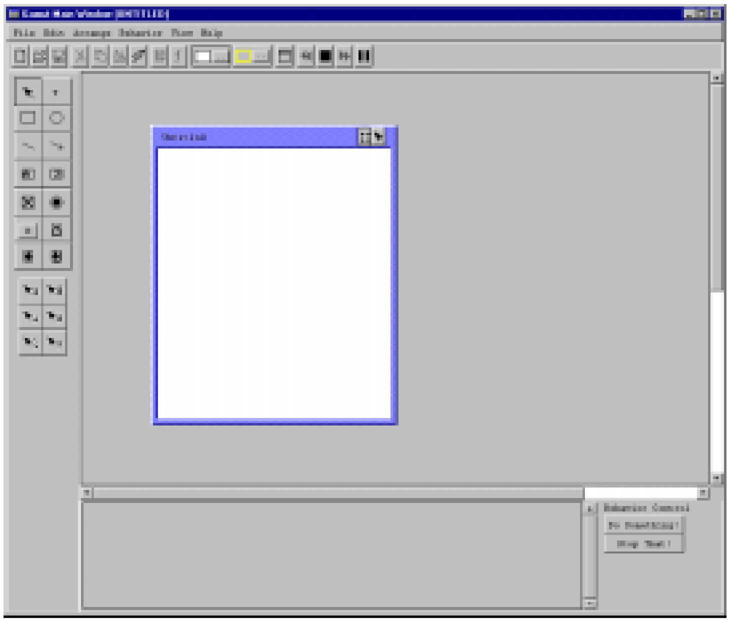

Gamut Main Window [UNTITLED]
File Edit Arrange Behavior Run Help
Behavior Control
Do Something!
Stop That!

Figure 4.1: Gamut’s main editing window. It consists of a large drawing surface with various palettes surrounding it. The area directly below the drawing area is the dialog for feedback about demonstrated behaviors.

them and using the menus and dialog boxes. The user cannot usually select an object in a painting program. Generally, drawing editors provide better support for programming by demonstration. With a drawing editor, the system does not need to interpret the pixel-level contents of the image. Furthermore, the objects are modified by discrete, repeatable commands that are easily captured in a history list.

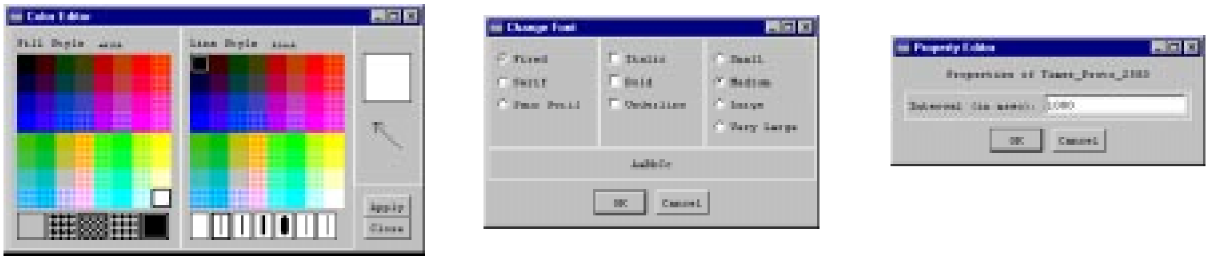
Gamut uses standard interaction techniques to create objects and edit objects on the screen. The tool palette acts as a switch; when an object’s picture is selected on the palette, dragging in the drawing area will create a new object of that type. Gamut supports several basic graphical objects like rectangles, circles, bitmapped images, lines, and arrow lines. When the arrow button on the palette is selected, the drawing window is in “select mode” where clicking on an object displays selection handles around that object. The developer can also “lasso” a group of objects by dragging the mouse in an empty region of the background. Most menu and dialog box commands operate on the selected object(s).

4.1.1.1 Property Dialogs

Gamut supports most of the same graphical properties and operations that other editors support. The developer can change the color or font by selecting the affected objects and using the appropriate menu command. Gamut’s color and font dialogs use a standard design. The developer can also modifiy numercal properties by using a generic property editor. Figure 4.2 shows examples of Gamut’s color, font, and numeric property editing dialogs.

4.1.2 Guide Objects

Guide objects are graphical objects and widgets that the developer uses to demonstrate an application’s behavior that are not visible to the player. To the developer, though, guide objects are cre-



Interval (in msec): 1000

Figure 4.2: Examples of Gamut's property editing dialogs. From left to right are the color, font, and mumeric property dialogs.

ated, moved, and modified in the conventional manner using the tool palette and selection handles. Guide objects act as "guides" that affect the behavior of other objects (including visible ones). Gamut supports two kinds of guide objects. The first is a feature borrowed from DEMO II [26] and Metamouse [54] where the developer draws objects that can be made invisible on demand. In Gamut, these objects are called "onscreen guide objects" because they are used directly in the part of the drawing area that is visible to the player. The guide objects are kept visible while the application is being created and are made invisible when the application is run by the player. The other kind of guide objects are "offscreen guide objects" that are usually used to represent the application's data. Offscreen guide objects were also used in Rehearsal World [36] where they are called "offstage actors." Offscreen guide objects are normal objects that are placed outside the player's viewing area. Offscreen guide objects do not have to be made invisible since they would not be seen anyway.

The distinction between onscreen guide objects, offscreen guide objects, and normal graphical objects is mostly a matter of the developer's perspective. Gamut treats all graphical objects in mostly the same way. The difference between onscreen and offscreen guide objects is a matter of color selection and screen location. For example if an object is always offscreen, then it can be considered an offscreen guide object, but Gamut does not know that an object will always be offscreen. For instance, the object might move onscreen at some point. Therefore, Gamut treats the object as it would any other.

4.1.2.1 Onscreen Guide Objects

The developer specifies an onscreen guide object in Gamut using color selection. In DEMO II, guide objects were created using separate icons on the tool palette, but this prevents the developer from converting a regular object into a guide object and *vice versa*. Instead, Gamut uses special colors that become invisible on demand. The color palettes shown in Figure 4.3 are popup dialog boxes that are activated from the toolbar. There are two different color dialogs, the left one is for normal colors that are always visible, and the right one is for the guide object colors. The normal color dialog is shown in Figure 4.2, and the guide object color dialog is similar.

To make an onscreen guide object, the developer draws a graphical object as usual. If the guide object color is active, then the new graphical object will automatically be a guide object. Otherwise, the developer can select the object and set its color using the popup dialog box. To help keep onscreen guide objects distinctive, Gamut limits guide object color to a small set of pastel colors. Gamut also does not set the fill style of guide colored objects so that they have a more "skeletal"

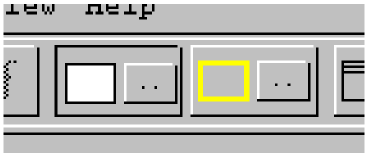


Figure 4.3: Gamut's two color selectors in the toolbar. The left selector sets objects with normal colors. The right selector sets objects to have a "guide object" color that can be made invisible.

look. To make the onscreen guide object invisible, the developer switches them off using a global mode switch. Currently, Gamut must have all guide objects visible or invisible at the same time.

4.1.2.2 Offscreen Objects

Rehearsal World [36] used a separate window for creating "offstage actors." In the Rehearsal World metaphor, graphical objects are "actors" and the "stage" is the application window. On the other hand, Gamut uses a window frame object (see Figure 4.1) to separate onscreen from offscreen areas. Everything that appears inside the window frame is visible to the player and the outside areas are not. By putting a frame in the drawing area, the drawing area remains a single window. The developer does not have to switch windows to find the offscreen objects, and the developer is not tempted to put what ought to be offscreen objects onscreen for convenience sake. The window frame also has other uses as discussed in Section 4.3.4.

The kinds of objects that are made into offscreen objects are typically widgets used to track the state of the application. Besides being placed offscreen, the objects are just like other graphical objects. Common offscreen objects include checkboxes to indicate mode changes, number boxes for counting things, or graphical images that will appear later in the application. For instance, if an object's behavior is to flash five times, the initial number five and the number of flashes the object has left to go can be represented with offscreen number boxes. In a two player game like Chess, a checkbox might be used to represent whether it is the first or second player's turn.

4.1.2.3 Guide Object Idioms

Using guide objects is likely the most challenging task a developer must master when programming with Gamut. Designing guide objects requires the developer to think beyond the surface-level activity of the application and consider how the application actually works. However, there are some common ways that guide objects are used. Four common idioms for onscreen guide objects are path following, vectors, location marking, and bumpers.

4.1.2.3.1 Path Following

A simple way to have an object move at run-time is to make it follow a path. In some applications, the path will be part of the game's interface, but in most cases, the developer will have to draw the path explicitly using lines or arrows as guide objects. Figure 4.4a shows a close-up of a board game that has been augmented with arrow lines. The arrows clarify which rectangles are part of the path, which are not, and which direction the piece is meant to follow. In the maze game in Figure 4.4b, the developer has drawn lines that show where the monster is allowed to move. If the monster is demonstrated to only follow the lines, its behavior can automatically avoid some obstacles while still moving through others.

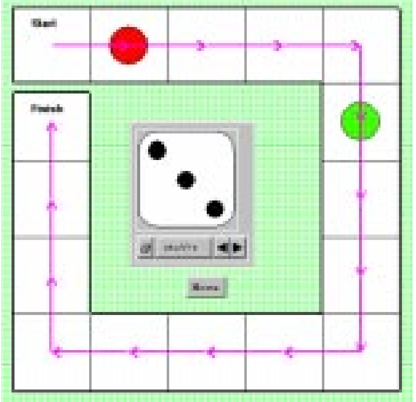


Figure 4.4a: The developer added arrow lines as guide objects to this board game to show which directions the pieces are allowed to move.

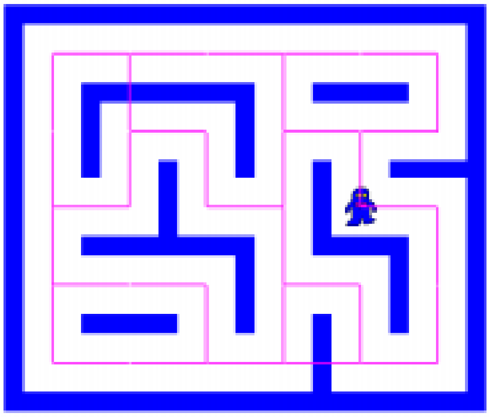


Figure 4.4b: Here the developer drew lines connecting the areas where the monster is allowed to move. Note that the monster may travel through some walls but not others.

4.1.2.3.2 Vectors

Vector following is closely related to path following. In vector following, the object still follows a path such as an arrow line but the path moves as well. The arrow line (or vector) that the object follows stays attached to the object so that the object can continue following the vector repeatedly no matter how far it has moved (see Figure 4.5). In essence, an object is given a notion of direction. This technique is reminiscent of the figures used to calculate motion in Physics classes. The Pavlov system also supports vector following by providing a special object with a vector already attached [97]. Pavlov's vector notation is fixed, however, whereas Gamut's guide objects can be used in whatever manner the developer wants. For instance, it is common practice to use an arrow line to represent the vector, but Gamut can use any graphical object as long as it can be clearly oriented.

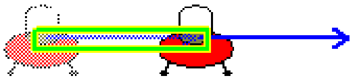


Figure 4.5: An example using vector following. Here the arrow line is trained to move with the spaceship so that the ship always has a path to follow.

4.1.2.3.1 Location Marking

Guide objects are also useful for marking hidden locations. Many game applications have special locations where events occur that are not visible in the interface. For instance, in an adventure game, the developer may put triggers into the floor panels (see Figure 4.6). When the player's character intersects the panel it might cause monsters to appear and chase the player. The developer may also use guide objects to mark locations within objects. Wires may be attached to the circuit element in Figure 4.7 only at designated points along the edge of the object. The developer can mark these positions using small circles as guide objects that are grouped to the element.

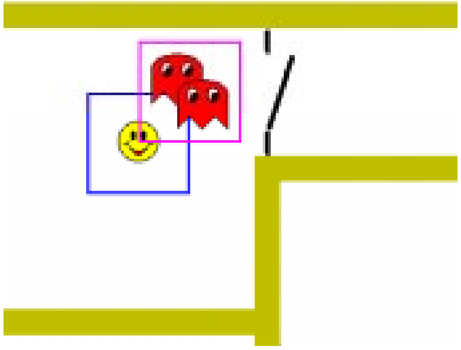


Figure 4.6: In this game, the developer has drawn a guide object rectangle as a floor panel. When the player's happy face character intersects the rectangle, the game releases two monsters. Also, the happy face is surrounded by a rectangle. Because the rectangle is bigger, it will intersect walls first. Thus, it is possible to detect when the character comes too close to a wall so that it will not walk through it.

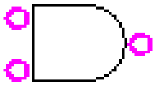


Figure 4.7: This And gate uses little circles as guide objects to show where wires may be attached.

4.1.2.3.1 Bumpers

Bumper guide objects are similar to location markers except they are used to maintain spacing between objects. A common form of bumper is to extend a region outside an object to prevent the object from intersecting a barrier. The character in Figure 4.6 uses a rectangle as a bumper to prevent it from intersecting a wall. The rectangle can be made to extend just far enough so that in a single step, the rectangle will intersect the wall before the character does. Another example uses guide objects to maintain space between objects. In Figure 4.8, two cards are stacked so that the suit of the covered card is still visible. To indicate the size of the constant offset, the developer grouped a line to the top of the covered card.

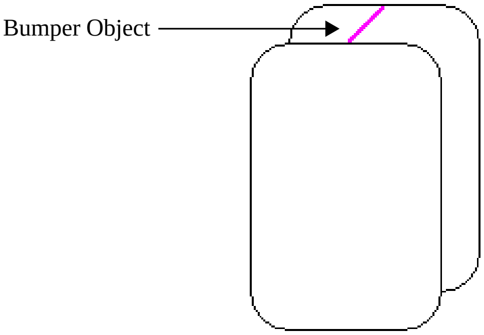


Figure 4.8: Here a line is used as a bumper to maintain the offset between two stacked cards.

4.1.3 Widgets

Widgets are graphical objects that have a system-defined behavior. The developer can use widgets as part of the application interface as well as a means to store application data. Though Gamut does not permit the system-defined behavior of a widget to be changed or eliminated, the developer may augment the widget's behavior by demonstrating responses when the widget is used.

Gamut provides five standard widgets: push buttons, radio buttons, check boxes, text input fields, and number input fields. Each widget's behavior generates events to the system. For instance, pressing a button creates a button event which the developer can use to demonstrate a response. The other widgets also store data. The check box and radio button store a toggle bit, the text field stores text, and the number field stores a number. Each widget (including the button) also has a label that can be used to store text data.

4.1.3.1 Selecting Versus Operating a Widget

Since a widget defines its own behavior, there is a conflict with wanting to both edit and use a widget at the same time. When the developer first creates a new button, there is a tendency for the developer to want to push it right away. But the developer may also want to move, resize, or change the button's label which would require the button to be selectable. This is called the "use-mention" problem [92]. A common solution is to use a Run/Edit mode switch. In Edit mode, the button is selectable and can be edited, and in Run mode, the button can be pushed to generate events. However, results from the paper-prototype study (see Section 7.1.3.3) showed that developers ignored the Run/Edit mode switch and would constantly forget which mode was currently active. Unlike typical interface builders where the amount of time spent writing code can mitigate the problems between switching modes, a developer using programming-by-demonstration is hampered by a Run/Edit mode switch because it directly affects how the developer manipulates objects in the interface. Therefore, it was important to reduce the impact of the Run/Edit mode distinction for widgets.

The first attempt to fix the mode problem used the solution found in the tool palette of a common word processor called FrameMaker [1] (see Figure 4.9a). FrameMaker defines two selection options: the first one, called "smart selection," has a little cursor icon next to the arrow and allows the user to add text to unselected objects (it performs the widget behavior). The other option, called "object selection," will only select objects so that they may be moved or resized. In order to move a text block, the user first must switch the mode to "object selection," then the text block may be selected and dragged. The same idea was originally carried over into Gamut. Figure 4.9b shows the tool palette containing the two selection icons. The icon with the selection arrow with the little arrow next to it is the "smart selection." When smart selection was active, buttons could be pressed as buttons and text widgets could have text typed into them. The other, plain, arrow was "strict selection" where widget behavior was suspended and widgets could only be selected as though they were graphical objects.

Informal testing indicated that the two arrow icons were confusing. The developer needs to switch between using a widget and selecting a widget frequently and often forgets the state of the selection icons. Also, developers were confused by the icons and thought they actually were a Run/Edit mode switch that they were not. A Run/Edit switch would make guide objects invisible, and cause mouse events to act as player input. The selection icons, however, only affect selection. In

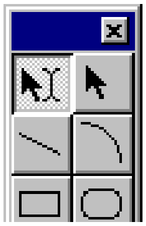


Figure 4.9a: Part of FrameMaker's tool palette. The arrow with the cursor-like icon is the "smart selection."

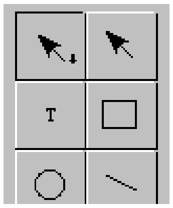


Figure 4.9b: Part of Gamut's original tool palette. The icon with two arrows is the "smart selection."

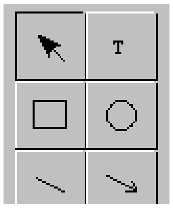


Figure 4.9c: Part of Gamut's final tool palette. The extra selection item is removed.

essence, the two-icon approach created all the problems of a Run/Edit mode switch without providing much of the benefit.

Gamut's final design for widget selection was inspired by the actions of an informal test subject who persistently tried to select a button by clicking the mouse near its edge. We incorporated that behavior in Gamut by giving each widget a selectable frame. Clicking in the center of the widget performs the widget behavior, but clicking on the edge selects it. Some widgets such as text input fields also have large, inactive regions like the label area that are also made selectable. This design allows the developer to operate a widget as soon as it is created as well as select it for further editing without the overhead of changing a mode. It also allowed us to remove the confusing selection option from the tool palette (Figure 4.9c).

The final design for widget selection has worked well. Subjects in the final usability study generally had no problems with it (see Section 7.2.4.8). Since the selection region for some widgets is small, it could be difficult to find the border quickly. Sometimes subjects would have to click multiple times near the border until they found the right spot. This might be improved if the cursor changed when it hovers over the widget's selectable border. Other subjects would "lasso" the button if it proved to difficult to click on (by selecting a region that included the button). Subjects also were able to unselect the widget if they somehow selected one by accident. None of the subjects seemed confused by the behavior or had significant trouble accomplishing the tasks.

4.2 Demonstrating Behavior

Gamut's technique for demonstrating examples is called "nudges." This metaphor emphasizes that behavior is demonstrated *incrementally* in Gamut. The developer gives Gamut several small "nudges" to demonstrate new behavior and to correct the system when it makes a mistake. Each nudge demonstrates an aspect of the desired behavior that is built up one small portion at a time in no particular order.

Gamut defines two nudges named "Do Something" and "Stop That." Both nudges are invoked by a single button press that moves the system into Response mode. This is the only significant mode that Gamut uses. The Do Something nudge tells the system that the developer wants to define a behavior for an idle object. Stop That tells the system to stop performing whatever behavior just occurred on a set of objects. Using Stop That essentially demonstrates a negative example because it tells the system what *not* to do. In Response mode, the developer gives the system hints by “highlighting” the objects that are important. The system may also ask the developer questions during Response mode to resolve inconsistencies in the new behavior. The developer responds to these questions by highlighting objects. Response mode ends when the developer pushes a button and finishes answering Gamut’s questions.

4.2.1 Demonstration By Nudging

The principle behind nudges is to allow any behavior to be augmented in any order at the time when the developer wants to work on it. The developer does not necessarily know beforehand when a behavior is wrong; therefore, it is important that the developer be allowed to revise a behavior as soon as the problem surfaces and in the context in which it occurs. As a result, examples in Gamut tend to be small, incremental revisions to a larger behavior. First, the developer demonstrates the fundamental actions the behavior performs. Then, each new example provides a nuance to the fundamental behavior until the system knows the behavior entirely.

In order to demonstrate a behavior in Gamut, the developer first performs the event that causes the behavior to occur. For instance, the developer may push a button or drop a mouse icon (see Section 4.3.5) to generate the event. If the behavior has not been defined yet, or if the behavior has a problem, then the system will misbehave in response to the event. The developer optionally selects the objects that did not do the right thing and immediately “nudges” the system. To nudge the system, the developer pushes one of two buttons labelled “Do Something” and “Stop That” (at the bottom right corner of Figure 4.1). Once a button is pressed, the system will begin putting instructions into the text region of the behavior dialog area and the nudge buttons are replaced with buttons that the developer uses to complete the example and answer questions. Instructions will continue to be displayed until the system leaves Response mode indicating that the example is complete. This process is outlined in Figure 4.10.

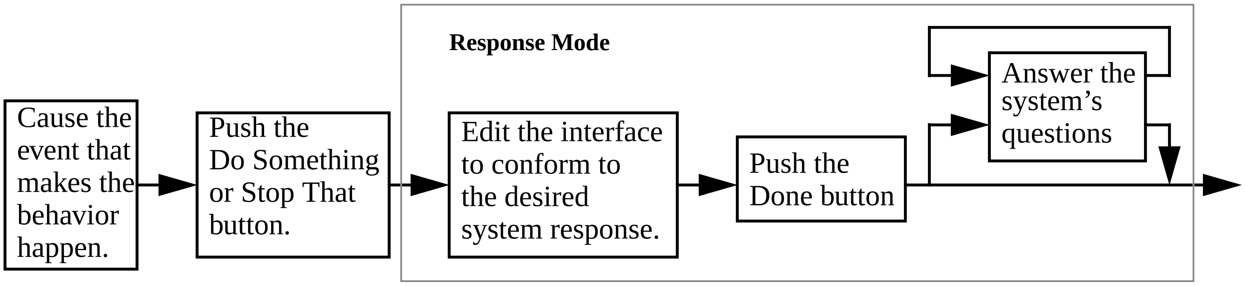
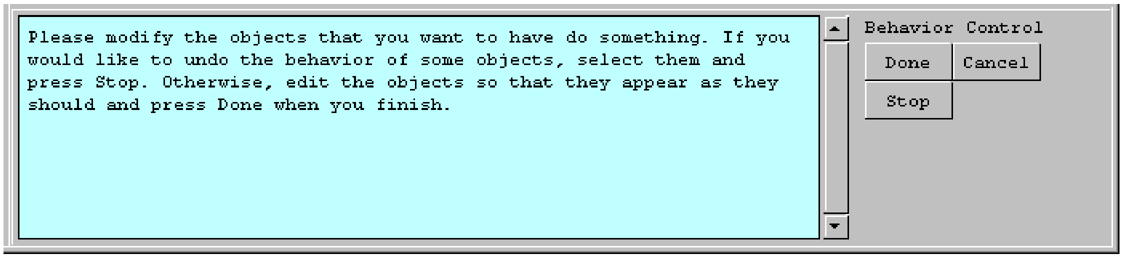

Response Mode
Cause the event that makes the behavior happen.
Push the Do Something or Stop That button.
Edit the interface to conform to the desired system response.
Push the Done button
Answer the system’s questions

Figure 4.10: The phases for demonstrating a behavior. Note that the developer’s role consists mainly of arranging the demonstration, and that the system’s query phase may not even occur.

4.2.1.1 Do Something, Stop That

The Do Something nudge is used to create new behavior and to augment existing behavior. Normally, the developer pushes Do Something because an object that was supposed to change did not perform any action. After the developer performs the event and sees that an object did not respond, he or she pushes the Do Something button. Gamut will ask the developer to demonstrate what to do by presenting the dialog shown in Figure 4.11 and shifts the system into Response mode. Do Something is used to demonstrate both “do something new” and “do something more.”

The Stop That nudge is used to stop objects from performing actions caused by the event. After the developer causes the event, he or she selects objects that responded improperly and presses the Stop That button. Stop That performs undo on the selected objects before switching to response mode. Once the objects are stopped, the developer has the choice of either leaving the objects as they are or demonstrating a new response for the objects. Commonly, Stop That is also used as a way to "do something differently." That is, the developer does not actually want the objects to stop altogether. Instead, the developer wants the objects to stop doing what they were doing so that another behavior may be demonstrated in its place.



Please modify the objects that you want to have do something. If you would like to undo the behavior of some objects, select them and press Stop. Otherwise, edit the objects so that they appear as they should and press Done when you finish.

Figure 4.11: The Do Something dialog. The developer has just pushed Do Something because the system needs to be taught new behavior. The dialog area fills with text and the buttons on the right change.

In cases where the system modifies an object but was supposed to do something else, both the Do Something and Stop That buttons are applicable. In these cases, the developer can use either button to proceed because Gamut is not actually influenced by the manner that the developer reaches Response mode. Stop That provides a convenient way to undo actions on a set of objects, but the developer could also press Do Something and undo the actions by editing the objects manually. Gamut interprets the final state of the application after the developer has finished demonstrating.

4.2.2 Response Mode

All demonstrated behaviors in Gamut must be created using Response mode. It is called Response mode because it is the time when the developer shows the application's response to the event causing the behavior to occur. In Response mode, the system records all actions converting them into a single example "trace" to be presented to the inferencing algorithm. Gamut will only record and modify behavior using examples gathered from Response mode. All other times that a behavior runs are not recorded as examples. In other words, the system does not assume that a behavior has run correctly when the developer does not explicitly correct it using nudges.

Gamut makes the transition to Response mode dramatic to remind the developer that its purpose is to demonstrate and not to perform routine editing tasks. Gamut changes the background color of the drawing area to be bright, creates "temporal ghosts" for all the objects that recently changed (see Section 4.2.2.2), and puts text into the behavior dialog prompting the developer about what to do next. The transition into Response mode promotes a degree of urgency so that the developer will not forget that the mode is active. The developer must complete the example in Response mode before moving on to other editing tasks.

Most of Gamut's palettes and menu commands are available during Response mode. The developer may perform any edit on any object. The developer's goal is to modify the look of the interface and objects to reflect the operation that the behavior performs. Gamut requires that the developer modify the interface to look exactly as it should and not just a close approximation. To aid the developer, Gamut provides gridding to help align objects precisely. In the future, it would help further to provide snap dragging and visual feedback when objects become aligned in ways that Gamut will recognize (see Section 8.1.4).

All edits performed during Response mode are assumed to occur as an instantaneous transition for the player. Animated behaviors that occur over a span of time or in multiple steps must be constructed using a timer widget (Section 4.3.3) or similar event-producing widget to tell Gamut how long the interval between steps is. As a result, Gamut is good at learning behaviors where several actions occur at once to produce a combined result. However, sequential behaviors where there is a visible time delay between each step are more difficult to generate since they involve animation or timer widgets.

4.2.2.1 Hint Highlighting

Designing a system that makes effective use of hints is a major contribution of this thesis. The developer uses hints to show relationships that would be too costly for the inferencing system to discover on its own. Objects that represent modes and conditional states are often not manipulated by the behaviors they affect. Typical applications can have hundreds of objects in a single display. It is possible that many objects besides the ones affecting a behavior have changed state since the last time that behavior was demonstrated. It is essentially impossible to discover the objects that matter in a behavior without help from the developer in all but the most trivial applications. The way that hints are used by the inferencing system is discussed in Section 6.5.2.

To give a hint, the developer *highlights* the desired objects. Highlighting is basically a form of selection but we have separated it from normal selection for two reasons. First, selection is needed for other operations such as editing objects. If the developer were to select an item as a hint and then edit other objects, it is likely that the hinted objects will be forgotten and not be selected when the developer finishes. Second, the inferencing algorithm does not want to have too many objects highlighted at once. Gamut only lets the developer highlight one object at a time. The selection handles would let the developer lasso the entire window. If the system is given all objects as a hint, it will bog down into potentially endless speculation about which objects really matter. Thus, Gamut's hinting interfaces discourages the developer from highlighting too many items at once.

A highlighted object is indicated by surrounding it with a green rectangle (see Figure 4.12). The developer highlights an object by clicking on it with the right mouse button. To unhighlight an object, the developer clicks on it again. As was previously mentioned, Gamut does not allow the developer to lasso objects when highlighting to encourage the developer to highlight only a small number of objects.

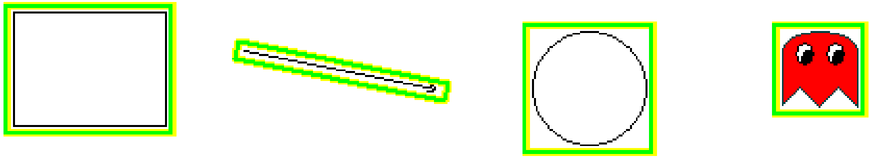


Figure 4.12: Highlighted objects are surrounded by a greenish rectangle.

Highlighting is available during Response mode and at no other time. It becomes active as soon as the developer pushes Do Something or Stop That and continues until the example is complete. Highlighting is optional while the developer edits the example before pushing the Done button. An expert developer will highlight objects during this time that are known to be important to the system. A novice may choose not to highlight anything and wait for the system to ask questions. When the system finds behaviors impossible to distinguish, it will ask the developer questions to resolve the impasse (see Section 4.2.3.2). The developer answers the system's questions by highlighting objects (or choosing one of the special buttons besides Done).

In Maulsby's thesis, another kind of hint is suggested that Gamut does not implement. In Cima, the developer was allowed to give a "negative hint" [55]. A negative hint would imply that an object that would normally incorporated into a behavior should be ignored. This feature is not useful in Gamut because the system typically does not incorporate objects that are not highlighted. Furthermore, in order for developers to provide a negative hint, they must first know that the system would likely incorporate the object and that it should not do so. Since we expect developers to be nonprogrammers such an inductive leap seems unlikely. A more plausible scenario would be to allow the developer to see which objects the system gathered after a behavior is created and strike out those that are incorrect. This is mentioned in the future work Section 8.2.2.

Another feature that Gamut could implement in the future is the ability to tailor hints. Currently, Gamut only allows whole objects to be highlighted, but the behavior may only depend on only one feature of the object like its color. The developer could be allowed to highlight only the properties that the behavior actually uses. This might be accomplished by augmenting the property editor to allow the developer to highlight an object's fields.

4.2.2.2 Temporal Ghosts

Many behaviors depend on the past state of objects that are changed by actions in the behavior. For example, a piece following a path in a board game must usually move relative to the position where it started. Gamut provides access to the prior application state of modified objects with "temporal ghosts." A temporal ghost is a dimmed, translucent image of the object as it appeared before it was changed (see Figure 4.13).

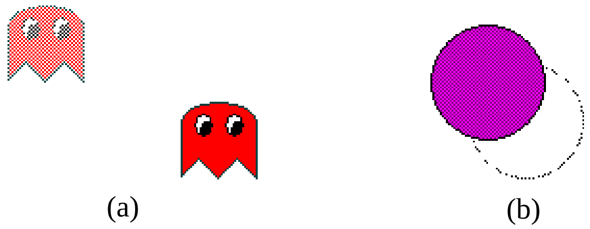


Figure 4.13: Objects with their associated temporal ghosts. The object marked (b) did not move, therefore its ghost is shown in a shifted position.

The main purpose of ghost objects is to give the developer an object to highlight when the past state of an object matters. Since the object was changed, its original position or color would be lost if the ghost was not present. Ghosts also indicate to the developer which objects were modified by the behavior. Unmodified objects do not have ghosts. Also, a temporal ghost cannot be edited since its state is fixed by the object's past. As a result, ghost objects cannot be selected.

The position where the ghost object is drawn depends on the type of the original object and how the object was modified. If the modified object is moved, then the ghost will usually appear in the object's original position. However, if the object is not moved or if another object is positioned in the same location as the ghost, then the ghost will be placed at an offset. For instance, when a circle's color is changed, the appearance of the circle's ghost with its old color is placed directly underneath the original but offset so it can be seen as in Figure 4.13(b). Gamut tries to maintain the original z-order of the objects as closely as possible. If the z-order of the object is the property that changes, then the z-order of the ghost is kept in the original position. The ghost will also be placed in the original z-order position if the original object is moved to another window (such as a card or deck widget, Section 4.3.1 and Section 4.3.2). If the original z-order is not changed, then the ghost is placed one step below the original.

An issue for future work is that it is possible for a ghost object to become completely inaccessible. If two ghosts are aligned at exactly the same point, the lower ghost cannot be highlighted. If a normal object obscures a ghost, it can potentially be moved aside, but this assumes that the developer knows the ghost is present. Gamut uses offsets for shifting ghosts even if the obscuring object is not that ghost's original object. This helps in situations where a ghost might be obscured by a moving object, but it does not solve problems where two ghosts overlap or where the obscuring object is very large. Gamut could potentially shift ghosts in different directions when they overlap or mark ghosts in a special way if they are completely covered by other objects, but this has not been implemented.

4.2.3 Dialog Feedback during Demonstration

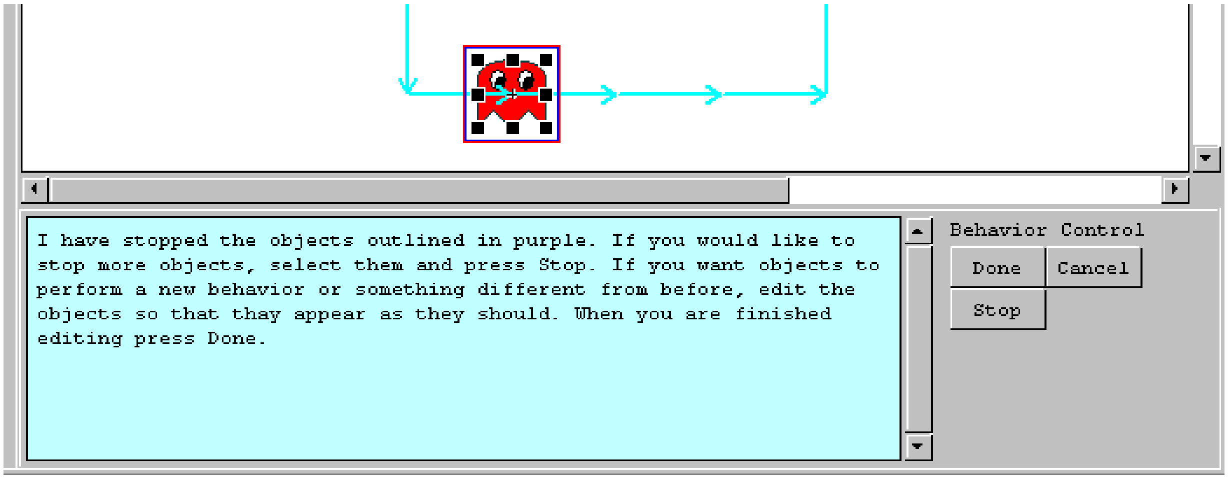
The behavior dialog region at the bottom of the Gamut application window (see Figure 4.11) is designed to help the developer successfully complete the demonstration process. The dialog gives the developer information about how to proceed and asks questions when necessary. Most of the time while the developer draws objects, the dialog remains blank and the buttons are Do Something and Stop That. But during Response mode, the text region is filled with information pertinent to the developer's task and the buttons contain choices that the developer makes to complete the demonstration process.

Gamut uses three kinds of dialogs in the behavior dialog region. The first dialog type asks the developer to demonstrate a new example. The second dialog forms questions in which the system asks the developer to supply information that the system could not determine on its own. The third kind of dialog is used to display help messages.

4.2.3.1 Demonstration Dialogs

The first kind of behavior dialog asks the developer to demonstrate a new example. The text in the dialog depends on whether the developer uses the Do Something or Stop That nudge and whether there are objects selected in the display. Figure 4.11 shows a common dialog that might appear. The developer has generated an event and pressed Do Something. A similar dialog appears when the developer selects an object and presses Stop That as shown in Figure 4.14. The system undoes any actions performed on the selected object and marks that object with a purple outline.

Both the Do Something and Stop That dialogs give the developer three choices with which to continue. The developer can signal that the example is complete by pushing the "Done" button. The



I have stopped the objects outlined in purple. If you would like to stop more objects, select them and press Stop. If you want objects to perform a new behavior or something different from before, edit the objects so that thay appear as they should. When you are finished editing press Done.

Figure 4.14: This is a Stop That dialog. In the drawing region (only half visible), the stopped object is shown marked with a purple outline.

developer can also select objects and have the system perform Stop That on them by using the "Stop" button. (If the dialog is for the Stop That command, the newly selected objects would be stopped in addition to those stopped before.) Finally, if the developer decides that entering Response mode was a mistake, then he or she can always push the "Cancel" button.

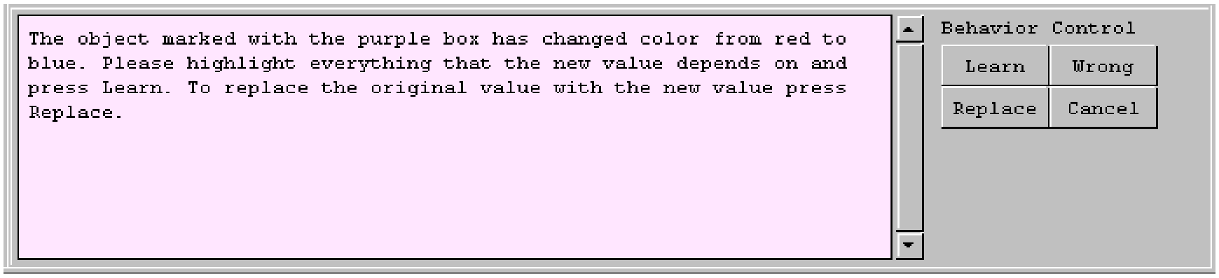
The Cancel button is available in both the example creation and question dialogs. Cancelling will immediately stop Response mode and bring the system back to normal editing mode. If the developer or the system has made edits to the objects while the system was in Response mode, those edits are undone when Cancel is pressed to bring the state of the application back to how it appeared before the nudge button was pressed. Gamut undoes all the edits during Response mode to bring the system back to a safe, consistent state. Often, developers use Cancel to escape from a situation where they lose track of what they were demonstrating. The state of the interface right before it was nudged is typically a good point from which to restart.

The developer may highlight objects at any time while the example is being created. The system will not actually interpret the highlighted objects until the example is complete, and the developer pushes "Done." Highlighting is strictly optional during this phase, and the system is not guaranteed to use all objects that the developer highlights nor will it necessarily remember what objects were highlighted in a previous example. Thus, novices might be advised to not highlight any objects and wait for Gamut to ask a question.

4.2.3.2 Question Dialogs

The question dialogs are Gamut's means for communicating behavior-related problems to the developer. The developer normally answers Gamut's dialogs by highlighting objects and pressing the "Learn" button. Unlike the example-creation dialogs that all use fixed text statements, the query dialogs are dynamically generated. The contents of the query dialog depend on actions and values in the previous behavior as well as the manner in which the developer has changed the behavior during example creation.

As will be explained in more detail in Section 6.5.5, Gamut generates questions based on the differences it finds between the new example and the previously inferred behavior. A difference is composed of two fields: what the behavior used to do, and what the new example does instead. If the original behavior changed an object's color to blue and the developer changes it so that the color now becomes red, then that difference is recorded as the change of a literal value from blue to red. The algorithm that Gamut uses to reduce a complicated behavior to a set of difference records is described in Section 6.4.4. The differences Gamut finds determine the format for the questions Gamut asks. The basic dialog sentence is structured as "The original behavior used to perform X but now it performs Y, please highlight the object that controls this part of the behavior." An example question dialog is shown in Figure 4.15.



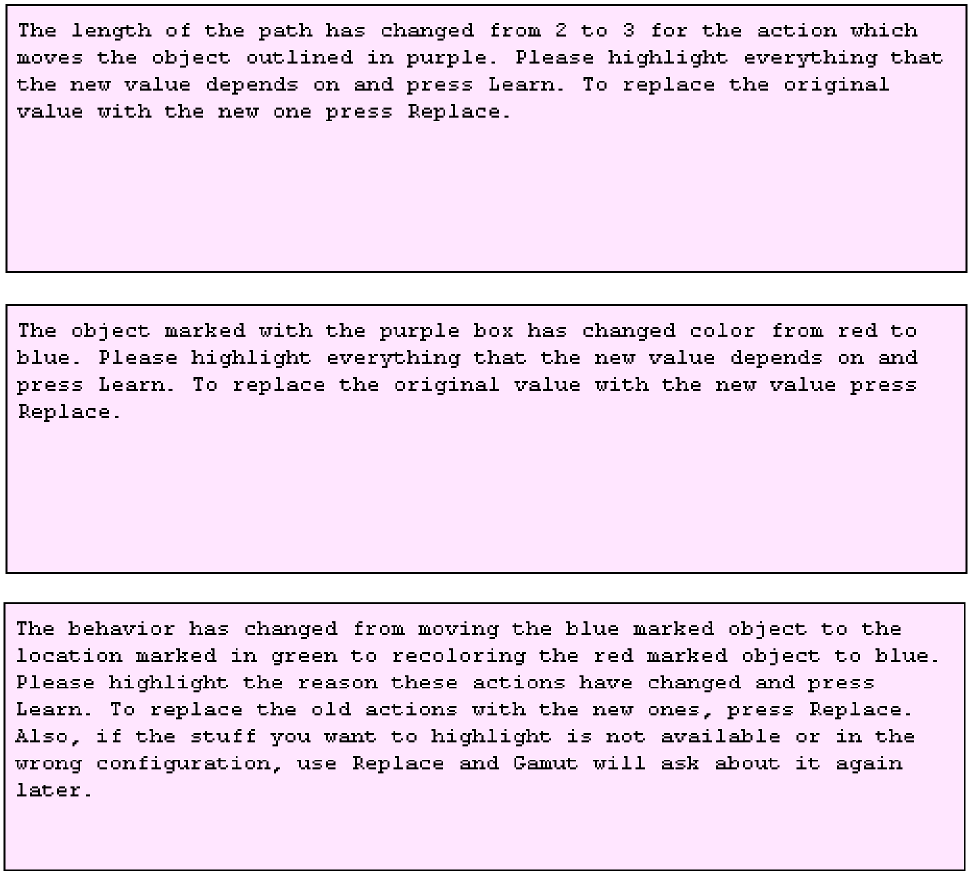
The object marked with the purple box has changed color from red to blue. Please highlight everything that the new value depends on and press Learn. To replace the original value with the new value press Replace.

Figure 4.15: A sample question dialog. The question is asking why the color of an object has changed from red to blue. More complicated actions generate more complicated questions.

Gamut's questions use three sources of information: the type of the data that has changed, the immediate context of the behavior that uses the data, and the action in which the data was applied. The type of the data is used to print its old and new values. If the data is a number or a color, then the name of the values can be printed directly in the dialog text as shown in Figure 4.15. However, if the value is an object, then the value has no printable name. Instead, Gamut draws a colored box around the noted objects and refers them by the color of the box. Gamut uses the same marking scheme to indicate graphical locations.

When a behavior contains a conditional statement that changes which actions are executed, then Gamut must write out the original and new set of actions in the question dialog. Gamut writes each set of actions as a list separated by commas. If the changed set of actions contains multiple kinds of actions then Gamut only writes a one word description for each. For a single action, Gamut will also write out any values that the action uses. For example, if the actions originally moved one set of objects and deleted another, Gamut would use the phrase, "moving and deleting the blue marked objects." If Gamut is setting the value of a toggle switch, it might say "setting the red marked object's toggle value to true."

A value's context can also be used in a question dialog. Understanding a value's context requires an understanding of Gamut's code structure that is presented in Section 6.1. A critical issue with respect to the context is that some values are embedded more deeply in the code than other. If the value is embedded deeply enough, Gamut will describe both the immediate context of value and the action where that context is used. Values embedded less deeply may only have their action described, and values at the topmost level (i.e. actions in a condition) will be described by themselves. Figure 4.16 shows three cases where a value is described with both an action and context, with an action alone, and finally as a value alone.



The length of the path has changed from 2 to 3 for the action which moves the object outlined in purple. Please highlight everything that the new value depends on and press Learn. To replace the original value with the new one press Replace.

The object marked with the purple box has changed color from red to blue. Please highlight everything that the new value depends on and press Learn. To replace the original value with the new value press Replace.

The behavior has changed from moving the blue marked object to the location marked in green to recoloring the red marked object to blue. Please highlight the reason these actions have changed and press Learn. To replace the old actions with the new ones, press Replace. Also, if the stuff you want to highlight is not available or in the wrong configuration, use Replace and Gamut will ask about it again later.

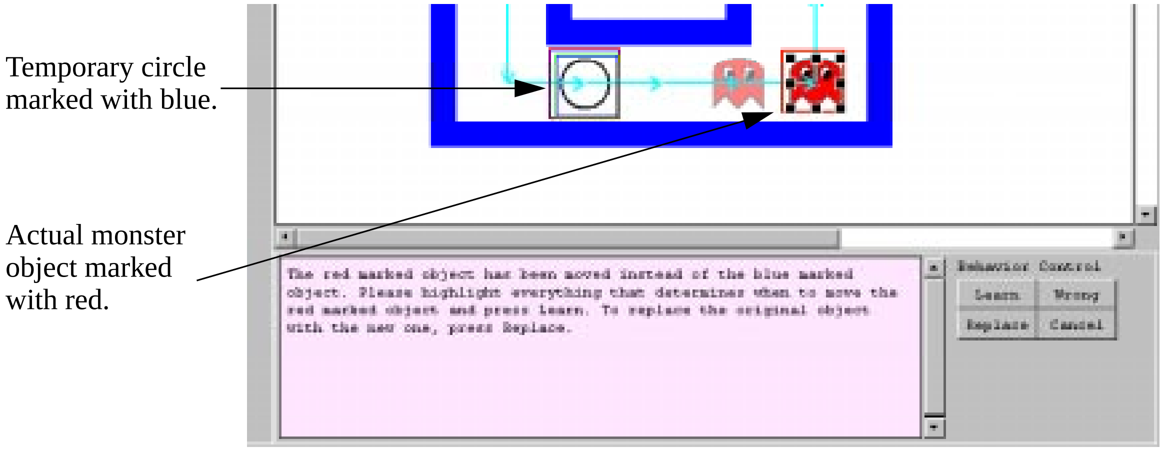
Figure 4.16: Three question dialogs. The first shows a question that refers to both the context of the value (a path description) and the action in which the context resides (a move/grow action). The second describes the values (red and blue) along with its action (change property). The last shows a dialog where the values are actions themselves (move/grow and change property).

4.2.3.2.1 Answering the Question

Gamut asks the developer to highlight objects that resolve the differences it finds between the old behavior and the new example. Question dialogs provide four buttons labelled “Learn,” “Replace,” “Wrong,” and “Cancel.” The Cancel button aborts the question as described earlier (see Section 4.2.3.1). The other buttons are used to answer the question. The most commonly selected button is “Learn” that the developer pushes after highlighting the desired objects. The Replace and Wrong buttons do not require the developer to highlight an object. The Replace button is an editing command that overwrites a portion of the old behavior with the value demonstrated in the new example, and the Wrong button is used to choose a different question when the current query does not apply to the behavior as the developer intends.

The implication of using Learn is that the original behavior is correct but needs to be refined and the system must learn about the new circumstances in the new example. The highlighted objects will show which part of the application state caused circumstances to change. The Learn button is the most useful to developers as it is the command used to combine multiple simple examples to create one complex behavior.

The Replace button lets the developer make changes that overwrite existing behavior. For instance, suppose the developer wants to train a complicated monster behavior but currently does not have the image of the monster available. In Figure 4.17, the developer trains a circle (in lieu of the monster) to follow the path of arrow lines. Later, the developer draws the image of a monster and then wants the monster to follow the path and not the circle. When the developer demonstrates an example, Gamut will move the circle as it has been previously taught, but the developer wants the monster to move so he or she stops the circle with the Stop That button and moves the monster instead. The system will want to know what causes the monster to move instead of the circle. Instead of highlighting something, the developer pushes Replace that causes the circle object to be directly replaced by the monster object. This tells the system that the circle is no longer important and the monster should always be moved.


Temporary circle marked with blue.
Actual monster object marked with red.
Behavior Control
Learn
Wrong
Replace
Cancel

The red marked object has been moved instead of the blue marked object. Please highlight everything that determines when to move the red marked object and press Learn. To replace the original object with the new one, press Replace.

Figure 4.17: The developer uses Replace to replace a temporary circle object with the actual monster object that the game is supposed to use.

The algorithm that determines the differences between the original behavior and the new example can make mistakes. Because it is based on heuristic searching methods, it can occasionally generate spurious and incorrect differences that in turn lead to spurious questions. Therefore, the developer will sometimes be asked an odd question. The likelihood of this depends on the behavior being demonstrated and how unusual the intended behavior is from the norms that the heuristics assume. Odd questions also occur if the developer makes mistakes while demonstrating a behavior and does not notice them. The developer may choose to ignore a question by pressing the “Wrong” button. By declaring that a query is “wrong,” the developer is stating that the question makes no sense. Of course, the Wrong button can easily be abused. The developer could declare that all questions are wrong and never make any progress. To prevent this scenario, Gamut maintains a hierarchy of questions where more general questions occur if the developer dismisses the more specific questions. It is possible for the developer to answer the more general question in order to resolve the system’s conflict. If the most general question is also dismissed with the Wrong button, then the developer has essentially contradicted what was demonstrated in the example. At this point, Gamut treats the Wrong button like Replace so that it does not have to display an error message. Another option might be to eliminate the Wrong button when the system reaches its most general question so that developers are unable to contradict themselves this way.

4.2.3.2.1 Special Replace Technique

There is another situation where the developer must use Replace besides making the system replace one behavior for another. It occurs when the system is determining the attributes for a conditional statement (see Section 6.6.3). Gamut can only recognize “true” constraints. For instance, if the developer highlights two objects, the system can tell whether the two objects overlap. If the objects do overlap, then Gamut can create a predicate that tests whether the two objects overlap in the future. However, if the two objects do not overlap, then the system cannot assume that they are ever supposed to overlap and the system will not create a predicate to test a nonexistent constraint. If the developer wants to demonstrate a condition based on constraints that do not exist at the time when Gamut is posing a question, then the developer ought to use Replace to tell Gamut to move on and ask the question later.

Suppose the developer wants to demonstrate that when a piece in a board game lands on another, the landed on piece moves away. The pieces are meant to move when the player pushes the “Move” button. In prior examples, the developer trains the pieces to move and alternate turns. In the first example to demonstrate the landing behavior, the developer lands one piece on another and moves the landed on piece as desired (see Figure 4.18). Gamut does not know at the time that the behavior is even conditional, so it accepts the example without question. To Gamut, the movement of the second piece seems like an additional part of the behavior which it can just add to the original.

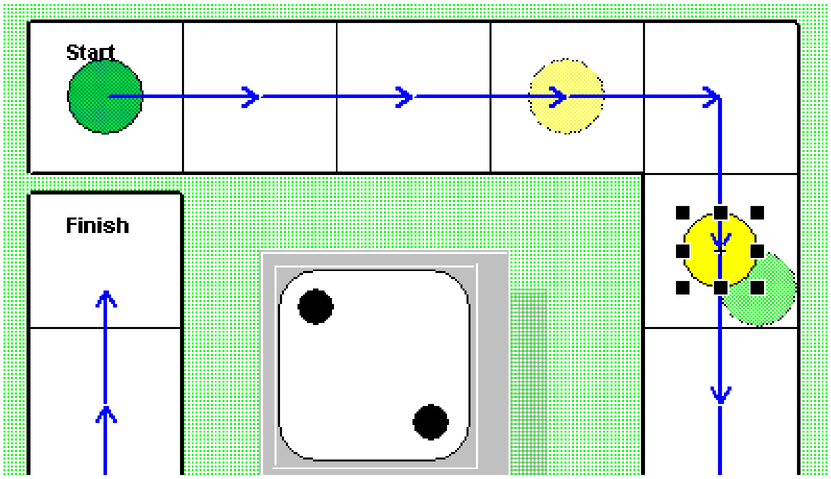


Figure 4.18: In the first example, the developer lands the light-colored circle on the darker one to make the landed on circle move to the start.

The next time the developer pushes the Move button, the system has no condition so it moves the previously landed on piece even though no object landed on it (see Figure 4.19). The developer uses Stop That to correct the behavior at which point Gamut knows that the behavior is conditional, so it asks the developer to highlight the objects that control the condition. However, the condition depends on having pieces that overlap and there are no overlapping pieces at that time.

If the developer just picks two pieces to highlight, Gamut will not create a predicate that tests overlapping because no objects are overlapping. Therefore, the developer uses Replace.

When the developer demonstrates the landing behavior again, Gamut will already know that a condition exists. Therefore it can ask what the condition depends on which in this case would occur when objects do overlap (see Figure 4.20). Though one might expect that only an expert developer could understand this technique, it turned out that two participants were able to use it successfully in experimental tasks (see Section 7.2.4.1). Other developers might need more experience to understand why Replace is needed in these cases, but it is still a useful technique. The technique prevents the developer from flailing when objects are not in the right configuration when Gamut asks a question at the wrong time.

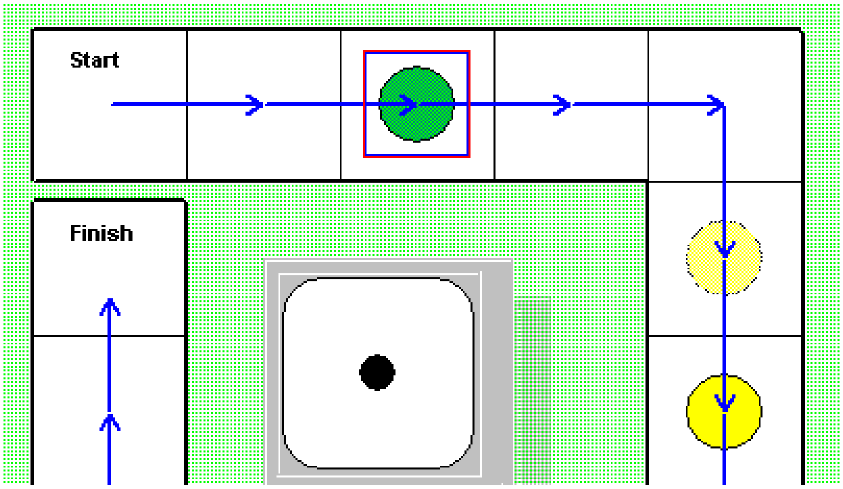


Figure 4.19: In the second example, the developer stops the system from always moving the circle that was previously landed on. The system asks why but since no circles are overlapping, the developer uses Replace.

In principle, it is possible for Gamut to learn this behavior without the third example. For instance in the first example, the developer might highlight the overlapping pieces even though Gamut would have no place to use the highlighted information. Gamut would store the information in case it became useful later. In the second example, Gamut can use the stored information from the first example to create the condition and thus eliminate the need for the third example. Of course, to use this technique requires the developer to know what a conditional branch is and have the foresight to know when Gamut is going to create one in order to highlight the right objects ahead of time. It did not seem likely that the developer would ever do this, so the idea was not explored.

4.2.3.3 Help Dialogs

The third kind of behavior dialog is used to give the developer helpful information when the developer is not making progress. A novice developer may not understand that hidden state is needed for a particular behavior or may think that an object is too obvious to highlight while demonstrating. In these cases, the help dialogs attempt to coax the needed information from the developer.

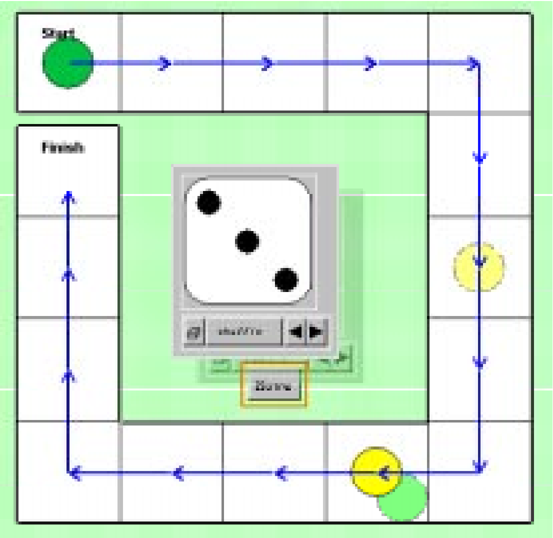
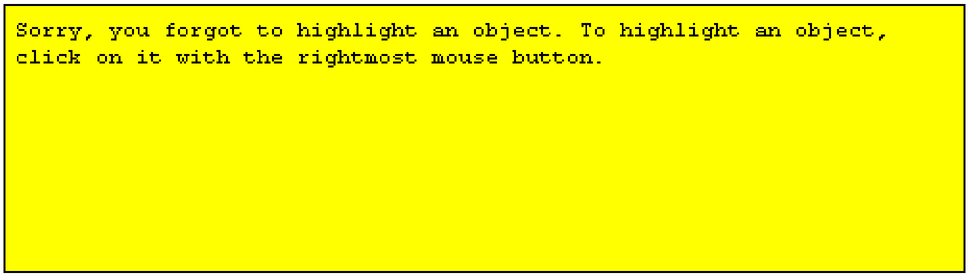


Figure 4.20: In the last example, the developer shows the landing on behavior again. This time the system asks the question when the circles overlap so the developer can highlight them. (The jumped on ball had been purposely moved so that the other could land on it.)

Since Gamut cannot infer state, it cannot tell the developer what to do in the help message. Gamut can only provide suggestions about what might be wrong and show solutions for common problems that may or may not be related to the behavior the developer is showing. Unlike the question dialogs, all help dialogs are static text messages. Help messages appear after the developer responds to a question and the inferencing algorithm has analyzed the response. Messages appear in two situations: either the developer has not highlighted any objects or the objects that the developer highlights provide no extra information to the current situation.

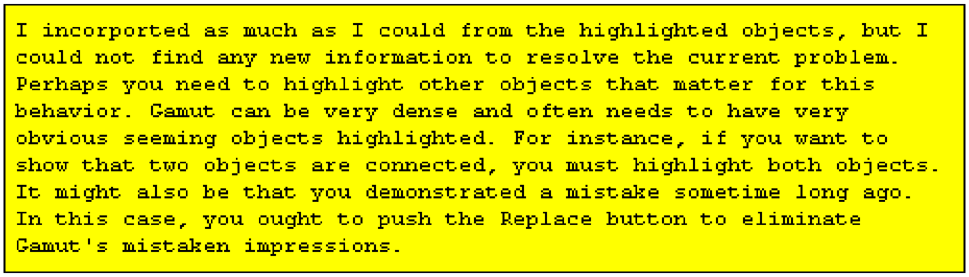
When the developer presses the Learn button but does not highlight any objects, the message in Figure 4.21 appears. The developer may made this mistake for a number of reasons. Novice developers may not actually know what highlighting is. The developer may also think that marked objects are equivalent to highlighted objects or that an object is too obvious to highlight.



Sorry, you forgot to highlight an object. To highlight an object, click on it with the rightmost mouse button.

Figure 4.21: Dialog that appears when the developer presses Learn but does not highlight any objects.

A more troublesome situation occurs when highlighted objects are not sufficient to explain a given behavior. In these cases, developers think they are being helpful because they are highlighting objects as requested, yet the system still does not seem to understand. This can happen for a number of reasons. The most common is that the developer thinks that the system already understands the connection to an object and therefore does not have to highlight it. For instance, the board game application shown in Figure 4.20 was used as a task in the final usability experiment (see Section 7.2.2.3). In the game, the piece follows the path around the board, jumping the number of spaces shown on the die. To demonstrate the behavior, the developer moves the piece the correct number of spaces. Some subjects did not think to highlight the path apparently because it seemed too obvious. With each new example, Gamut asked the developer what the position of the piece depended on, and the developer responded by highlighting the die, the piece, the button, and just about any other object besides the path. Since the path was never highlighted, Gamut could only infer that the piece moved to a different location for each different number on the die. Eventually, the die's number repeated and Gamut asked again about the location of the piece. The developer highlighted the die and other objects as usual, but Gamut could tell that there was something more because all the highlighted objects were in the same state they were in previous examples but the piece's position was different. This is the situation when the dialog shown in Figure 4.22 appears.



I incorported as much as I could from the highlighted objects, but I could not find any new information to resolve the current problem. Perhaps you need to highlight other objects that matter for this behavior. Gamut can be very dense and often needs to have very obvious seeming objects highlighted. For instance, if you want to show that two objects are connected, you must highlight both objects. It might also be that you demonstrated a mistake sometime long ago. In this case, you ought to push the Replace button to eliminate Gamut's mistaken impressions.

Figure 4.22: When the developer never highlights anything the system can use to explain a behavior, the system eventually says that it needs to have different objects highlighted.

4.3 Advanced Widgets and Player Input

Gamut supports interaction techniques besides nudges and its various dialogs. For example, applications require objects to represent their state. Game applications often need to represent data that is more complex than simple numbers or text. For instance, it is common for players to roll dice to determine outcomes in board games. Gamut must be able to store sets of common objects in a single structure and have a source of randomness.

Gamut uses a "deck of cards" metaphor to answer all these needs. Objects can be grouped in cards and decks to create complex data structures. Decks can be shuffled to provide a source for randomness and can also be used as an ordered set. Another widget supports timing that is useful in video games as a source for animation and timed events. For example, the developer might want to make a monster object that moves on its own. Finally, some games require the player to use the mouse directly on the screen, so Gamut needs a way to demonstrate mouse events. Gamut uses "player input icons" that the developer drops into the window frame to create mouse events. This section describes each of Gamut's special widgets and input techniques.

4.3.1 The Card Widget

Gamut introduces the "card" widget as means for storing data and creating abstractions. The card widget is based on a playing card metaphor like the cards used for Old Maid or the Chance deck in Monopoly. A card begins as a blank, rectangular area, but by using the card editor (see Figure 4.23) a card becomes a large drawing surface for the developer to store application data. Cards are used to group related data and act as extra drawing space when the main window is not enough.

Gamut's card metaphor is different from the "card" metaphor in Hypercard [4]. In Hypercard, cards act as screens. By creating buttons that transfer the screen from one card to the next, the developer creates applications consisting of a network of connected screen shots. Hypercard's card abstraction stacks cards into functional layers. Each layer of cards provides a different function for the application. In Gamut, cards are a data abstraction and are no more related to the screen than any other graphical object. In fact, a deck of cards might serve an application by invisibly performing actions offscreen (decks are introduced in Section 4.3.2). Gamut's cards are more like windows on another world. They have a large drawing surface of which only a portion is visible.

4.3.1.1 The Card Editor

Semantically, a card is a large surface, like the main drawing window, onto which the developer draws application objects. The card widget itself is a rectangular object that acts like any graphical object (see Figure 4.24). Only part of the card's surface is visible at any one time. Cards have an extensive "offscreen" area which the developer can use to draw offscreen guide objects. To draw items on the card, the developer invokes the card editor (by double clicking on the card or using a menu item). The card editor looks very much like the main window except that it is smaller, has fewer items on its palette, menu, and toolbar, and the window frame is replaced by a raised area with a roundtangle (see Figure 4.23). Below the drawing area is the deck viewing area. The card editor doubles as a deck viewer which is described in Section 4.3.2.1. The card editor is a separate window so that multiple cards can be viewed at once and so the developer can drag and drop objects from one drawing area to another (e.g. from the main window to a card).

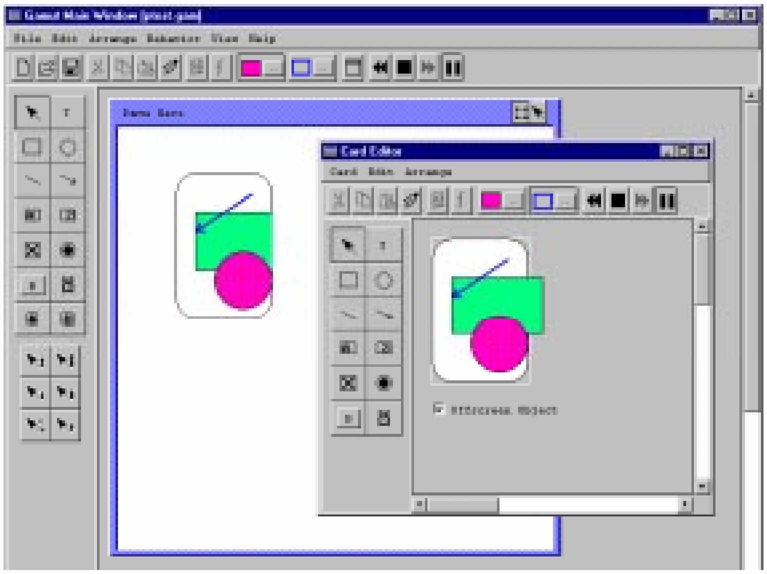

Gamut Main Window [pixel-gam]
File Edit Arrange Behavior View Help
Card Editor
Card Edit Arrange
Offscreen object

Figure 4.23: A card and its associated card editor window. The majority of the screen is a drawing surface like the main window. The area that looks raised is the region visible from the card.

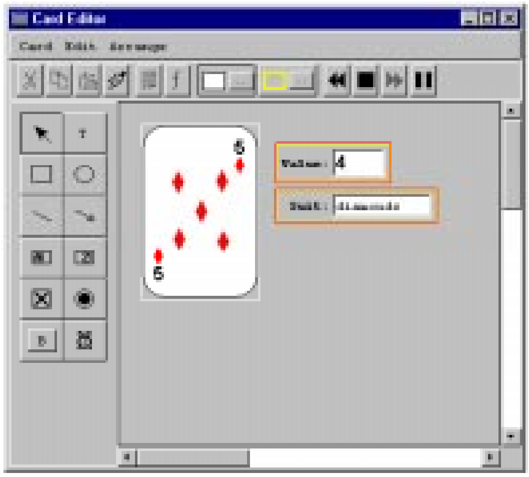
The card editor works essentially the same way as the main drawing window. Clicking on an item in the tool palette allows the developer to create new objects. The editor supports all the basic objects found in the main window. Guide objects are colored in the same way and all the basic editing operations are still found on the card editor. The commands associated with demonstration and behavior are not available on the card editor and the card editor does not have a behavior dialog area. Usually demonstrating behavior on cards is not necessary, but if it were, it is always possible to use the demonstration controls on the main window to demonstrate behavior on a card.

Because the developer might find it confusing to put a card or a deck within a card, the card and deck widget's icons are omitted from the card editor's tool palette. The concern stems from the notion that it is not possible to put a physical playing card inside another physical playing card. Note that this is not quite true, as it is possible in some games to have decks of decks. That is, the player uses one deck to select a card from a choice of several other decks. However, breaking the card metaphor might confuse some developers. Semantically, Gamut has no trouble with the developer putting a card or a deck within a card. The object would simply be present just as a rectangle or a button would be. Gamut allows cards and decks to be placed within a card, but the developer must take an extra step. The developer can create a new card or deck in the main drawing window and then transfer it into the card editor using drag and drop or cut and paste. The developer can also demonstrate cards or decks being created inside a card using this technique.

A card has an onscreen and offscreen area similar to the main window. The raised rectangle in the card's drawing area indicates the frame for the card's viewing area. The size of the frame controls the size of the widget but the position of the frame is independent of the widget's position. Anything drawn within the frame appears in the card's widget (see Figure 4.23). A new card is created with an opaque roundtangle that acts as a background (so that the card would not be invisible). The color of the roundtangle may be changed by selecting the frame and using the color dialog boxes. It is possible to make the frame invisible by converting it into a guide object. The card's background besides the roundtangle is transparent so cards can be layered on top of other objects to produce graphical effects (as in Hypercard [4]).

The area outside the frame of a card is an offscreen area just like the offscreen area of the main window. The developer stores the card's data in this area. For example, suppose the developer is creating a deck of the typical 52 playing cards. Figure 4.24 shows a picture of the five of diamonds. The raised area contains the typical arrangement of numerals and images found on a playing card. Next to the raised frame are the data of the card. Often it is useful to store the data in a format that is more easily recognized by Gamut's inferencing. For instance, the suit of the card is represented by a text input field with the label "Suit" and the value "diamonds." This property can be easily discovered by the system by looking for text fields with the label, "Suit," as opposed to searching the visible area for the corresponding graphical representation. Note that the value of the card is 4 and not 5. Recall that the Ace is actually the highest value card and the lowest valued card is the two. This means that if the cards are numbered is ascending order from 1 to 13, the number cards actually have a value one less than their printed number. Gamut's inferencing would have great difficulty discovering this property on its own so it is beneficial when the developer provides this information in the offscreen area.

Gamut provides a feature that makes certain objects in a card stand out from the others for inferencing. Gamut allows the developer to "pre-highlight" objects in the card to show that they are



Value: 4

Suit: diamonds

Figure 4.24: Objects drawn inside the card editor's raised area appears in the visible portion of the card widget. Data for the card may be placed in the offscreen area.

important. Section 4.2.2.1 discussed how Gamut allows the developer to highlight important objects during demonstration. Highlighting objects inside a card during a demonstration can be onerous because the developer has to first open up the card editor before he or she can get to the actual desired objects. Pre-highlighting provides a shortcut because Gamut will automatically highlight the pre-highlighted objects whenever the card that holds the objects is highlighted. The boxes drawn around some objects in Figure 4.24 indicate that those objects are pre-highlighted. Pre-highlighted objects only need to be marked once so the developer is less likely to forget. Of course, the developer may also manually highlight objects in a card by opening up the card editor and selecting objects individually. This may be desired if a card contains much data and the developer only wants to mark a small portion of it in a particular example.

4.3.1.2 Advanced Card Use

Though a novice developer may not consider these uses of the card widget, the card abstraction can perform many interesting functions in Gamut. A card is a means for hiding data in much the same way that a module or a procedure hides data in a programming language. Cards can also act like objects in an object-oriented language.

For instance, a button can be placed on a card and the behavior attached to the button can change the card's internal state. Consider a card that has been trained to add together all number boxes that overlap the card widget's area and put the result into the number box within the card. By placing the card over different sets of numbers, the card produces different results. This could be used to scan regions of numbers on a map for a set that meets a minimum threshold.

Combining cards with timers can also be used to create powerful effects. Timers are discussed in Section 4.3.3. A timer can be activated and deactivated by setting a property that allows it to be controlled by arbitrary events. Thus, by embedding a timer onto a card, it is possible to turn the card into an event-producing entity as well as a container for data. Each timer on the card is given a behavior in the usual way through demonstration. Then, by activating the card's timer, that behavior will be made to happen. Thus, it is possible to activate an arbitrary behavior by simply pointing to the right card. For instance, suppose in a game there are a set of random events such that the developer has to demonstrate each even separately. Maybe in one event, the colors of all the monsters change, whereas in another event, the player is given a bonus 100 points. The developer trains a separate timer to perform each event (as well as turn the timer off when it is done). Then, the developer puts the timers into a deck of cards with one timer stored per card. The developer pre-highlights the timers to save having to do so later. To demonstrate the random event occurring, the developer shuffles the deck, opens up the top card of the deck, and pushes the "play" button on the timer in that card. The developer does not have to show how to stop the timer because each timer has been trained to stop itself as part of its behavior. The developer would also highlight the deck in order to show Gamut that the timer is in the top card. Later, when the game is played, Gamut will activate the timer in whatever card is on top resulting in vastly different behaviors being performed at different times. In programming language terms, a card with a timer stored in this way would be like a procedure pointer or a virtual method. No other purely demonstrational language can currently claim this ability.

4.3.2 The Deck Widget

Gamut's "deck" widget (see Figure 4.25) is the tool for collecting multiple items into a single list. Based on decks of playing cards, the deck is used to represent an ordered list of related items. In addition to the card widget just described, the deck can hold any object. The deck and card widgets are not really related in terms of Gamut's implementation though both use a similar editor for display. The developer is allowed to make decks of anything that the application requires.

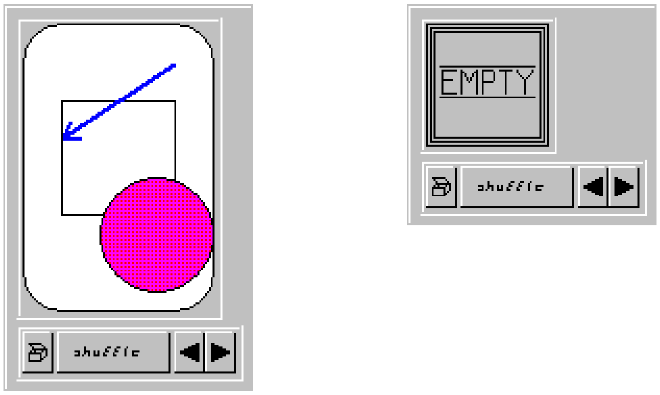


Figure 4.25: These are two deck widgets as viewed by the developer. The one on the left is filled with several card objects. The one on the right is empty.

4.3.2.1 Appearance

A deck does not have a visible appearance at run time. To a player, a deck looks like its topmost object. To the developer, a deck is a rectangular frame with a set of buttons below it. The buttons and the frame are guide objects so they will disappear when guide objects are made invisible (see Section 4.1.2.1). The deck, itself, only has a visible appearance when it is empty in which case it displays a bitmap that says "EMPTY" (see Figure 4.25).

The entire deck can be viewed using the "deck editor" portion of card editor as shown in Figure 4.26. By double-clicking on a deck to edit its properties, Gamut will bring up a deck editor window that is just a card editor window with an added section. Along the bottom of the card editor is the “deck viewer” window where each object in the deck is displayed. The developer can select a deck item by clicking on it. If the object is a card, then the card’s contents are shown in the drawing area of the card editor. The drawing area is left blank if the object is not a card.

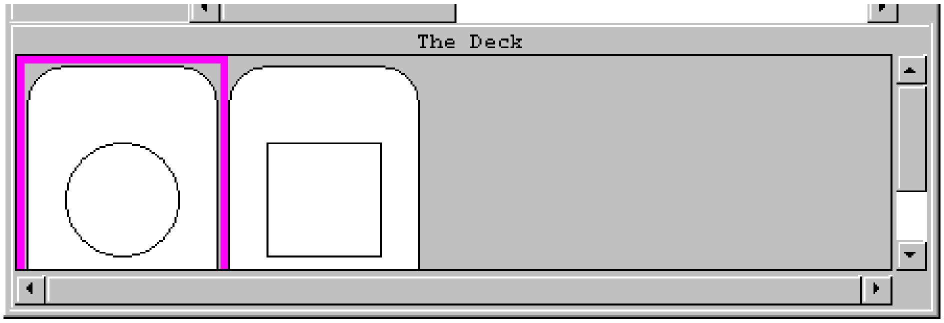


Figure 4.26: The bottom portion of the card editor (renamed the “deck editor”) shows the elements inside a deck. Otherwise, the editor behaves as before.

4.3.2.2 Using a Deck

The developer puts objects into a deck by dragging them over the top of the deck’s area. While an object is being dragged over the deck, the deck will display a thick purple outline around itself to show that it will accept the object. When the object is dropped, the deck will reposition the object so that its top left corner is aligned to the deck’s rectangle and the object will become the deck’s top card. Only the top card of the deck is ever visible at a given time.

The top object of the deck can be selected separately from the deck itself. Objects are removed from a deck by selecting and dragging them away. When the mouse pointer is no longer over the deck, the purple outline will disappear and the object will fall onto the drawing area when dropped. Similarly, objects can be transferred from one deck to another by dragging. Gamut does not currently support a menu-based method for putting objects into decks.

4.3.2.3 The Deck’s Button Panel

The deck’s button panel has four buttons as shown in Figure 4.25. The two rightmost buttons are used to set which object is displayed on the top of the deck. These buttons do not affect the order of objects in the deck. The actual order of the deck remains constant though different objects are displayed on top. In other words, the “top object” should actually be interpreted as “the object being displayed” and has nothing to do with being first in the deck. The buttons wrap around at the ends of the deck so that the deck objects continuously cycle.

To the left of the object display buttons is the “shuffle” button. Shuffling Gamut’s deck is similar to shuffling a deck of playing cards in that the order of all the items is randomized. Shuffling a deck is a common occurrence in many game applications so the operation is made easily available with the deck’s button panel. Shuffling a deck does not change the index of the object being displayed. Thus, when the objects are shuffled, a new object is likely to be placed at that index and become the newly displayed object.

Gamut does not provide a way to select a single object at random without also randomizing everything else in the deck. Hypothetically, picking a single item could be performed by randomly choosing a new display index. Though this technique would be more efficient than randomizing all the objects, developers who are not programmers are not likely to know the difference. In real card games, the whole deck is normally shuffled even when only a single card is selected; therefore, Gamut does not provide a command to randomize the display index.

The leftmost button on the panel is used to open and close the deck's "lid." Sometimes the developer will want to use a deck as though it were a graphical object. For instance, the developer may want to move another object over the top of the deck or to group the deck with another object. Normally, dragging an object over a deck would cause the deck to incorporate the object as one of its members. When the deck's lid is closed, the deck does not respond when objects are dragged over top. Also, the top object is no longer selectable and cannot be dragged out. The metaphor is that when the lid is closed, the deck is in its box and new objects cannot come in or out. By opening the lid, the deck can be made to accept objects again. It is still possible for a closed deck to gain and lose objects through a demonstrated behavior. If a previously demonstrated behavior changes the objects in a deck, its actions will not be affected by the deck's lid. The lid only prevents the developer from adding or removing objects at edit time.

4.3.2.4 Strategies for Using a Deck

In programming terms, a deck is similar to an array. It represents an ordered sequence of objects. The displayed object index can be set numerically like an array, and if the index is kept constant, adding and removing objects from a deck will act like a stack. A deck has two major uses: it can hold a list of items in order, and it can be a source for randomization.

As a simple example, consider an application where the background color of the board cycles through a fixed set of three colors: white, yellow, and red. The developer stores the color sequence in a deck using three rectangles each colored with one of the three hues. To cycle the board's color, the developer needs to demonstrate moving the current card index forward one step and setting the board to the correct color. The developer would highlight the deck to indicate that it controls the board's color. With this design, it is easy for the developer to change the sequence of colors and even to add and remove colors as the game progresses.

Gamut requires that the developer use a deck for any random behavior. For instance, the game of Monopoly has three sources of randomness: a pair of dice for moving; and two decks of cards, the Chance deck and Community Chest that are used for random events. Once appropriate decks are created, other behaviors are trained to use the decks to generate the desired outcome. Gamut only provides an even probability distribution for randomizing the objects but events can be weighted by putting multiple copies of the same item into the deck.

Decks can also be used for graphical effects. For instance, each board in a game with multiple boards can be stored in a single deck in the background. The entire background can be changed by changing the deck's display index. Short animations can similarly be created by putting the images into a deck and flipping through the images in order.

The displayed object in a deck is treated graphically as though it were simply grouped to the top of the deck. Therefore, other objects around a deck may use deck objects as location markers. One clever technique groups a deck of arrows with an object to cause the object to move randomly (see Figure 4.27). The object in this case is trained to follow arrow lines. Because objects placed in a deck are repositioned to the top-left corner of the deck, the arrows have to be grouped with a rectangle to keep their starting points in the center. The developer stores several arrows pointing in different directions in the deck as shown in the figure. The arrows and their surrounding squares are colored as guide objects. The object that is trained to follow arrow line will still follow lines even if they are embedded in a group. The developer also demonstrates that the deck shuffles when the object moves. To keep the arrows attached to the object, the developer groups the deck and the object together.

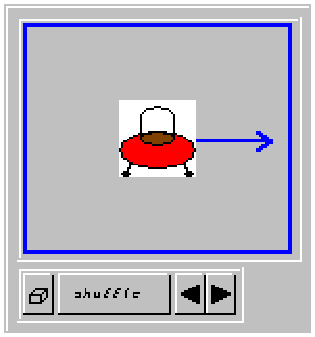


Figure 4.27: A clever trick that makes an object move randomly on the screen. The basic elements are an object trained to follow arrow lines and a deck of arrow lines pointing in different directions. The deck is grouped to the object.

4.3.2.5 An Early Deck Design

In an early design for decks, there was a stronger relationship between decks and cards. Currently, the only relationship is that the deck viewer is part of the card editor (see Section 4.3.2.1). Originally, there was no deck object, just cards. Cards would have the feature that if one were stacked on top of the other, they would automatically join to form a deck. Though this design is more in keeping with the playing card metaphor, it suffers from several drawbacks. The main problem is that other objects become much more difficult to form into decks. First, an object must be placed into a card, then the cards can be joined to another card to make a deck. Another problem is the lack of consistency about where a deck is placed and whether one exists or not. An empty deck can be a good placeholder that shows where objects go. In the card design, an empty deck would never exist; the developer would have to mark the a potential deck's position with other guide objects.

4.3.3 Timer Widget

Gamut provides a timer widget (Figure 4.28) in order to generate a continuous stream of events automatically. This kind of widget is available in other toolkits as well such as Visual Basic [60]. The interval between ticks can be set using the numeric property dialog. The buttons along the bottom of the widget control whether the timer is running or is stopped as well as a "step" button. Pressing the step button generates a single tick event without activating the timer. A developer uses step to demonstrate what the timer does without having to turn it on.

Besides using the timer to create animated effects, it also provides the essential ingredient to any behavior that repeats. This includes many forms of loops. For instance, the developer might use a

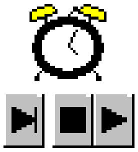


Figure 4.28: The timer widget. The buttons in order from left to right are used to generate a single step and to stop and start the timer.

timer with a number box to count out the number of times a player can move. Note that many tasks that use loops in programming languages actually would not be a loop in Gamut. When the developer modifies a set of objects in the same way, Gamut will treat the modification as a single action and will try to describe the objects as a single set.

Timers are also important for behaviors that occur over a span of time instead of all at once. By chaining timers together so that one activates the next, complex sequential behaviors can be composed. Using timers to build sequential behaviors is probably not optimal. It would likely be better to provide special widgets or inferencing to handle such tasks directly. For instance, it might be better to use a score sheet-like abstraction. Still, it is interesting that such behaviors are possible even without special purpose widgets.

4.3.4 Window Frame

In an application builder like Gamut, there is a conflict between displaying the windows of the tool itself and displaying the window of the application being created. There are essentially two choices for arranging the windows of an application builder: either one whole window is dedicated to be the application's interface, or the application's window is made a subpart of the main drawing area. Visual Basic [60] (shown in Figure 4.29a) and Rehearsal World [36] use a dedicated window for the application. Gamut (in Figure 4.29b) uses the subwindow approach. With a dedicated window, the developer is generally limited to only drawing objects that are visible to the user on the screen. If the visible surface area of the window is larger than the window itself, then developer has no way to show what is outside the window's bounds. With the subwindow approach, the interface's window is only a widget within the drawing area. By making the interface window a widget, it brings the frame into the domain of the system. It also becomes possible to manipulate the window as though it were a graphical object which is useful for scrolling effects as described next.

Other objects in Gamut's drawing area coexist alongside and in front of the frame but are not contained inside it. Moving the frame does not move objects behind the frame making it useful for scrolling effects where the background of the environment is larger that the application window. The frame can also be used in demonstrations and it can be grouped to other objects. In Figure 4.30, the window frame has been grouped to a character that moves through a maze. For the developer who sees the main drawing window, the frame appears to move about the screen along with the character. But to the player, who only sees what is inside the frame, the application window and the character are stationary and the background maze moves behind the character.

In graphics toolkits, translational effects like the kind provided by the frame widget are performed by a "viewport" [29]. A viewport represents a translation and scaling relationship. It is used to create a view of that image from a different perspective. Since Amulet, the toolkit in which Gamut

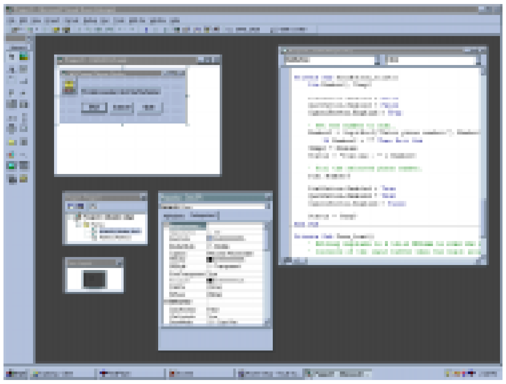


Figure 4.29a: Window configuration of Visual Basic. The top left window is dedicated to be the application's form. The developer cannot draw outside the form's visible area.

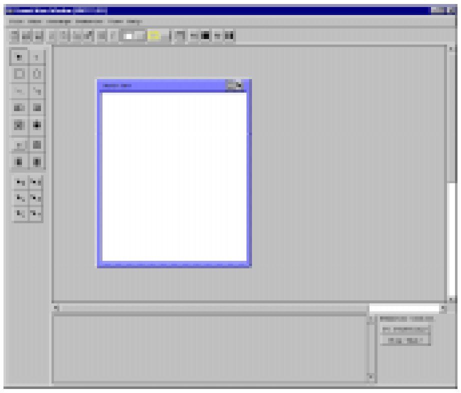


Figure 4.29b: Configuration of Gamut where the application window is a widget within a larger drawing area.

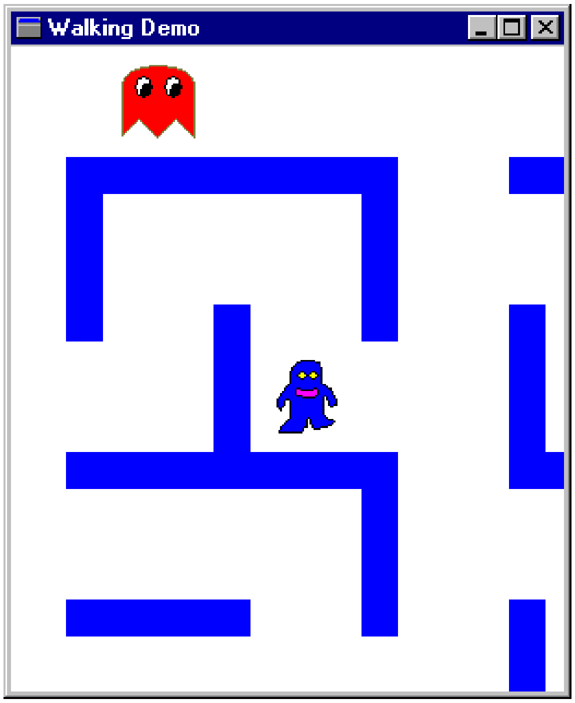


Figure 4.30: The application window as seen from the player's perspective. The background maze seems to shift behind a stationary character.

was written, does not support scaling, Gamut is only able to support translation and not zooming. Changing the size of the frame widget changes the actual size of the application window and does not change the scale of the contents. (Changing the actual position of the application window is not supported in Gamut.) The frame widget has been found to provide a good abstraction for viewports that is easy to use for application effects.

The raised area in Gamut's card editor (see Figure 4.23) can be used in the same way as the frame widget. Objects in the card's background area can be revealed by moving the frame over top of them. This behavior can be used to create scrolling effects as in the frame. It can also be used to make widgets that change appearance. For instance, in a PacMan game, the developer may put a picture of a ghost on a card along with the picture of the same ghost turned blue from a power pellet. The developer could demonstrate that the frame of the card moves to the blue monster when the PacMan eats a power dot.

4.3.5 Player Input Icons

Almost all systems in the past have used a Run/Build mode switch to alternate between player input and developer editing. But a Run/Build mode switch is unwieldy and error-prone so Gamut allows player events to occur while the developer edits. Gamut introduces “player input icons” to allow the developer to generate player mouse events while still in edit mode.

The player input icons (sometimes called mouse icons) represent events the player can perform with the mouse. Events include clicking, dragging, moving, and all the usual mouse idioms. The design of the mouse icons were developed from the icons that were used in Marquise [76]. Gamut defines six icons (see Figure 4.31). The little arrow next to the arrow that looks like the mouse pointer distinguishes the different icons. The arrow with two heads (one pointing up and the other pointing down) means click and the arrow with three heads (one up and two down) represents double-click. Similarly, the icons with the arrow pointing down and the arrow with two heads pointing down are mouse down and double mouse down. These correspond to the first event of a drag. The arrow pointing up represents mouse up which is the last event of a drag. The squiggly arrow is used for mouse move or mouse drag depending on the context. The mouse move event occurs when the developer has not previously dropped a mouse down icon, otherwise the event is a mouse drag. Currently, the developer can only demonstrate events for the leftmost mouse button as Gamut does not have a way to specify the other buttons. A future version of Gamut might use little letters inside the mouse pointer to indicate which mouse button was pressed.

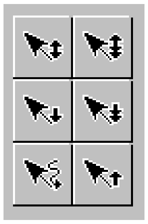


Figure 4.31: The player input icons or “mouse icons.” Each icon represents a kind of mouse event. The double arrows are clicks. The others are down, move/drag, and up event. Double arrows pointing down are double clicking events.

To specify that a mouse event has occurred, the developer selects the event from the input icon palette. Then the developer clicks on the desired event position within the window widget. An icon appears where the developer clicked. The developer is only allowed to drop mouse icons inside the frame widget because player input only occurs within a window.

The effect of dropping a mouse icon is treated similarly in Gamut to performing any other event like pushing a button. If the developer has demonstrated a behavior for the mouse event, then that behavior will occur. The advantage is that the developer does not have to switch to a separate mode in order to create or test behaviors. Furthermore, the developer does not have to emulate the kind of mouse event being generated. The mouse icon palette is used the same way for clicks as for double clicks as for down events, etc. In contrast, a developer using Inference Bear [30] that uses the augmented macro-recorder method must first set a timer, use the mouse to emulate the desired event, and hold the mouse still until the timer expires.

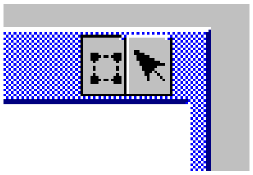


Figure 4.32: Mode switch that activates whether or not mouse events can be performed directly with the mouse.

If the developer wants to test a mouse behavior from the player's perspective, a mode switch exists in the title bar of the frame window widget (see Figure 4.32). Activating the switch lets mouse events occur directly from the mouse pointer. The developer can click, drag, or perform any other demonstrated behavior within the frame. This switch is not a full Run/Build mode switch since it does not effect any other system behavior besides the frame window. The developer can still drop mouse icons if desired and it is still possible to select objects that are not inside the frame. This switch does represent a system mode, though it is not a critical mode because it is not needed to build an application and it is normally not changed frequently. (A true Run mode does exist in Gamut, see Section 4.4.3.)

4.3.5.1 Input Icon State

Gamut permits one input icon to be dropped at a time, but will leave previous icons visible when the situation warrants. For a click event, a single icon is sufficient. Gamut does not distinguish one mouse click from another. During dragging events multiple icons are needed to refer to the different mouse positions in the drag. The drag uses icons to show the starting mouse down location, a movement location, and a mouse up location. Gamut can have as many as three icons on the display at one time.

Gamut defines three possible mouse positions: the “start point,” the “moving point,” and the “last point.” The start point only occurs during dragging events and represents the location of the mouse down event. The last point is used by all mouse behaviors. It represents the current position of the mouse. For a clicking behavior, the last point is just the position of the click and there is no start or moving point. For a dragging behavior, the last point is either the last position of the mouse while the pointer is still being dragged, or it is the up event when the mouse is released. For mouse moving event, the last event is always the last position of the moving mouse.

The moving point is a derived position. When a drag or move occurs, the moving part of the event sequence usually occurs many times. Furthermore, a behavior may depend not just on the last position where the mouse moved but the position right before that as well. When the developer drops move or drag icons onto the window (the icon with the squiggly arrow), the system maintains the last two positions where the icons were placed. The last icon represents the “last point” whereas the second to last represents the “moving point.” For example, a behavior where the player draws a free-hand line (see Figure 4.33a) needs to draw a line between the moving point and the last point. On the other hand, a rubber-band line (Figure 4.33b) only requires the last point and the start point. If the developer demonstrates the creation of a new line for each mouse position but does not use the moving point, the result will be a fan of lines (see Figure 4.33c). Providing the moving point permits Gamut to distinguish these two behaviors.

4.4 Debugging and Testing an Application

Like any programming environment, Gamut cannot tell the difference between erroneous behavior demonstrated by the developer and correct behavior. All bugs must be caught by the developer

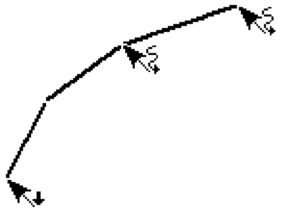


Figure 4.33a: Demonstrating a free hand line behavior requires the developer to have an intermediate moving point.

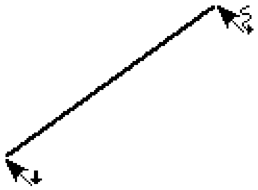


Figure 4.33b: In a rubber band line only the first and last points are needed

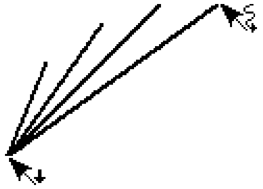


Figure 4.33c: Without the intermediate point, a free-hand line would be a fan.

who can then initiate debugging. Gamut can provide two things to make debugging easier: good feedback to help the developer see problems when they arise and the ability to quickly correct problems once detected. Gamut's nudges techniques described in Section 4.2.1 makes correcting problems simple. When a problem is discovered, the developer can immediately create a new example to correct it. For feedback, Gamut provides continuously visible state and markers to indicate changes in state. To control the state Gamut supplies a completely functional history mechanism with which the developer can move backward and forward in the application's history without having to use undo. Gamut also provides a means to delete behaviors outright, and provides a Run mode so that the developer can see the game as the player would.

4.4.1 Feedback

Using guide objects makes an application's state visible in Gamut. All guide objects are represented graphically so their value is immediately apparent. These guide objects represent all the variables that the application possesses. When the developer causes a behavior to occur, the objects that ought to be affected are inspected visually. If the objects deviate from the desired outcome, then the developer knows that more demonstration is required. Since guide objects are created by the developer, the burden of creating an easily debugged and testable application is still the developer's responsibility.

Gamut also provides feedback mechanisms that show how the application's state has changed. In Section 4.2.2.2, we discussed the temporal ghosts that are shown in Response mode. Gamut also marks newly created objects with a letter "C" because such objects are often hard to notice. Likewise, Gamut marks deleted objects with a letter "D." Other feedback could be added to Gamut as well. Some of these interactions are mentioned in the Section 8.1.1 that covers the "lost techniques" in the Future Work chapter.

4.4.1.1 Creation and Deletion Markers

Gamut must provide special feedback for actions that create and delete objects. Although it would seem like a created object would be obviously visible when it appears, in fact, creating an object can easily have no visible effect on the display as discussed below. Likewise, though it is generally easier to notice when an object is deleted, the developer sometimes needs to select a deleted object, especially when invoking the Stop That interaction.

When the developer creates an object manually, the developer places the new object in the desired position. The developer usually does not place the new object directly over top of another object that looks identical to the one being created. However, a demonstrated behavior may not be as intelligent. Often in a partially demonstrated behavior that involves creation, Gamut may not learn where to place the created object correctly with the first example. Yet, the behavior will still create the object with the expectation that the developer will correct the position if necessary. The creation action makes a copy of a prototype object. That prototype is usually an exact match of the object on the screen that the developer used to demonstrate the create action originally. It matches all properties of the object on the screen including size, color, and position. If the demonstrated behavior makes a copy of the prototype object without changing any of its parameters, the new object is placed in exactly the same location as its prototype, rendering it effectively invisible. Gamut solves this problem by marking created objects with a large letter “C” over their center (shown in Figure 4.34). For consistency, Gamut will mark manually created objects with a “C” as well. When two or more objects are created in the same location at once, the creation marks will grow so that one will surround the other as shown in Figure 4.34. Both the “C” marks as well as the “D” marks discussed below are kept in a separate graphical layer above all objects so they cannot become covered by other graphical objects.

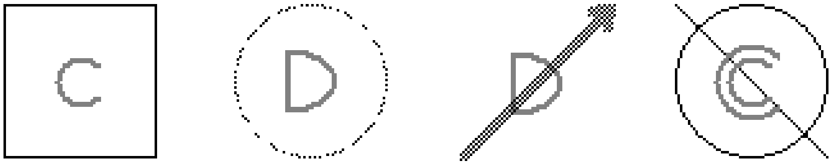


Figure 4.34: Creation and deletion markers. Objects marked with a “C” have just been created. Objects marked with a “D” are deleted (which is sometimes visible on the deleted object’s ghost as well). When two or more markers would overlap, the markers grow so they nest.

Just as created objects have a big “C” printed over their centers, the positions of deleted objects are marked with a big letter “D” as shown in Figure 4.34. In Response mode, the “D” will coincide with the deleted object’s ghost but outside of Response mode only the “D” will appear. The purpose for the deletion markers is different from that of the creation markers. It is not likely that an identical object will be placed directly below an object meant to be deleted. Instead, problems arise because deleted objects no longer exist on the screen and typically cannot be further selected or manipulated. Using Stop That requires that the developer select the object to be stopped. The developer could not select the object’s ghost because ghosts do not exist outside of Response mode. Unlike creation markers, deletion markers are selectable to serve their primary purpose. By selecting the deletion marker of a deleted object, developers can use Stop That and undo the associated delete action so they will not have to redraw the deleted object by hand.

4.4.2 Controlling Timers and History

The history control buttons are part of Gamut’s toolbar as shown in Figure 4.35. The four buttons are used for two functions in which each function uses two buttons. Two buttons are used to move back and forth in the application’s history, and the other two buttons stop and pause timer widgets.

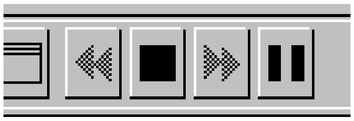


Figure 4.35: The history control buttons are part of the toolbar. The arrow buttons are used to move back and forth in the history and the stop and pause buttons are used to affect timer widgets globally.

4.4.2.1 History Buttons

The buttons marked with arrows are used to control the history. Gamut stores all application history from the time the application is opened to the last event. The arrow pointing left moves Gamut backward in the history while the one pointing right moves forward. The granularity of the history record is at a the level of single events. An event occurs each time the developer performs any operation. For instance, pushing a button is an event. Since editing the interface can affect the history as well, Gamut treats each editing operation as an event, too. For example, creating a new object is an event, and changing a selected group of objects' color is an event. If the developer has demonstrated behavior for a particular event, that behavior is considered part of the event and not separate. For instance, if a button causes a monster to move and change color, the whole behavior including the button press is considered an event. The developer cannot back up over just the part of the behavior where the monster changes color.

Similarly, any editing that occurs during Response mode is treated as a single unit. The system treats examples as though the system had performed the entire behavior in response to the initial event. This allows the developer to back up the state of the application without undoing any examples just demonstrated. (Using undo backs the system into all the dialogs that occurred when an example was demonstrated and will erase an example if undone completely.) This is useful when a behavior can perform different actions from a similar starting configuration. Suppose the developer is making a game piece follow a colored path of lines. When the piece is colored black, it can only follow black lines, when white, it only follows white lines. First, the developer places the piece (originally black) at a fork in the path that has both black and white lines. The developer demonstrates that the piece follows a black path. Then, instead of moving the piece back to its starting position manually, the developer pushes the back button and the system resets the state automatically. The developer manually changes the color of the piece to white and demonstrates the new situation. The developer can continue showing different starting configurations by using the back button and making minor alterations and mode adjustments to demonstrate all the various alternatives.

Use of the history buttons is recorded in Gamut's undo history list, but using a history button is not considered an event itself. For example, if the developer pushes the "back" button four times and the "forward" button once, then the application appears in the state it did three events prior. If the developer uses undo, Gamut undoes the use of "forward" and the application is four events back. Another undo will eliminate one step back and the application will be three steps back again. The history buttons only affect the state of the application and do not affect changes made to the editor itself. For example, if the developer makes onscreen guide objects invisible, the history buttons would not affect that. On the other hand, undo affects all operations the developer performs and thus using undo after making the guide objects invisible would make the guide objects visible again.

4.4.2.2 Global Stop and Pause

The stop and pause buttons are used to control timer objects. These buttons are included with the history buttons because all four buttons are related to time issues. The stop button will turn off all active timer widgets. Having a single point to stop timers is useful because timers can be placed offscreen and may not be immediately accessible. Furthermore, when there is an error in a timer's behavior, the developer may not know immediately which timer is at fault so turning off all timers can prevent the developer from having to search while the wayward timer potentially wreaks havoc on the state of the application. Once the timer is inactive, the developer can search for the bug under more controlled conditions.

The pause button also stops timer events, but it does not affect the timers' state. The pause button only suspends timer events. In other words, though the timers are still turned on, the system does not generate any events for the timers to use, so nothing happens. When pause is deactivated, all timers that are still turned on will once again generate events. Timers that are turned off will continue to remain idle. On the other hand, the stop button will reset the state of all timers so that they are turned off. The stop button is not modal whereas the pause button is. The stop button changes the mode of all the timers to be set to stopped as a single command. The pause button, however, activates a mode where all the timers are made inactive but the timers' mode is left alone.

Like the stop button, pause can be used to stop timers when the developer notices a problem, but pause can also be used to train a timer widget itself. Consider a behavior that activates a timer. It would be difficult to demonstrate turning on or off a timer without activating pause because an active timer would be continuously generating its own events that would cover the events the developer was trying to demonstrate. In Response mode, Gamut automatically pauses all timers so that they will not cause extraneous behaviors to occur while the developer is demonstrating. Once the system leaves Response mode, however, the timers become active again. So, if the developer wants to demonstrate an event that causes a timer to turn off, then activating pause will prevent the timer from intervening with its own event before the developer can press a nudge button.

4.4.3 Run Mode

Gamut does not have a single Run/Build mode switch *per se*. Instead, the system provides a set of switches that each affect a different aspect that most systems collectively consider Run mode. In Section 4.3.5, the switch for activating the mouse was already mentioned, and the switch that makes onscreen guide objects invisible was presented in Section 4.1.2.1. Gamut also has one other switch that causes the application frame window to become an actual window and the editing panes such as the menubar and behavior dialog are hidden.

The last Run mode-like switch causes the blue frame window widget to become a normal window. The switch also causes Gamut's normal main window and any card or deck editor windows to disappear. The developer is not allowed to demonstrate new behaviors or use Gamut's editing commands to affect objects in the frame window. Gamut also displays a small dialog window shown in Figure 4.36. The dialog allows the developer to return to normal editing mode and to switch the other two aspects of Run mode. The developer would be able to make onscreen guide objects visible and invisible and turn on and off mouse input by using the appropriate checkbox. However, since Run mode was never especially useful and cannot be used to create behavior, the dialog was never implemented and the checkboxes do nothing.

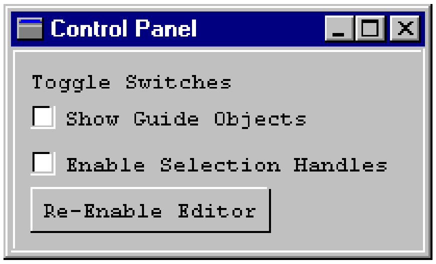


Figure 4.36: Small dialog that controls other aspects of Run mode when Gamut is displaying only the application's window.

The reason the developer might be allowed to enable selection handles as one of the dialog's options is so that a player would be able to use Gamut's usual selection handles to move and grow objects in the application. Currently, if developers want to implement selection handles for their application, they would have to do so by demonstrating everything by hand. However, Gamut does allow the developer to demonstrate behaviors in response to moving objects with the system's selection handles. Thus, it would be possible to let the developer use Gamut's selection handles in the application and have behaviors in the application respond to the player as if the selection handles belonged to the application. However, this idea was never implemented and only its vestiges are apparent in Gamut's interface.

4.4.4 Deleting Behaviors

Sometimes a developer will want to eliminate behaviors from an application so that a new behavior may be demonstrated in its place. Though it is possible to use the "Replace" feature to override old behaviors with newly demonstrated examples (see Section 4.2.3.2.1), it is sometimes easier to start from a clean state.

To delete a behavior, the developer selects the object that causes the behavior's event to occur and uses the "Delete Behavior" item from the menu. If the behavior is attached to a button, the developer would select the button. Similarly, the developer could select the timer or deck if they have an attached behavior. To delete a mouse behavior, the developer selects the blue frame window. Currently, there is no way to delete a behavior attached to some editing commands such as moving an object or changing its color except through demonstration. The developer can always use Stop That or similarly undo a command's behavior and use Replace when the system asks why the behavior has changed. However, this technique may require several demonstrations to fully eliminate the behavior.

When the selected object defines more than one behavior, Gamut presents a list of the events that the object produces. Events with attached behaviors are listed as a set of checkboxes as shown in Figure 4.37. Events with no behavior are not listed. Originally, all the checkboxes begin checked. If the developer wants to keep any of the behaviors, he or she must uncheck those events before pressing okay. Several objects can contain multiple behaviors. For instance, the frame window can contain a separate behavior for each kind of mouse event. The deck widget has two events: one when the deck is shuffled, and the other when the top item of the deck is changed. The timer has three events: the first is the typical "step" event that occurs each time the timer ticks, and it also has separate events for when the timer is turned on and off.

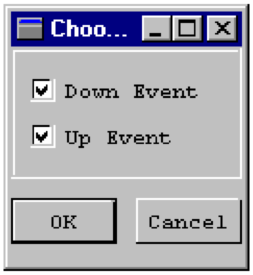


Figure 4.37: Dialog for selecting behaviors to delete. This shows that the frame window has behaviors defined on the mouse down and mouse up events.

4.5 Summary

Gamut uses new interaction techniques in all aspects of its interface. Novel new widgets, feedback, and demonstration techniques make it possible for novice developers to create whole applications without using a written programming language. New widgets like the card and deck allow the developer to prepare complex data and create useful application effects. Using guide objects allows the hidden state of applications to be represented in full. And the nudges technique for demonstrating behavior, along with hint highlighting, is much simplified from other systems' techniques and allows complex, conditional behaviors to be demonstrated efficiently.

Of the new widgets, the card and deck widgets are among the most significant. Cards and decks apply a well-known metaphor to the task of representing complex data structures. Decks are like arrays because they hold a sequence of objects and cards are like records because they group multiple objects into one. Also, decks provide interactive features such as shuffling that would be difficult to provide using any other widget. Furthermore, actual games make use of cards and decks directly. Not only are cards and decks data storage mechanisms, but they can also perform many other roles as parts of an interface.

The concept for guide objects has existed since a draftsman first drew a dotted line to indicate a view that was not visible in the real world. A guide object in Gamut is a graphical object or widget that is visible and manipulable by the developer but is invisible to the player. These include onscreen objects that represent graphical connections and can be made invisible via a menu command as well as offscreen objects which are widgets placed outside the viewable region of the interface used to represent an application's state. Other systems have incorporated guide objects such as the "offstage actors" of Rehearsal World [36] and the "guidewires" in DEMO II [26]. Yet, there has been a reluctance in recent systems to incorporate a general guide object mechanism. For instance, neither Cocoa [22], Director [53], or Grizzly Bear [30] allow the developer to create hideable objects onscreen. Instead, Cocoa and Director use variables in their scripting language to represent hidden state and Grizzly Bear omits hidden state data altogether. Gamut uses the guide object concept extensively because making an application's data visible can help the developer find errors. It also simplifies the heuristics that the inferencing system must provide since guide objects provide structure that would otherwise need to be inferred by Gamut.

The Do Something and Stop That nudges are an easy way to demonstrate behaviors. Unlike the extended macro recorder metaphor that has separate modes for demonstrating both stimulus and response, the nudges technique only uses one mode transition. The stimulus of a behavior is implied by the last event the developer performed while editing. The advantage of this technique is its ability to augment a behavior in any situation, incrementally, as the need arises. This allows a behavior to be constructed in phases so that the developer need not consider every implication and nuance of the behavior from the beginning. It also allows straightforward debugging since a problem can be corrected as soon as it occurs. Also, the technique provides an intuitive method for creating negative examples with the Stop That feature. The developer does not need to enumerate all the situations where a behavior does not apply. Instead, negative cases are allowed to arise within the context of the application and the developer shows how the behavior changes directly.

Complementing nudges is hint highlighting that allows the developer to indicate relationships that would be difficult for the computer to infer on its own. While the developer constructs an example, he or she highlights objects that are relevant to the behavior but are not necessarily modified by the behavior. This allows Gamut to infer modes and conditional expressions that other systems cannot. Most systems require the developer to read and annotate code to add conditional expressions to a behavior. However, Gamut's inferencing algorithm can derive properties from the objects that the developer highlights and can convert these properties into the modes that control an application's behavior.

Gamut also provides useful feedback to remind the developer how the application has changed from one state to the next. In Response mode, Gamut creates temporal ghosts that are dimmed, translucent images of objects in their original state. Temporal ghosts often need to be highlighted when the original state of an object affects the application's behavior. Other feedback includes the creation and deletion markers. Gamut marks objects that have been just recently created or deleted with the letters “C” and “D” respectively. These markers allow the developer to see operations that might go unnoticed if left unmarked. Also, the deletion markers provide a handle for objects that no longer exist so that they may still be manipulated.

In the next two chapters, Gamut's inferencing algorithms are described. It is the inferencing that makes the interaction techniques work. The inferencing algorithms implement Gamut's incremental style and can incorporate Gamut's hints. Later chapters will show how Gamut has been tested both in its initial design phases and after it has been implemented, and they will discuss future work and how Gamut compares to other systems.

Chapter 5: Inferencing Overview

To convert demonstrated examples into functional behavior, Gamut uses inductive learning techniques. Inductive learning or learning by induction is a field in Artificial Intelligence in which the algorithm gathers and analyzes a body of evidence to form a hypothesis [95]. The essential process of induction is to build a data structure that captures the essence of a larger body of evidence into something useful. Gamut's algorithms are designed so that the data structure is never radically changed and is built up incrementally.

In general, inducing behavior is complicated. A typical AI induction problem attempts to "classify" a given example based on its attributes. The AI programmer prepares the example data so that all the relevant concepts and attributes of the examples are known ahead of time. The algorithm's task is to create a function that generalizes the attributes to return the correct concept for a given example. However to infer dynamic behavior, Gamut must not only classify examples, it must also learn the set of concepts and the important attributes that the examples are using.

Gamut's inferencing is more ambitious than most prior programming-by-demonstration systems have attempted. In Gamut, the game application's entire code is determined through demonstration and the developer is not asked to make any decisions that directly affect the structure of that code. As a result, Gamut's algorithms are atypical of the field. Gamut does not use a single inferencing algorithm, but instead hosts a set of algorithms, each of which plays a separate role in determining how the code is produced. The algorithms determine how a new example is compared to the application's existing behavior, how the system generates new code to resolve the differences it finds between old and new behavior, and how the system produces conditional, if-then-like, statements in the code.

Since Gamut's inferencing algorithms are fairly complicated, this chapter is devoted to explaining the process in broad terms and the next chapter will describe the details. First, the chapter will discuss some assumptions that Gamut makes about the developer and how to infer code. Second, the chapter will lay out the structure of the code Gamut infers. This will include defining the terminology the chapter will use to describe Gamut's parts. Second, the chapter will describe the overall process of inferring a behavior. This section will omit most of the details and complications that occur the actual process in order to be more clear. Finally, this chapter will discuss some of the implications of Gamut's algorithms and provide a perspective about how Gamut works.

5.1 Inferencing Goals

Though the developer does not directly manipulate Gamut's inferencing algorithms, the developer is still affected by the environment that the algorithms produce. A system can appear to be ignoring the developer when it does not reach a proper conclusion quickly enough. The system's mistakes can appear unusual (if not malicious) if they do not concur with the developer's own understanding of the behavior. In general, inferencing in a demonstrational environment strives to be efficient and responsive to the developer's demands. Also, when the system makes mistakes, the mistakes should appear to be reasonable; that is, the mistake should not appear to be randomly generated but a simple deviation from the intended behavior.

5.1.1 Inferring Code Iteratively

The essential way that Gamut tries to be responsive to the developer is to produce a functional version of the application after each example the developer gives. The goal is to give the developer something that can be tested immediately. Thus, Gamut does not store multiple examples at a time and try to infer behavior all at once. Instead, the developer is allowed to move quickly between the various behaviors of the application and refine these behaviors as desired.

An important criteria for Gamut's inferencing concerns size and efficiency. In order to be responsive and produce a functional application after each example, the size of a behavior's data structure should not grow exponentially with respect to the number of examples. Since a small number of examples can potentially represent any number of behaviors, a poor representation for the behavior could grow very fast. Ideally, the size of the behavior should be no more than linear with the number of examples. Similarly, the search process that assembles the behavior should also not be exponential with regard to the size of new example or the size of the original behavior. The cost of searching directly affects the time the developer spends waiting for the system to accept a new example. The goal is to accept the example quickly to maintain the developer's attention.

Theoretically, an example consists of the entire state of the interface at the moment that it is presented. Since an example is so potentially large, Gamut does not store the examples individually. Instead, behaviors are represented as compact pieces of code that Gamut revises each time the developer provides a new example. This prevents the size of Gamut's representation from becoming too large. It also allows the impact of a new example to be handled quickly. Since Gamut need only compare new examples to a single (though potentially complicated) behavior, it saves the system from comparing the new example to all previous examples.

Also, having a compact representation for a behavior allows the system to use that representation as the code that the application executes. Gamut does not separate the representation of how a behavior is inferred from how the behavior runs. This also helps to reduce the amount of data the system must track, and it provides a direct way to let the developer test the application after each example is incorporated.

5.1.2 Not Violating the Developer's Beliefs

The inferencing algorithm should not violate the developer's understanding of the behavior that has already been demonstrated. The basic premise is that if the developer were to provide a new example that was completely identical to a previous one, the system should provide the same result. However, this premise is flawed in that it does not recognize that examples provided in an incomplete application are inherently incomplete as well. This stems from two factors. First, the developer's notion of an identical state may presume that the system already knows what properties contribute to that state. Second, it is possible to change the state of the application by adding and removing objects. In essence, the state of the application is unbounded since the developer may create a new object and use its properties to regulate the actions of a previously defined behavior.

Still, a developer will likely be annoyed if the system violates previously defined examples in ways that seem arbitrary. Therefore, the system must make some effort to track the state of objects even when their value does not seem immediately relevant to the current behavior. In this regard, Gamut will track objects that are highlighted as hints. In general, Gamut will try to ensure that examples will be reproduced identically when the state of the highlighted objects are the same. Similarly, when the developer demonstrates a new example, the system will try to change as little of the original behavior as it can in order to violate as few of the developer's past examples as possible.

5.2 Assumptions

Gamut's most basic assumption is that the developer knows the internal workings of the application and will represent it using appropriate guide objects and examples. Consider a video game behavior like a monster in Pacman. The monster must move either toward or away from the player's character depending on whether the Pacman has eaten the larger "power dot" recently. Gamut requires that the developer somehow represent this state. For instance, the monster might use a checkbox to tell it which direction to head as in Figure 5.1. The developer might also use the color of the monster as the modal property (the monster turns blue when it runs away). It is the developer's responsibility to represent this modal state. Gamut cannot infer the correct behavior of the monster without it.

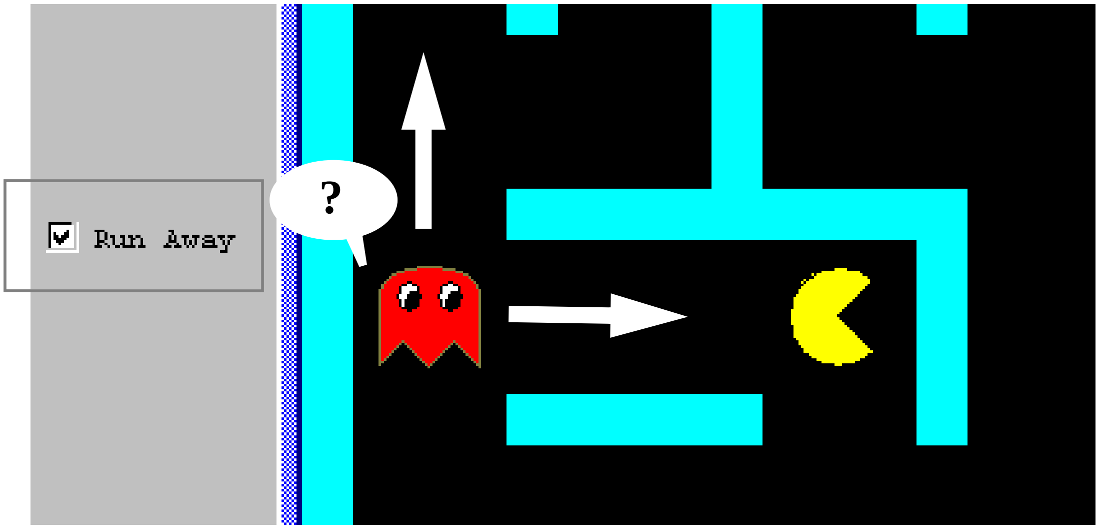


Figure 5.1: The developer uses a checkbox to select whether the monster moves toward or away from the Pacman. It is assumed that the developer will represent this sort of critical state for the system. Gamut would not be able to infer such a condition unless such state is available.

To summarize, Gamut makes assumptions about the abilities of the developer. These assumptions include:

- The developer knows how the intended application works.
- The developer can represent the internal mechanisms of the application using guide objects.
- The developer can demonstrate examples of all the application's behaviors.
- The developer will recognize when the application has not performed the appropriate behavior and will correct the behavior with an example.

Though violating these assumptions will not always prevent a developer from using Gamut, it can reduce the developer's effectiveness. The developer will spend more time making errors and trying to understand why Gamut cannot infer behaviors correctly. Essentially, this time is being spent to allow the developer to grow familiar with the application and to try to understand how to make it work. It is important though, that the problems that a developer encounters is not due to the system. Here, the focus of the assumptions is placed on understanding the application and not on the developer's ability to understand Gamut.

Because Gamut treats examples as the true intentions of the developer, it does not try to modify an example to make it more tractable. Other systems such as Marquise [76] have tried to adjust examples to remove elements that might be caused by sloppy editing. Such systems might adjust lines that are nearly horizontal and make them truly horizontal. Points that are almost aligned might be made perfectly aligned, and so on. Modern graphical editors use techniques such as gridding and snap-dragging [6] so that the developer actually can make objects become aligned as intended. Gamut uses gridding to help developers create the connections they desire and adding other techniques such as snap-dragging would not be difficult (see Section 8.1.4). As a result, the developer can make examples reflect all the connections and alignments without added help from the inferencing system. In fact, experience with Marquise suggested that when the system automatically infers connections that are not present in the example, the developer becomes annoyed and often has limited means to fix the problem. Therefore, Gamut assumes that the developer will draw examples correctly.

5.3 The Structure of Gamut's Internal Language

Gamut's internal language has two purposes: the first is to provide a concise representation of all the developer's examples, and the second is to provide executable code for the application. Since the language must be refined after every example, it must be easy to modify. Furthermore, it must be possible to refine so that a new example may be incorporated with only minor changes to the original structure. This has led to a modular language based on an object-oriented structure. As a result, Gamut's language tends to have more in common with data structures than written programming languages.

5.3.1 Graphical Objects

The basis for everything Gamut learns is derived from the graphical objects in the developer's application. In this case, graphical objects refers to all the things that the developer can draw in the application including rectangles, circles, bitmaps, buttons, guide objects, the window frame, cards, decks, etc. These graphical objects convey the state of the application. When the developer provides a new example, the developer changes the state by manipulating the graphical objects.

5.3.2 Events

The graphical objects also provide the hooks to which the application's behavior is attached. For instance, pushing a button may cause a set of bitmaps to move. These hooks, or *events*, are considered part of the graphical object and the object has pointers that refer directly to the events. Figure 5.2 shows the game Tic-Tac-Toe in which several of the events are marked along with the object that possesses the event. For example, the "Reset" button has a "Trigger" event. A Trigger event is the most basic event because it has no internal values and it does not cause the state of any graphical objects to change on its own. Of course, if the developer wants to use a button in the game, the Trigger event of the button would be augmented so that it performs a game-related behavior.

Other events are more complicated. For instance, the window frame object possesses events that control what happens when the player uses the mouse. The "Mouse Click" event, for example, is used when the player clicks the mouse button while the mouse icon is anywhere inside the window frame (see Figure 5.2). The Mouse Click event contains a value that shows the location where the mouse was clicked. Like the Trigger event, though, the Mouse Click event does not cause the application's state to change.

Some events can cause changes in the application's state. These are the events that the developer uses to edit the graphics and widgets. A simple example is the "Change Property" event that widgets like a checkbox possess (see Figure 5.2). When the developer presses a checkbox, it uses its event to change the its own value from checked to unchecked or *vice versa*. Gamut's own editing operations such as moving and resizing graphical objects or changing an object's color are also represented with events. For instance, when an object is resized, Gamut generates a "Move/Grow" event that has the effect of changing the graphical object's location. Gamut uses events to represent its own editing operations for the sake of internal consistency.

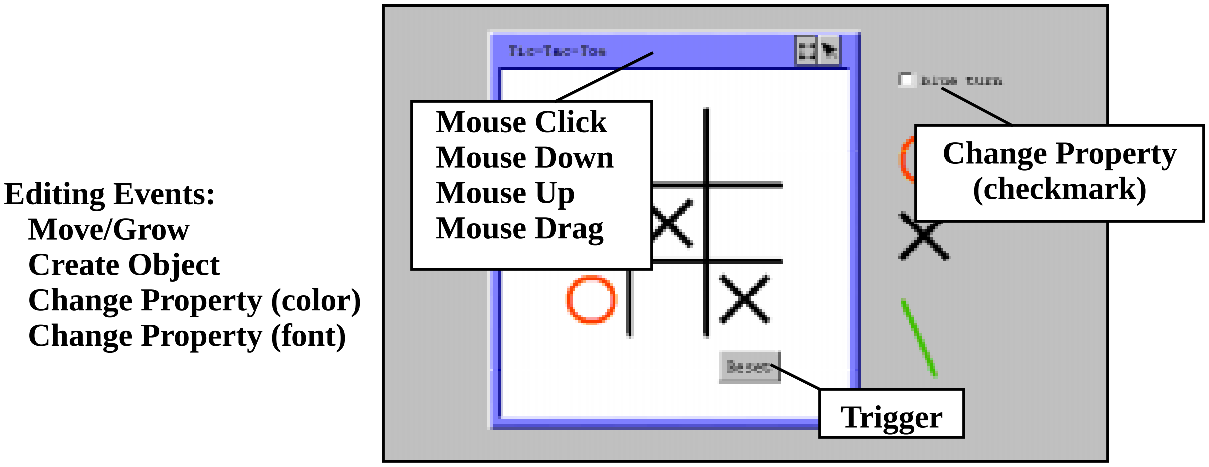
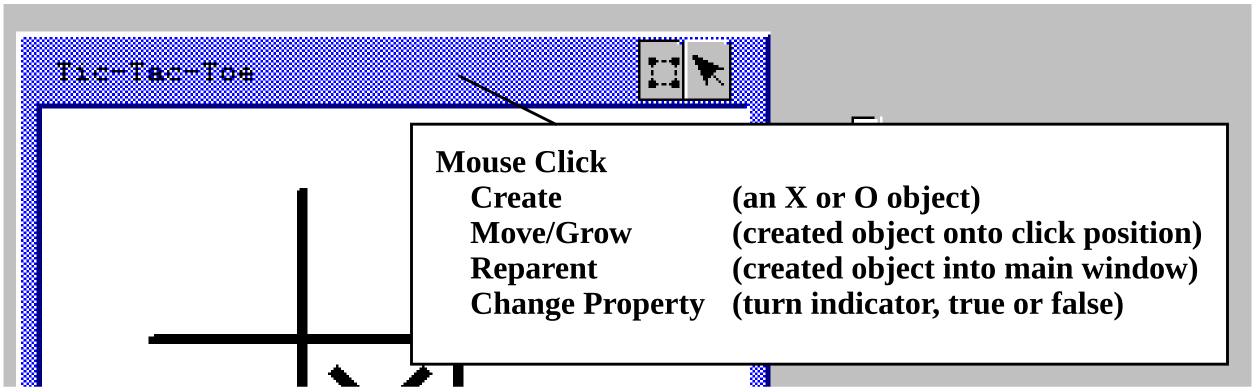


Figure 5.2: Several of the events in this game of Tic-Tac-Toe have been listed along with the graphical object that holds them. The "Reset" button for instance possesses a "Trigger" event. Some events such as the "Mouse Drag" event are not actually used in the game. These events, however, still exist though they have no actions defined for them.

5.3.3 Actions

Demonstrating examples causes Gamut to install *actions* into the events of the application. Actions are only used to change the state of the application. Though the set of events in an application is essentially fixed depending on the graphical objects that the developer has created, Gamut can make as many actions as it needs to define a particular application behavior. A *behavior* in Gamut is defined to be an event that contains actions. For example, the behavior defined for the Mouse Click event in Tic-Tac-Toe is shown in Figure 5.3. An event without actions could either be considered just an event or perhaps as an empty behavior depending on the circumstances.



Mouse Click

Create	**(an X or O object)**
Move/Grow	**(created object onto click position)**
Reparent	**(created object into main window)**
Change Property	**(turn indicator, true or false)**

Figure 5.3: The behavior for the Mouse Click event in Tic-Tac-Toe. The behavior consists of the event itself, plus a set of actions within the event that cause the behavior to change the application's state. Note the comments in parentheses are provided by descriptions that are discussed below.

Gamut's set of actions are designed to represent the minimal set of operations that the developer can perform to change the application's state. Each action is designed to only modify graphical objects in a single, well-defined manner. Gamut uses seven types of actions: Move/Grow, Change Property, Create, Delete, Shuffle, Reparent, and Change Z-order. The operation of most of these actions is readily apparent. The Move/Grow action moves objects or changes their size, whereas the Change Property action is used to modify other graphical properties such as color or font. The Create and Delete actions add and remove objects from the application. The Shuffle action is specific to decks and is used to randomize their contents. The Reparent action is used to move objects between windows, cards, or decks, and it is also used to add new objects to a window, card, or deck. Finally, the Change Z-order action changes the order in which objects in a single window overlap.

It is important to clarify that although some events can change the state of the application, those changes are not performed through actions. In other words, when the player clicks on a checkbox, the mechanism that turns the check on or off is considered part of the checkbox's Change Property event. The changes performed by events are considered intrinsic to the event and Gamut cannot change that behavior. Of course, the developer may add actions to an event that override its intrinsic behavior. For instance, the developer may add a Change Property action to a checkbox that always turns the widget's checkmark off. The apparent behavior of the checkbox would be that if the checkbox was checked the player could turn the check off, but could never turn the check back on.

5.3.4 The Difference Between Events and Actions

In some cases, actions are the direct analogs of certain events that occur in the application. For instance, the Move/Grow action is analogous to the Move/Grow event. In fact, this analogy has inspired some PBD practitioners to meld the two into the same concept. For example, in the DEMO and Pavlov systems there are only “events,” and the system can spawn copies of those events and install them into other events in order to define new behaviors.

Though there is nothing necessarily wrong with this practice, it tends to increase the difficulty of matching events and actions with one another. In general, there is much diversity in the kinds of editing events that the developer might demonstrate. There is one event, for instance, that groups a set of objects. In Gamut, this event is a special case of the “Create” event (because it creates a new group object), but in other systems, it might be labelled as the “Group” event. Grouping has a variety of effects on the developer’s application. It creates a new object that defines the new group. It moves the set of objects that are being grouped so that their coordinates are relative to the group object and not their original window. Lastly, the set of objects are reparented so that they are put inside the new group object. Since the Group event has aspects that create, move, and reparent graphical objects all at the same time, it can interact with any other event that also performs a subset of these operations. Any system that does not reduce a Group event into its constituent parts would have difficulty tracking and resolving the conflicts this sort of interaction can create. In Gamut, the system uses a small set of actions which are designed to be independent. Thus, interactions between actions are kept simple and easy to recognize and handle.

Another reason events and actions are kept separate in Gamut is to simplify the structure of behaviors. In Gamut, a behavior is an event in which actions have been installed through the act of demonstration. If actions and events were the same thing, then it would be possible for the actions within the behavior to contain further sub-actions and thus could be behaviors themselves. (Events can contain actions, actions are events, therefore actions can contain actions.) Matching actions that contain other actions would be difficult since it would be hard to distinguish to which set of actions a given change in the application’s state should be attributed. Gamut avoids this problem by restricting behaviors to a single layer of actions. In other words in Gamut, events can contain actions but actions cannot.

5.3.5 Descriptions

Though actions provide the mechanism for changing the state of the application, the actions still need to refer to the correct graphical objects to change and they need to put the correct values into those objects. Gamut uses *descriptions* to represent the expressions and values that a behavior uses. The name “description” comes from Halbert’s “data descriptions” that he defined in his Smallstar system [39]. Actions and descriptions have parameters that tell the code what to affect and what values to use. Descriptions are always placed in these parameters. Descriptions are designed to be composable pieces of code that can be chained together to form complex expressions.

For example, a set of descriptions in Tic-Tac-Toe returns an “X” object when the player turn checkbox is checked and returns the “O” object, otherwise. Those descriptions are placed in the prototype parameter of the Create action that is located in the Mouse Click event of the Tic-Tac-Toe game (see Figure 5.2 and Figure 5.3). This causes the Create action to create either an “X” or an “O” depending on whose turn it is. Other descriptions determine where the new object will be placed and what value to store in the player turn checkbox.

The descriptions are used to form the constraints between the actions of a behavior with the graphical objects in the application. It is the graphical objects that determine what the actions should be performing. So, when the event of a behavior occurs, the descriptions in that behavior retrieve the information from the graphical objects which is fed into the actions which in turn changes the state of the graphical objects. Figure 5.4 shows a representation of how this cycle occurs. In the figure, time moves to the right. The original state of the application is represented by the block on the left and the block on the right is the new state. One can think of this block as the set of all the values that all the graphical objects possess. The important values from the original state pass through the behavior’s descriptions which are transformed and passed along to the actions. At the end of the line, the actions set values in the new state of the application.

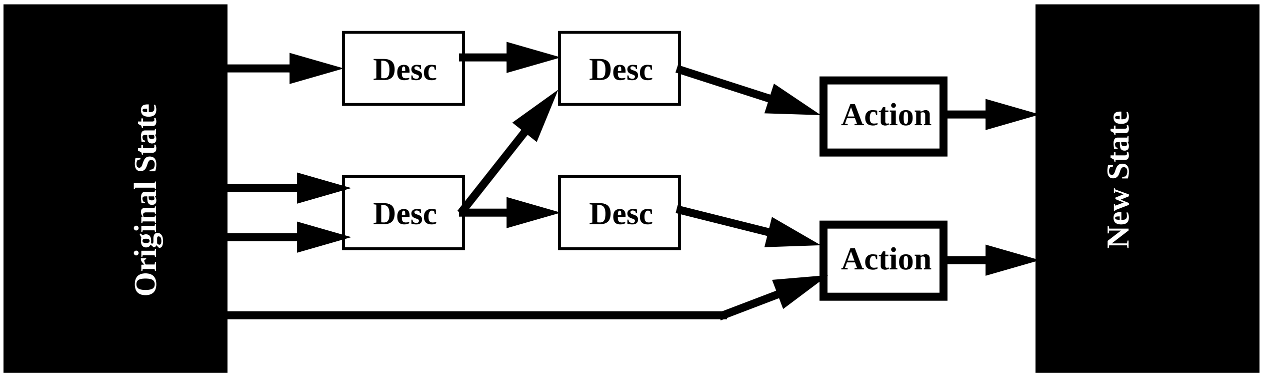


Figure 5.4: In this idealization, a behavior transforms the state of the application. Values from the original state pass through the behavior’s descriptions that transform the data and pass it along to the actions. The actions then set the values of the new state.

In Gamut, descriptions are also the containers that hold the system’s knowledge and heuristics about the various types and values that the application uses. Every description not only knows how to generate a value from its parameters but also knows how to change its parameters so that it could produce different values if it needs to be revised. This feature is used for a critical step during Gamut’s inferencing. Thus, it is the descriptions, more so than any other feature, that determines Gamut’s domain. Gamut contains descriptions that express and can search for graphical dependencies, color changes, simple arithmetic, and other operations that are common in game-like domains. If one were to change Gamut’s domain, the descriptions would be the most affected feature in the inferencing system.

5.3.6 Command Objects

Amulet is the user interface library in which Gamut was written. One of the key aspects of Amulet is that all elements of an application’s interface are represented using objects in Amulet’s object-oriented framework. This includes dynamic elements such as user input. For instance, to make an interactive menu, one adds an *interactor*, specifically a “Choice” interactor, to a group of objects. The interactor carries out all the work and records all the state needed to handle the user’s input. This same philosophy carries over to the commands that one might find in a menu bar as well as the application’s undo history list. Amulet defines *command objects* [75] that represent various operations like cut, copy, and paste. These command objects, like interactors, perform all the needed work and store all the needed state to perform their operation.

The design for Gamut's events, actions, and descriptions language is based on the same architecture that Amulet uses for its command objects. In fact, events are implemented as command objects. Gamut derives a new class of command objects as the base type for events. Events are different from other command objects only in the sense that Gamut uses them to define behaviors and can store actions within them. Actions are similar to command objects though by themselves, actions do not represent a command. An action must be stored in an event to become available to the user. However, actions have the same components that are found command objects so they share the same architecture. On the other hand, descriptions do not share many features with command objects hence they are represented by their own type. However, descriptions do follow the Amulet philosophy of representing an application's features using objects and uses the same architectural features that command objects use.

5.4 Inferring Behavior

In order to keep the developer involved in the process of demonstration, Gamut updates the revised behaviors immediately. Gamut focuses on the event that the developer uses in the example. Since events are static parts of the application, Gamut never needs to create new events or attempt to modify two events simultaneously. Within the event are the actions that the behavior already possesses from previous examples. When Gamut revises a behavior it tries to add actions or descriptions to the existing actions in the behavior to make it conform to the new example. Gamut never removes actions or descriptions, though in some circumstances, previously inferred actions or descriptions may become unreachable. Gamut's inferencing goal is to make as few additions as it can while at the same time trying to modify the most specific descriptions that relate to the values being changed.

This section describes the overall process of Gamut's inferencing algorithms. The process consists of three stages as shown in Figure 5.5. In the first stage, Gamut generates an example trace from the developer's demonstration. This trace is then compared to the actions in the behavior that the developer is modifying in the second stage. In the final stage, the differences found in stage two are resolved by adding new actions and descriptions to the behavior. The new behavior then replaces the original. If the developer revises the same behavior again, Gamut will begin with this new behavior as its starting point and the process continues.

5.4.1 The Example Game

In order to illustrate how this process works, this section shows how a small example behavior is constructed. The application is shown in Figure 5.6. In the application, the developer wants to make the player's piece (a circle graphical object) follow the path around the board each time the player pushes the "Move" button. The die in the center of the board controls how far the piece travels in each step. The developer will demonstrate this behavior with two examples. The first will show how Gamut creates new code starting from an empty behavior. The second will show how Gamut revises an existing behavior to make it conform to a new example.

The developer has included guide objects in the application to represent the key attributes of the game. To represent the path the piece follows, the developer has drawn a chain of arrow lines

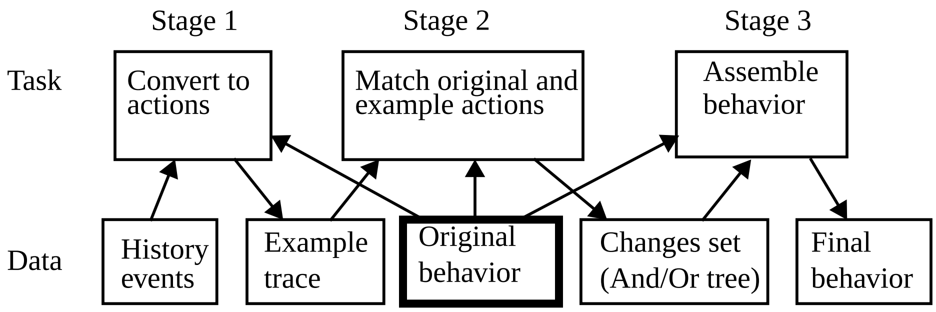


Figure 5.5: Gamut uses three stages to revise a behavior with a new example. In the first stage, Gamut converts the developer's example into a set of actions called the example trace that it then compares to the actions in the original behavior in stage two. In the third stage, Gamut resolves the differences it finds in stage two.

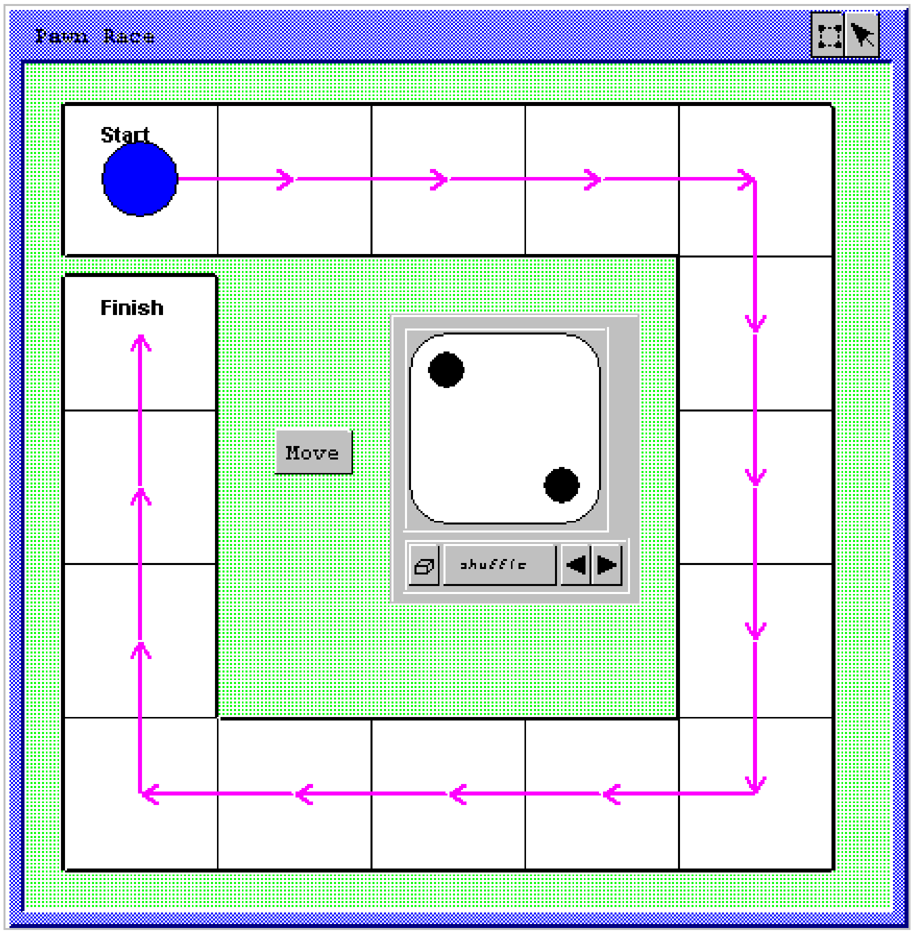
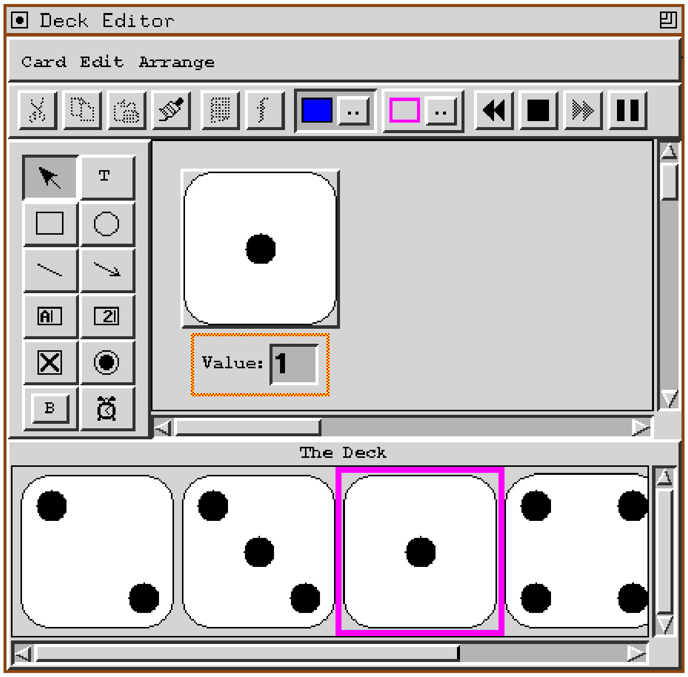


Figure 5.6: In this example game, the developer wants to make the player's piece follow the path around the board. The number of spaces the piece follows is controlled by the die in the center.

around the board. The system will need to recognize that the circle is connected to the lines and that the arrows form a path. The developer has also placed offscreen guide objects into the die. The die is actually constructed using a deck of cards as shown in Figure 5.7. Each face of the die is represented by a different card. In the offscreen portion of each card, the developer has added a number box and has typed in the numeric value of the die's face. The developer has also "pre-highlighted" the number box (see Section 4.3.1.1) so that it will be automatically highlighted when the developer highlights the deck. By installing numeric values directly into the die, the developer has saved the system from having to perform a more complicated operation such as counting the number of dots.



Value: 1

Figure 5.7: The die object that the developer is using is really a deck of cards. Inside each card, the developer has included the numeric value of the face the card represents.

For the first demonstration, the developer will focus on making the piece follow the path. At this point, the developer will ignore the die and just make the piece move a constant number of spaces, specifically two spaces at a time. Of course, the developer could demonstrate the die's involvement as well, but for the sake of this example, the developer will not include the die until the next demonstration.

The developer first pushes the Move button and nothing happens. The Move button's Trigger event begins empty so it has no actions to execute. The developer pushes the Do Something button at which point the system responds by asking the developer to edit the interface. The developer moves the piece, but through happenstance drops the piece in the wrong position. Correcting the error, the developer moves the pieces again and places it in the position two spaces along the path. To finish the demonstration the developer highlights the two arrows that the piece followed and presses the Done button. This first demonstration is illustrated in Figure 5.8.

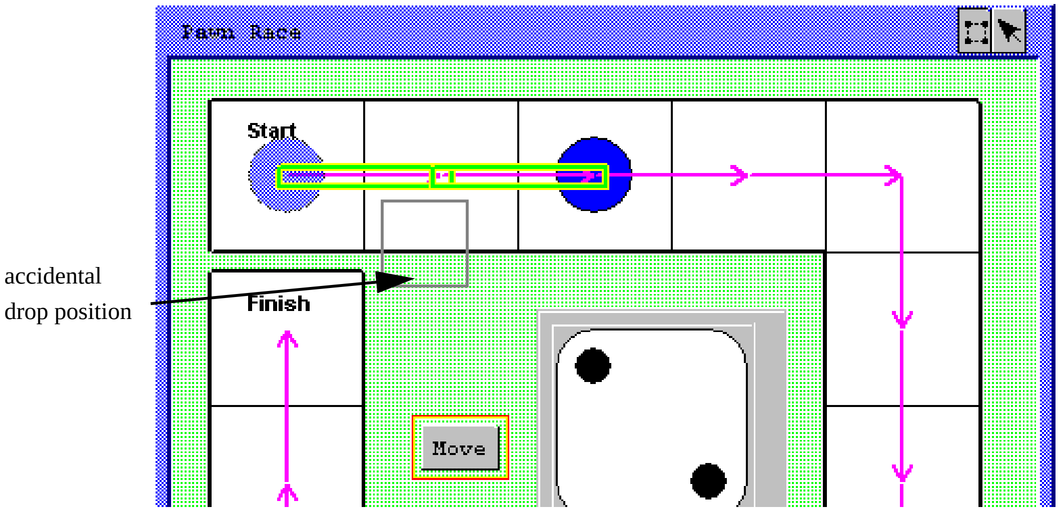


Figure 5.8: In the first demonstration, the developer pushes the Move button and uses Do Something to show that the piece moves two spaces. While moving the piece, the developer accidently drops it in the wrong position, then fixes it. The developer has also highlighted the two arrows the piece followed.

5.4.2 How a Demonstration Becomes an Example Trace

The inferencing process begins as soon as the developer pushes the Done button and completes a demonstration. At this point, the system takes over and starts interpreting what happened. The first step of interpretation is determining precisely what actions the developer performed in the example. This is handled by reading events from the undo history list.

Like many applications, Gamut uses a history list to implement the undo and redo commands. In Amulet, when a command object performs its operation, Amulet makes a copy of the command and stores it on the history list. The copy of the command object contains information captured at the time the command was executed and Amulet uses that information to perform undo and redo. As was mentioned earlier, Gamut's events are command objects. Gamut represents all of the standard editing operations such as moving graphical objects with events. So, when the developer edits the interface, Gamut stores copies of those events in the undo history list. The history list also contains other commands such as the commands for Do Something and the Done buttons. However, commands that are not undoable (such as highlighting an object) will not appear in the list. Figure 5.9 show what the undo history list will look like after the first demonstration.

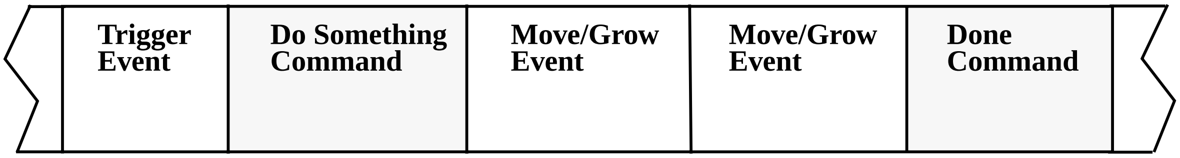


Figure 5.9: After the first demonstration, Gamut's undo history list contains the events that correspond to the developer's example. The actual editing events are enclosed between the Do Something and Done commands. All examples end with the Done command.

By reading the commands off the undo history list, Gamut extracts the developer's example. Every example begins with an event followed by either the Do Something or Stop That command. Examples always end with the Done command. All the events in between represent the developer's example. The system gathers those events and converts them into actions. Then, it eliminates all the redundant actions from that set. The resulting set of actions is the example trace.

Converting events into actions is relatively simple. Each event which has an intrinsic behavior is translated to actions that perform the same behavior. For example, a Move/Grow event simply translates into a Move/Grow action most of the time. A grouping event will be translated into a Create action for the group object, a Move/Grow action to move the other graphical objects into the correct positions, and a Reparent action to place the graphical objects into the group. A Trigger or a Mouse Click event has no intrinsic behavior so they are not converted to actions.

An event can also contain other actions if it happens to define a behavior already. When the event is executed, it will make copies of its actions in the same way that Amulet makes copies of command objects to place in the undo history. The event stores the copied actions in itself so when Amulet copies the event, the copies of the actions will also be present. When Gamut reads the copy of the event from the history list, it takes all the copies of actions it finds in the event and makes them part of the example. In this first demonstration, though, neither the Trigger event nor the Move/Grow events contain actions so this process does not apply.

Gamut eliminates redundant actions so that all actions will become independent of one another. In general, it is easier to handle independent actions because one need not worry about the temporal order in which the actions occur. This allows the actions to be inserted or moved within the behavior without regard for the other actions they might affect. Making actions independent also allows the behavior of each action to be considered separately. Thus, there will only be one action that affects a given property of a graphical object at any particular time. This allows the system to determine which actions are responsible for a given change quickly.

Because the developer slipped and moved the piece twice, the example contains two Move/Grow actions that affect the same graphical object. Gamut takes the two actions and merges them into a single Move/Grow action. During this step, Gamut would also convert any actions that affect multiple objects into multiple actions that each affect a single object. Breaking up actions allows the system to determine more easily which actions are redundant. Figure 5.10 shows how the events from the undo history are converted to actions and then combined to remove redundant actions.

5.4.3 Inferring a New Behavior

After completing stage one, Gamut takes the new set of actions and compares them to the actions in the original behavior. In the example, since the original behavior is empty, Gamut can immediately assume that all the actions are new. Thus, Gamut can essentially skip stage two and move to stage three. Stage two will be discussed when the second example is presented.

Once Gamut determines that there is new code, the goal of inferencing becomes describing the parameters of that code. Actions and descriptions have parameters that can be further refined by installing descriptions within them. Initially, the parameters of new code begin as literal values (called constants). However, if Gamut finds ways to describe those constants using the graphical

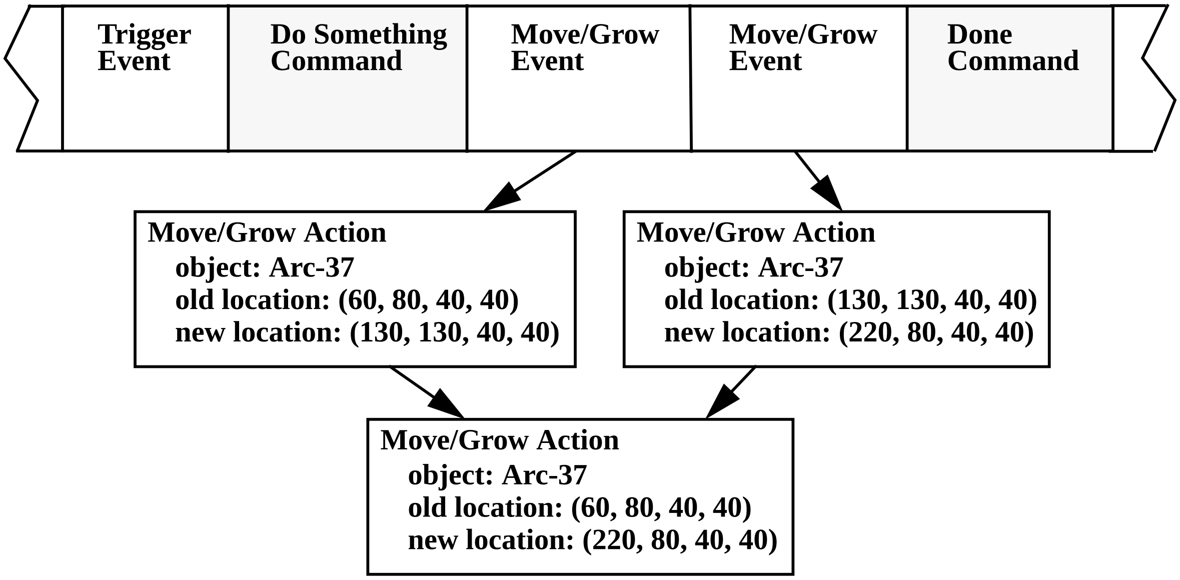

Trigger Event
Do Something Command
Move/Grow Event
Move/Grow Event
Done Command
Move/Grow Action
object: Arc-37
old location: (60, 80, 40, 40)
new location: (130, 130, 40, 40)
Move/Grow Action
object: Arc-37
old location: (130, 130, 40, 40)
new location: (220, 80, 40, 40)
Move/Grow Action
object: Arc-37
old location: (60, 80, 40, 40)
new location: (220, 80, 40, 40)

Figure 5.10: Gamut transforms the events from the developer's demonstration into actions then combines actions that affect the same objects.

objects that the developer highlighted or by re-using descriptions that already exist in the behavior, it will replace the constant value with a description.

5.4.3.1 Autohighlighted Objects

Various aspects of Gamut's interface were tested throughout its implementation. One of the early discoveries was that developers never seem to highlight certain objects. This phenomena also occurred during the usability experiments and is discussed more thoroughly there (see Section 7.2.3.1). One implication of this problem is that the system could not rely on the developer to highlight all the needed objects and would sometimes have to infer that objects were important regardless whether they were highlighted or not.

The kinds of objects the developer would fail to highlight varied but ghost objects were among the most likely to be ignored. Gamut therefore automatically highlights ghost objects under certain conditions. In the current example, the developer has moved the circle object leaving behind a ghost of the circle (see Figure 5.8). Though the developer has not highlighted the ghost, Gamut will treat the ghost as though it were highlighted while inferring descriptions for actions that affect the circle. In particular, the Move/Grow action affects the circle so Gamut will automatically highlight the ghost while the Move/Grow's location parameter is being described.

5.4.3.2 Queuing and Describing New Parameters

Gamut takes the parameters of the new code and puts them into a queue. As Gamut reads an element from the queue, it looks at the value that the parameter contains and tries to find a way to describe it. If Gamut creates a new description by describing the value, the parameters from that description are stored in the queue. In the example, there is one piece of new code, the Move/Grow action that has two parameters: the object to be moved and the location where it will be placed. Through testing, it was found that describing both the object and value parameters of a property-setting action like Move/Grow at the same time was not very fruitful. Often a reasonable description for the object's location could be just as easily used to describe the object itself. Therefore, only the value parameter of the Move/Grow action is queued. (Descriptions for the object parameter can be inferred on subsequent examples.)

Gamut dequeues the value parameter from the head of the queue. The value parameter of a Move/Grow action is a graphical location so it looks for descriptions that return a graphical location. It finds that the Align description suits the purpose well. The Align description is used to assemble a series of graphical constraints into a single location. The Align description searches for highlighted objects that are connected to the location it is describing and it finds that a highlighted arrow line is pointing directly at the center of the circle's location as shown in Figure 5.11a. The Align description records this connection in its data. The Align also finds that the width and height of the circle did not change because the width and height matches width and height of the ghost of the circle that was automatically highlighted. The Align creates two graphical constraints one for the arrow line object and another for the circle's ghost, both of which generate items for the queue.

Gamut then takes another value from the queue. This time it is the arrow line, and Gamut finds that a Chain description will describe it (see Figure 5.11b). A Chain is used to represent a repeating chain of graphical objects that are connected end-to-end. The Chain description determines that the ghost of the circle is the origin of the chain and that the chain is two links long. The origin parameter (the ghost) and the length parameter (2) are stored in the queue.

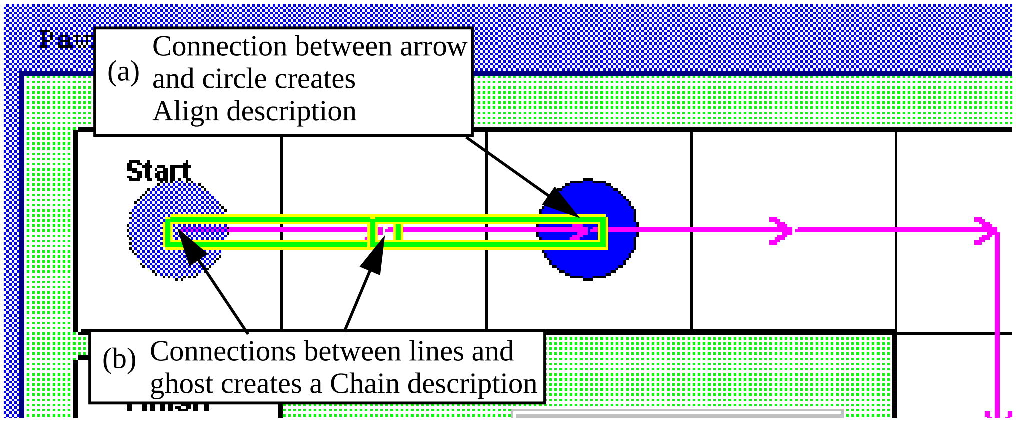

(a) Connection between arrow and circle creates Align description
Start
(b) Connections between lines and ghost creates a Chain description

Figure 5.11: Gamut uses the highlighted arrow lines to create descriptions for the behavior. The connection between the moved circle and the line creates an Align description, the chain of two arrows creates a Chain description, and the ghost of the circle create a Default Object description.

Gamut then dequeues the ghost circle that was stored by the Align description. Since the ghost circle is the same object as the object parameter of the Move/Grow action, Gamut is allowed to install a special description called the "Default Object" (see Figure 5.12). Property-setting actions that are used to modify an object often refer to that object in their value description. The Default Object description provides a simple way to parameterize the description of the value so that the object of the action never needs to be described twice.

When Gamut dequeues the length parameter from the Chain description, Gamut finds that there are no more highlighted objects it can use, so it leaves the parameter constant. When Gamut dequeues the Chain's origin parameter, it finds that it is equal to the object parameter of the Move/Grow action so it uses the Default Object description as before.

The behavior at this point is shown in Figure 5.12. The event is the Trigger event of the Move button. Stored within the event is a single Move/Grow action. The object parameter of the Move/Grow is set to a constant that refers to the circle representing the player's piece. (Circles are called "Arc" objects in Amulet.) The location of the Move/Grow contains the Align description that points to the other descriptions mentioned above. In the diagram, descriptions are represented using thin-lined boxes. Actions are represented using thick-lined boxes, and events are represented using hatched-lined boxes. The arrow lines point from parameters to the descriptions contained in those parameters.

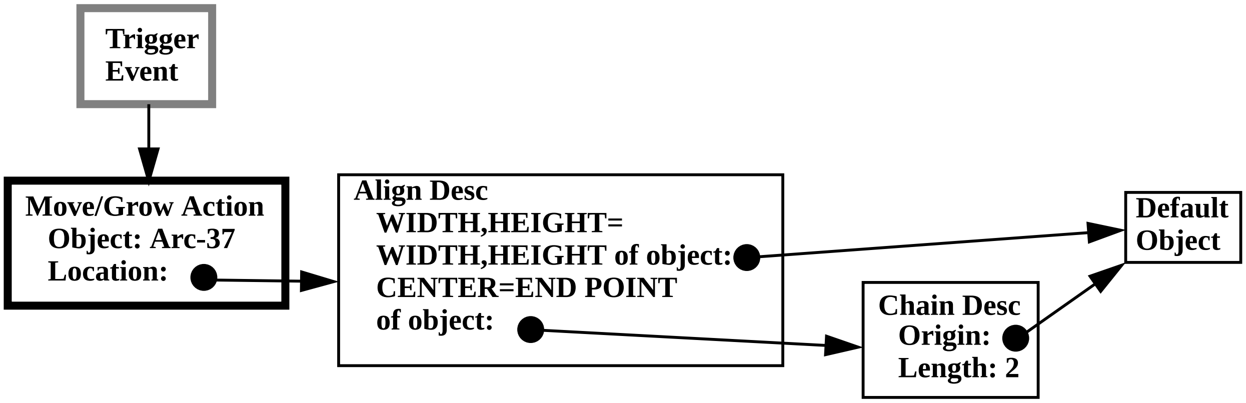


Figure 5.12: This is what the behavior looks like after the first demonstration. It consists of a single Move/Grow action with several descriptions to show how the piece follows the path of arrow lines. The descriptions use the "Default Object" description to refer to the object of the Move/Grow action, Arc-37.

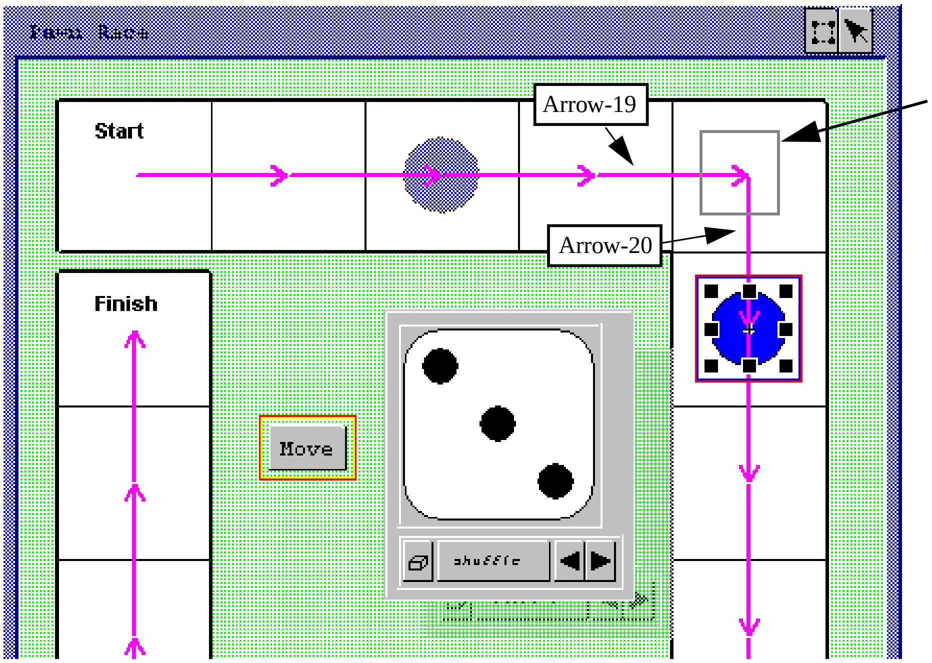
5.4.4 Revising the Existing Behavior

When the developer tests the newly defined behavior, the piece will always move two spaces at a time. The developer would like to refine the behavior so that it will also shuffle the deck of cards that represents the die and have the piece move whatever distance is shown on the die.

To begin demonstrating, the developer pushes the Move button as before. The piece moves two spaces. The die has not been shuffled so the developer does not yet know how far the piece was supposed to move and the developer decides to stop the behavior of the piece. The developer selects the piece and pushes the Stop That button causing the system to move the circle automatically back to its starting position. The developer pushes the "Shuffle" button on the die's deck to pick a random face. The number three happens to appear. The developer moves the piece three spaces and pushes the Done button to signal the end of the example. Figure 5.13 shows the state of the interface at this point.

5.4.4.1 Converting the Second Demonstration to Actions

In stage one, Gamut converts the developers new example into actions just as before. In the new demonstration, the history contains the events shown in Figure 5.14. As before, the demonstration begins with the behavior's event followed by a nudge command, specifically Stop That. Then, the



Piece’s position before pushing Stop That

Figure 5.13: To demonstrate the next example, the developer pushes the Move button, selects the piece and presses Stop That. Then, the developer shuffles the deck for the die and moves the piece to correspond to the number on the die. The developer presses the Done button when finished.

events that the developer demonstrated for the example appear which are a Shuffle event followed by a Move/Grow. The demonstration ends with the Done command.

In the same way as before, Gamut takes the events from the demonstration and converts them to actions. This time, the behavior’s Trigger event contains a Move/Grow action. However, the graphical object selected for the Stop That command is the same object that the Move/Grow action affects. Thus, the Stop That command deletes the Trigger event’s Move/Grow action from the final example as shown in Figure 5.14. The Shuffle event converts to a Shuffle action and the Move/Grow event from the history converts to a Move/Grow action. Since there are no redundant actions, these two actions become the example Gamut will pass on to stage two.

5.4.4.2 Comparing the New Example to the Current Behavior

Unlike the previous demonstration, there is already a behavior defined for the Move button. Therefore, Gamut must compare the new example to the actions in the original behavior shown in Figure 5.12. Comparing example actions to behavior actions is typically straightforward. In this case, there are only two actions in the example and one action in the behavior. The Shuffle action in the example is new and the Move/Grow action in the example partially matches the Move/Grow action in the behavior.

Gamut recognizes three different levels of matching for actions. First, the actions might match identically which occurs when the type and all parameters of the matched actions are the same. Second, the actions might be similar. This occurs when the types of the actions match and all but

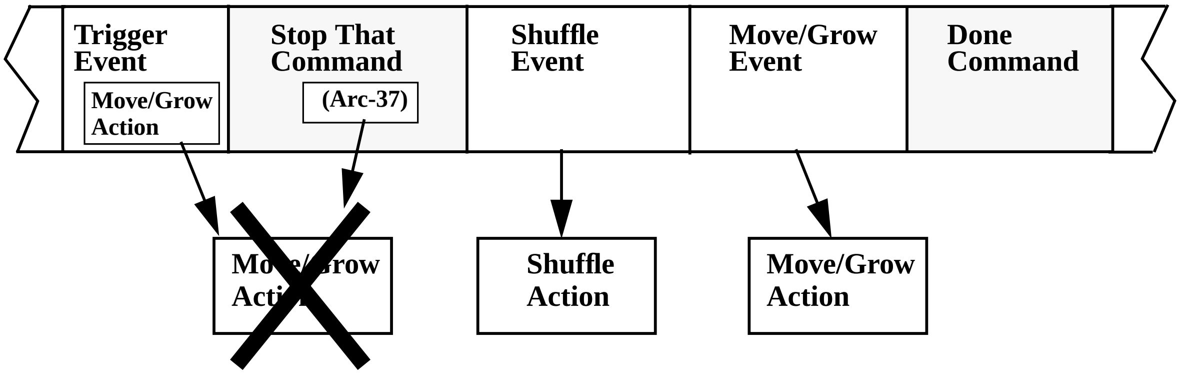


Figure 5.14: The history list contains these events after the second demonstration. The Stop That command eliminates the action contained in the initial Trigger event and the other two events are converted to actions as normal.

one parameter of the actions are the same. If the actions are neither identical nor similar, then Gamut considers the actions not to match at all.

When Gamut compares the Move/Grow action from the example with the Move/Grow action from the behavior, it finds that the object parameters of both actions are the same but the location parameters are different. Gamut treats the two actions as a match and will try to determine why the location parameter of the original action has been changed to the new value. The old and new values of this parameter are stored together and are used to revise the descriptions in that parameter.

Since the Shuffle action from the example did not match any action in the behavior, Gamut will treat it as a new action and add it to the list of actions in the Move button's Trigger event. In Gamut, the order of the actions is never important so Gamut will append the Shuffle action to the end of the list. If there happened to be unmatched actions in the behavior (actions that occured last time but not this time), Gamut would install a switch/case-like statement around those actions in order to make them conditional. Of course, behaviors that already contain switch/case statements may need to have their switching expressions revised in order to revise the behavior. However these complications will be left for the next chapter. The actions of the newly revised behavior is shown in Figure 5.15.

5.4.4.3 Propagating Changed Parameters to Descriptions

Once Gamut finishes comparing actions, it may have found several of the behavior's actions are similar to actions in the example. In these matches, the value of one of the actions' parameters is different. Gamut assumes that the original value of the behavior's parameter has changed to the new value in the trace action. However, the action in the behavior may contain a description. Gamut's next step is to propagate the parameter change to the descriptions stored within the changed parameters.

As previously mentioned, a description contains heuristics that, when given a new value, will try to find ways to change its own parameters to make the description generate the new value. The heuristics in this phase attempt to find options without asking for the developer's help or looking

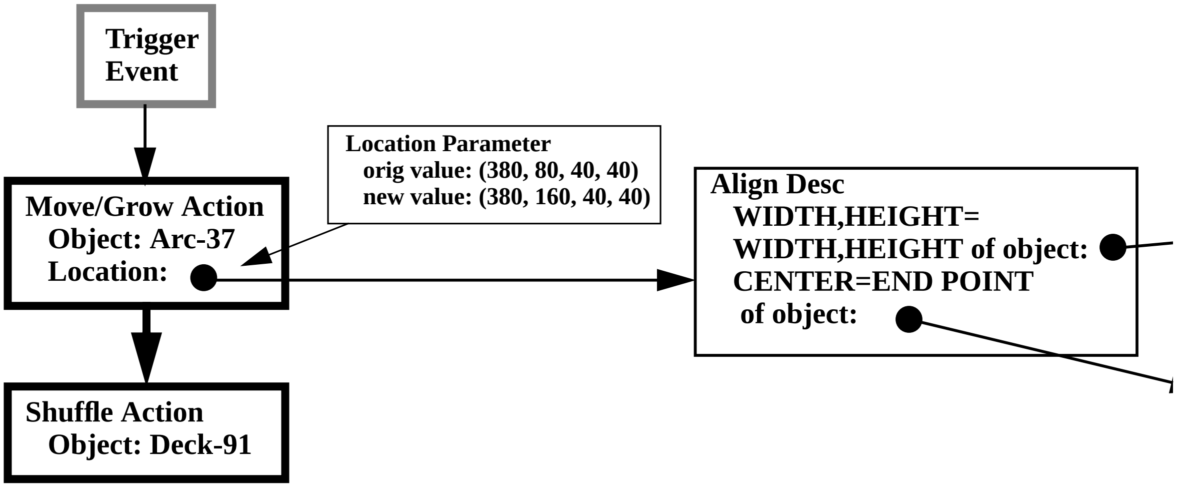


Figure 5.15: Gamut has installed the new Shuffle action into the behavior and has found that the Move/Grow's location parameter has changed value. The thicker line connecting the two actions indicates that the actions are members of a list.

at highlighted objects. The description's heuristics will search the state of the application to find new values that it might use in its parameters. Typically, the description will try to maintain as many of its parameters as it can at the same value in order to prevent the number of changes from exploding exponentially. When a description finds an acceptable way to change its parameters, it will send those changes to any descriptions contained in those parameters. This way, Gamut searches recursively through a behavior's descriptions in order to find different ways to revise the behavior.

Gamut stores the set of potential parameter changes in a single And/Or tree. The And-nodes indicate changes that must occur together and Or-nodes indicate choices. In order to properly revise the behavior, Gamut must select nodes in the And/Or tree so that the parameters of each action that was judged similar during the matching phase is revised.

The diagram in Figure 5.16 shows how the parameter values are propagated for the Move/Grow action in the example. First, the new location value is passed to the Align description (1). The Align's method searches the graphical objects to find new connections that will produce the new location. The Align description was originally using `Arrow-19` (one of the guide objects in the background) to generate the location but it finds that replacing that parameter by `Arrow-20` will generate the correct result. (`Arrow-19` and `Arrow-20` are marked in Figure 5.13). The Align description adds this finding to the And/Or tree then passes the value `Arrow-20` to the description in its parameter (2).

The finding that `Arrow-19` can be changed to `Arrow-20` is represented as a new node in the And/Or tree. It is added as an Or-branch to the node that shows that the Move/Grow's location parameter needs to be changed. This means that the system could either find a new way to describe the location or it could describe `Arrow-20` which would revise the behavior equally well. The Align description may also find other possibilities that it would likewise add to the And/Or tree and pass on recursively.

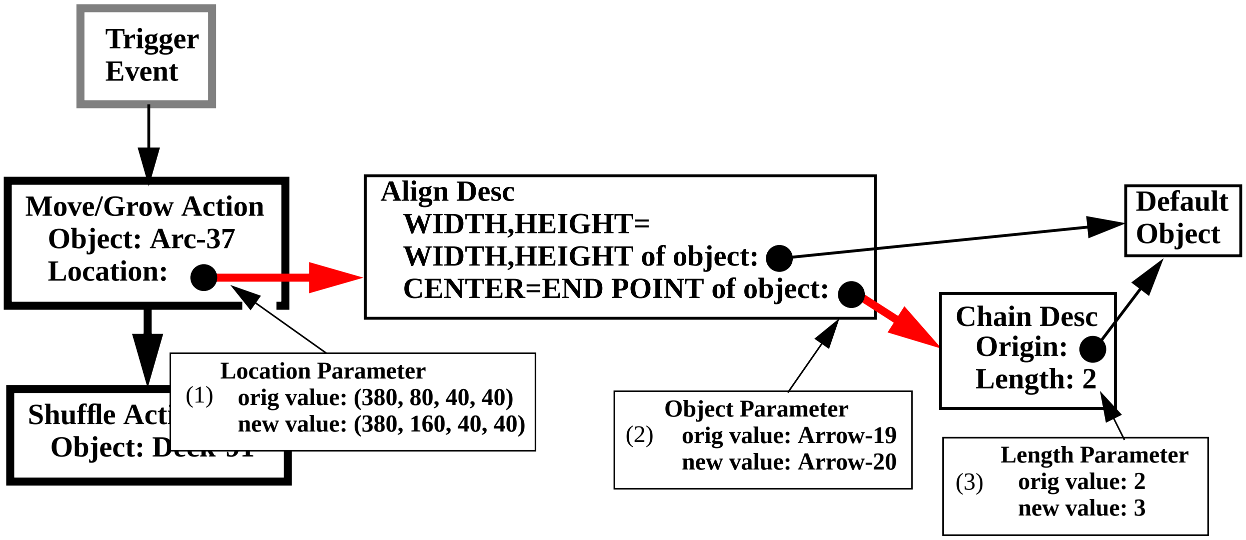


Figure 5.16: Gamut propagates the changes it finds while matching actions into the descriptions of the behavior. Each description contains a method that tries to find ways to revise the description so that it will produce the new value.

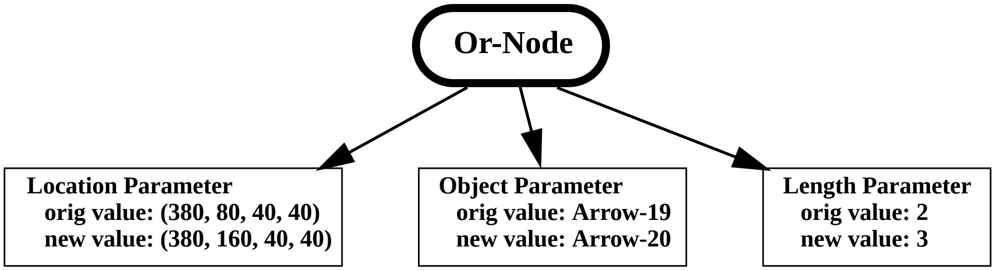


Figure 5.17: The And/Or tree shows the different ways the behavior could be modified so that will properly generate the new example. This And/Or tree shows three possible ways to revise its various parameters. This tree is actually simpler than the actual trees that Gamut produces. Several of the possible changes have been ignored to simplify this example.

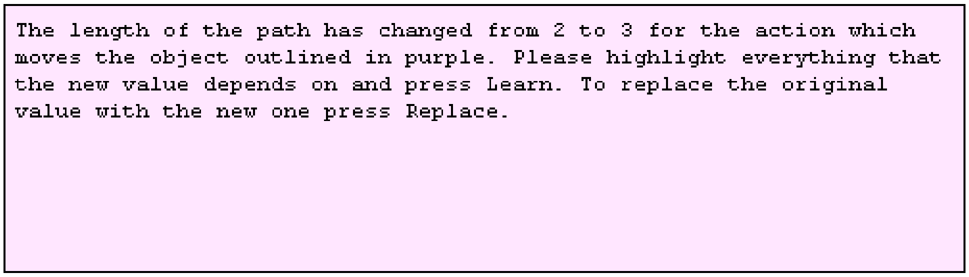
The `Arrow-20` value is passed to the Chain description which likewise searches for new ways to change its parameters. One of the possibilities it finds is that it can change its length parameter from two to three. The Chain description adds this new change to the And/Or tree as another Or-branch to the same node as before (3). This creates the And/Or tree shown in Figure 5.17. The Chain passes the new value onto its parameter, but since the parameter is constant, nothing more happens.

5.4.4.4 Asking a Question to Request Hints

Once Gamut determines a set of changes, it must choose which changes it should make. It will first try to make as many changes as it can automatically. For instance, if the behavior already contains a description that produces a desired value, Gamut can use that description to make a change. Similarly, if the developer has highlighted objects, Gamut will try to create new descriptions using those objects. In the example, however, the developer has not highlighted objects and Gamut cannot find an existing description to resolve any of the changes.

Since the And/Or tree in the example cannot be resolved automatically, Gamut must ask the developer a question. Gamut picks one node from the And/Or tree and converts it into an English question that asks the developer to highlight objects appropriate to resolve that node. Gamut picks nodes from the tree that were generated by descriptions that are the deepest in the description hierarchy. In other words, Gamut picks nodes generated by the longest chains of descriptions in the behavior. These nodes are more likely to be relevant to the new concept the developer is trying to show in the new example. Furthermore, the most distant descriptions signify portions of the behavior that are the most refined, so they questions generated from those description’s nodes are likely to be the most specific.

In the example, the And/Or tree node generated by the deepest description is the one that changes the length parameter of the Chain description. Gamut uses the Chain description to form its question which is shown in Figure 5.18. The question asks a specific question about why the value of the length parameter changed from two to three and tries to put it in words that the developer can understand. Gamut also includes a phrase mentioning the Move/Grow action and marks which object the action affects in order to provide context.



The length of the path has changed from 2 to 3 for the action which
moves the object outlined in purple. Please highlight everything that
the new value depends on and press Learn. To replace the original
value with the new one press Replace.

Figure 5.18: Gamut forms this question from a node in the And/Or tree. It asks why the length parameter of the Chain description has changed uses words relevant to the developer.

The developer is expected to highlight an object to respond to the question. In this case, the developer highlights the die because that is what indicates the number three. When the developer highlights the deck that represents the die, Gamut automatically highlights the top card of the deck. Furthermore, since the developer pre-highlighted an object on each card, Gamut will also automatically highlight the number box inside the three card. Thus, the developer has effectively highlighted three objects, the deck, the top card of the deck, and the number box with the value of three. The developer presses the Learn button to finish answering the question.

5.4.4.5 Resolving the Discovered Differences

The number box with the value three turns out to be a good object for resolving the And/Or node that the system selected. Gamut creates a Get Property description that takes the number box and retrieves its numerical value. Resolving the node also resolves the Or-node for the entire tree, thus Gamut need not ask any more questions. The remaining inferencing will now try to describe the parameters of the new code.

Gamut can also use a third method to resolve differences besides searching for existing descriptions or creating new descriptions based on highlighted objects. If Gamut cannot create a suitable description for a given node in the And/Or tree, Gamut will group the original and new values into a switch/case-like statement and use a decision tree learning algorithm to select between them.

The switch/case statement is called a Choice description in Gamut. If the code already contained a Choice for the parameter in question, Gamut will add the new value to the Choice that is already there. Gamut will use the highlighted objects to create attributes for the decision tree's expression. These attributes are boolean functions that test various properties of the objects the developer highlighted. The attributes are created using heuristics that are similar to the heuristics Gamut uses to create other kinds of descriptions. Gamut will track which attributes were true and false each time the developer demonstrates an example that uses the Choice description. This record is given to the decision tree algorithm that creates the actual expression that selects among the description's various choices. The current example does not require the use of a Choice.

5.4.4.6 Inferring the Remaining Code

Gamut created two new pieces of code for the second demonstration, the new Shuffle action and the Get Property description. Both of these items have object parameters that could potentially be described. Gamut does not find a suitable description for the object parameter of the Shuffle action. Figure 5.19 shows the code that Gamut creates for the Get Property description's parameter. For the number box object, Gamut finds that the number box is contained within the highlighted card object. This creates a Parts of Object description (1) that takes the card and is able to return the card's contents. The Parts of Object description wants to describe the card at which point, the system finds the deck object. Gamut uses the Top Card description (2) to retrieve the top card from the deck. At this point, Gamut has used all the highlighted objects so the deck can no longer be described; thus, it stays constant.

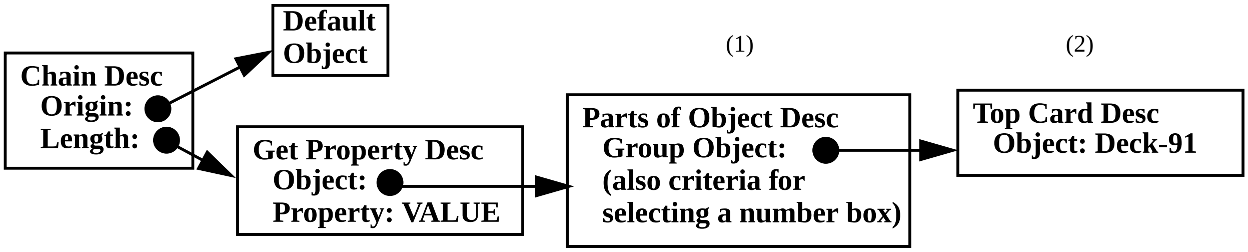


Figure 5.19: This is the code that Gamut produces to refine the new Get Property description. The Get Property gets the number box object from a Parts of Object description which, in turn, uses a Top Card description.

5.4.4.7 Anticipating the Deck's Value

Note that the Top Card description is not referring to the previous top card, but is referring to the top card after the deck has been shuffled. In fact, the piece's location depends, not only on the configuration of the objects in the application, but also on a value that is derived from the behavior. Gamut treats actions in its behaviors independently. The effects of one action are not allowed to affect the result of another, yet here is a case where it seems like the Shuffle action should be affecting the results of the Move/Grow action.

Gamut keeps actions independent by sharing descriptions among actions that use the same results. In this case, the Top Card description needs to use a description that tells it what the shuffled value of the deck will be. In other situations, the process of sharing descriptions can be as simple as copying the value of one action's parameter into another place. In this case, since the shared resource is a random shuffle, Gamut uses a special description called a Virtual Deck to represent the dependency. The Virtual Deck description contains the information needed to determine the result of a shuffle and it can be evaluated even before a corresponding Shuffle action occurs. Gamut will ensure that whenever the actual shuffling is performed, all descriptions and actions will receive the correct value. The final behavior is shown in Figure 5.20.

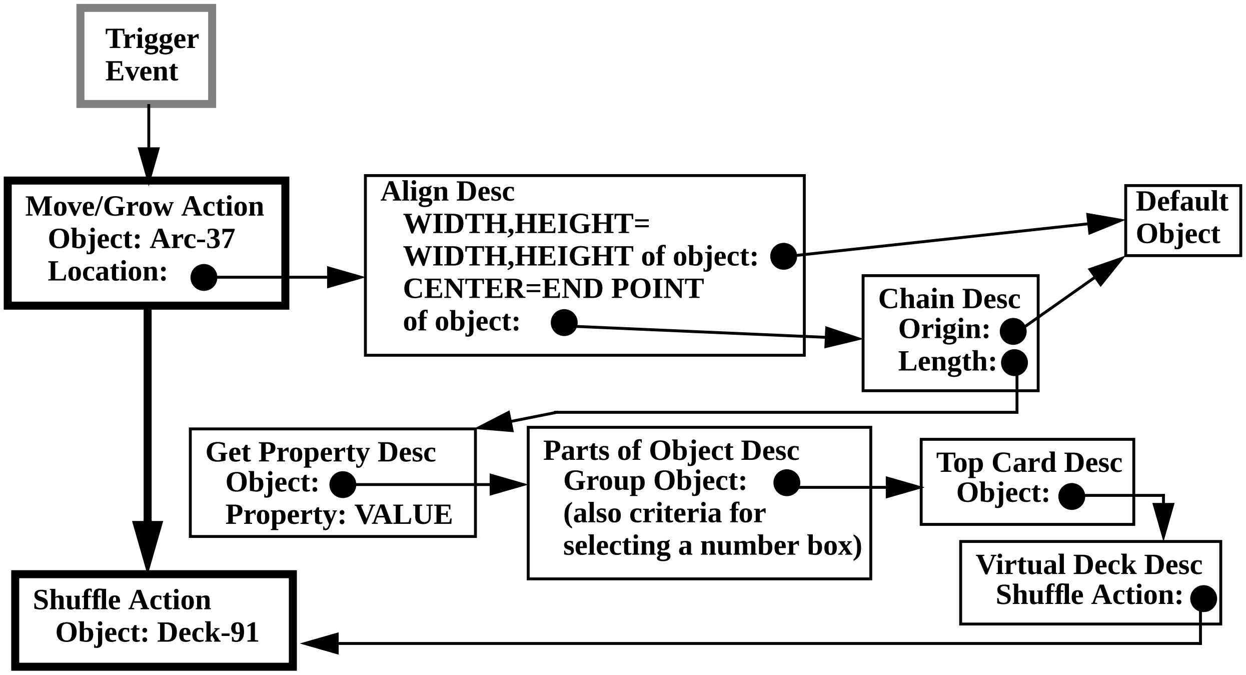


Figure 5.20: Here is the final code that Gamut produces after the second demonstration. The system has added a Virtual Deck description at the end of the chain to represent the dependency between the Move/Grow and Shuffle actions.

5.5 What Gamut Can Infer

Though Gamut infers behaviors that are common in game applications, it infers behaviors at a much lower, and general, level than rules found in games. It is the combination of Gamut's inferencing along with interaction techniques like guide objects and widgets like cards and decks that make Gamut suitable for creating games. Gamut can infer actions on objects including creating them, moving them, or changing their color; it can infer constraints between objects such as one object's color being derived from the color of another; and it can make both actions and constraints conditional based on objects that are not directly affected by the behavior or used in a constraint.

The actions that Gamut can infer are based on the operations that the developer has available during Response mode. Objects can be created, manipulated, and destroyed as one would expect. Gamut can infer some expressions that other systems cannot. For instance, Gamut can infer the creation of a variable number of objects such as the bar chart in Figure 5.21 where the number of objects created in each column depends on the number at the bottom. Also, new objects do not need to be copies of objects displayed in the window and can be derived directly from objects available from the tool palette. Gamut also supports actions that others do not. For example, Gamut can transfer objects between different windows and similar contexts. Most systems only support a single window, but Gamut allows objects to move between multiple drawing areas such as windows, cards, and decks. This action is called “reparenting.” If one considers a container for graphical objects to be a “parent” and the objects it contains to be its “children,” reparenting changes the parent of a given object from one container to another. Since Gamut supports reparenting, it can also support grouping since this also involves reparenting a set of objects into the group object. This allows Gamut to create complex, grouped objects with properties that depend on the environment.

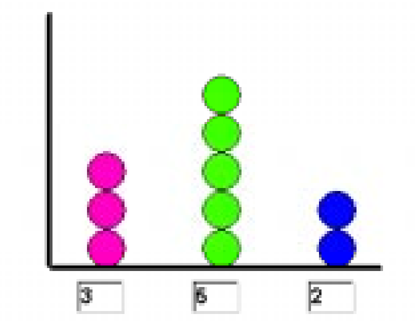


Figure 5.21: Gamut can create a variable number of objects such as in this bar chart where the number of objects created depends on the number below each bar.

Gamut’s description language is built from small composable units that can be grouped and chained together to form more complicated expressions. This contrasts with other systems that use monolithic rules that cannot be split or composed. Furthermore, Gamut uses algorithms that incrementally revise the descriptions to allow each composable unit to be separately modified. For instance, an object’s description can form an arbitrarily long chain of relationships. For example, in the description for a move in a Monopoly game: “the square that is the dice’s number of squares away from the initial position of the current player’s piece.” Gamut’s algorithms can piece together each link of this complex description from simpler components that describe graphical relationships and retrieve the properties of graphical objects. Furthermore, if part of the description were not fully described, Gamut’s algorithms can find the portion that needs to be changed and modify it independently from other parts of the description. For instance, if the distance the piece moved was originally assumed to be a constant number, Gamut can replace that constant with a description for the value on the dice as just shown in the example.

The actions that occur in a behavior and the descriptions that an action uses can be affected by an application’s modes and other factors. For instance in Pacman, the monsters turn blue and run away when the Pacman eats a big dot. The behavior that moves the monsters refers to a mode that is controlled by a completely separate behavior. In other words, objects referenced in a behavior can be independent from the objects affected by the behavior. Gamut can infer conditional behavior where the effect of a behavior is independent from the reason the behavior occurred. Gamut only uses the developer’s examples and hint highlighted objects to do this. Other PBD systems require the developer to specify such relationships by directly editing code.

5.6 What Gamut Cannot Infer

Gamut has limitations that stem from a number of sources. First, there are limits in the interaction techniques that the developer uses to create examples. Gamut cannot read the developer's mind, so the system will inevitably make mistakes. Another source of problems is that developers can make mistakes on their own. Gamut cannot distinguish whether the developer has made a mistake or that the developer is simply not giving enough information. Second, there are domain limitations. Gamut only supports a limited set of widgets and heuristics. Finally, Gamut does not make high-level qualitative judgements about the game the developer is creating. Thus, Gamut only knows the low-level details about an application's behaviors and cannot make judgements about the game as a whole.

5.6.1 Communication Problems

Gamut attempts to infer an application's behavior from the developer's examples. Sometimes it will interpret the examples incorrectly which is to be expected. The developer corrects these errors by demonstrating more examples that help to resolve the ambiguity that confused the system. However, the developer might also demonstrate an example with a mistake in it. For instance, maybe the developer moved the wrong object or changed an object to the wrong color. Gamut cannot know that the developer has made such a mistake, and it will incorporate the example into the behavior in the usual way. To correct behavior derived from bad examples, the developer may need to use stronger measures than demonstrating more examples, such as explicitly telling the system to replace one behavior with another. This is discussed in Section 9.1.5.

Gamut also does not infer application state. Gamut infers relationships between state objects, but cannot generate new state objects by itself. This problem is called the "hidden object" problem in the Artificial Intelligence literature [93] because it concerns the creation of unknown variables not supplied in any example. The hidden object problem stems from a lack of communication between the developer and the system. If the relationship between hidden objects and the behavior is not made concrete, the system will never generalize the examples into a coherent behavior. Thus, it is imperative that the developer reveal all the sources of state to the system. However, the developer may not know that certain state even exists or may not know how to represent the state to the system. Some state can be derived from the visible objects in the game. For instance, the squares on a Chess board show the locations where pieces are allowed to move. However other portions of the state must be represented by extra objects such as counters, switches, and other widgets. For example, Chess requires a switch to know which player's turn it is. Currently, it is the developer's responsibility to represent this kind of state and to properly point it out to the system at the appropriate time.

Some of these problems in communication might be corrected by providing better feedback as discussed in Section 8.1. For instance, the developer might better know that an example contains a mistake by seeing what expressions Gamut has inferred from it. If Gamut does not detect a feature that the developer intended to show, then the developer might correct the problem immediately by re-editing the example. Similarly, Gamut might be able to detect when the developer is not properly representing some piece of state to the system and might be able to provide help. For instance, the developer might be shown a set of examples that solve common mistakes.

5.6.2 Domain Restrictions

Gamut also has domain restrictions. For instance, Gamut's graphics editor cannot rotate objects. A developer wanting to construct the game Asteroids, which involves a small space ship that rotates, cannot use Gamut because it does not support a necessary operation. Some domains would require whole new editors to be added to Gamut. For instance, if the developer wanted to create a 3D interface for a "Doom"-like game or a flight simulator, Gamut would need to have the ability to create and edit three-dimensional objects. Section 8.4 discusses several extensions that would enlarge Gamut's domain.

Similarly, Gamut's heuristics also create domain limitations. Simply adding rotation to Gamut's editor would not be sufficient. If the system is to perform rotation operations, it would be necessary for the heuristic rules to be able to discover rotational relationships and form descriptions from them. Other heuristic limitations exist for objects that are available in Gamut's editor. Section 8.3 discusses a number of Gamut's heuristic limitations and what would be needed to overcome them. Heuristic limitations are harder for the developer to understand than limitations in the editor. To the developer, Gamut simply seems to refuse to understand the examples. Gamut cannot know that a heuristic is missing, nor can a developer add the heuristic through demonstration. New heuristics could be added by writing extensions to Gamut's code in the same way that new widgets or other features could be added. Gamut's inferencing language is extensible (see Section 6.1.3), and new descriptions could be added as long as they do not interfere with others.

5.6.3 High-level Inferencing

Finally, Gamut does not infer high-level, qualitative judgements of an application's behavior. For instance, although one can demonstrate the behavior of a Tic-Tac-Toe game, Gamut will not infer that it is a two-player game. Some might expect Gamut to be able to classify a game into broad categories such as knowing the difference between a war game and a card game. However, Gamut infers behavior at a much lower level, drawing connections between the various input events the player can generate with actions like moving objects and changing their color.

If Gamut could infer higher-level attributes of games, it would be able to use those attributes as properties of other behaviors. For instance, in Chess, a player wins when the other player's king is "in check" and there is no move that would take the king out of check. If Gamut could infer everything that constituents a player's move, it could learn what "in check" meant and would know when the opposing player cannot make a legal move. Since Gamut cannot make such inferences, the developer must represent such concepts using objects that show the hidden state. For instance, the developer might use onscreen guide objects in a Chess game to show when the king is in check, as in Figure 5.22.

5.7 Summary

Gamut's infers behaviors incrementally by adding new details to the currently defined behaviors after each demonstration the developer performs. After each example, the developer may test new behaviors immediately because Gamut represents the behaviors as executable code and refines the code directly.

Gamut's code is comprised of three elements: events, actions, and descriptions. Events are part of the application's interface that represent when a behavior occurs. Typical events include Triggers

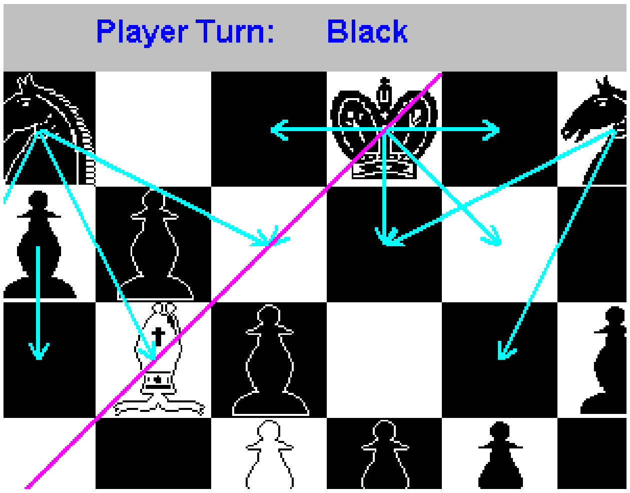


Figure 5.22: The developer uses a guide object line to show that the king is in check. Other lines in the figure show the places where the black player's pieces may move.

that are used in buttons and Mouse Clicks that occur when the player clicks the mouse in the windows. Actions are used to change the application's state. For instance, the Move/Grow action changes graphical objects' position. Finally, Descriptions represent expressions that return values for the parameters of actions and other descriptions. A standard description is Align which returns a location that can be used by a Move/Grow action from a set of graphical constraints.

The inferencing algorithm progresses in three stages. In the first stage, Gamut converts the events that the developer demonstrates into a set of actions to represent the new example. Gamut then compares the new example trace to the actions of the behavior being modified in the second stage. The differences between the example trace and the behavior are propagated into the behavior's descriptions to find a more refined set of differences. In the third stage, Gamut attempts to resolve the differences it finds in stage two. Gamut can resolve the differences by searching for existing descriptions that return the correct value, by generating new descriptions using the graphical objects the developer highlights, or by grouping the different values into a list and using decision tree learning to select among the values.

Gamut's limitations in its inferencing are problems that most AI inductive systems share. The most basic limitation is Gamut's inability to infer hidden state. Gamut cannot create guide objects on the developer's behalf. Instead, the developer must create objects to represent all state within the application and must demonstrate all the relationships between the state.

Though Gamut has limitations, it can infer a broad range of behavior that is found in game-like applications. It can find several kinds of graphical and numerical relationships. It can create and delete graphical objects as well as move objects from one owner to another. It also has deck and card objects that can be used represent groups and collections of objects that other systems cannot. In all, Gamut's inferencing is quite powerful and combined with its interaction techniques, it is reasonably easy to demonstrate entire games using it. The next chapter fills in the details of how Gamut's inferencing works in full.

Chapter 6: Inferencing Implementation

It is more accurate to consider Gamut's inferencing techniques not as a single algorithm but as a set of complementary algorithms. Gamut uses different techniques for inferring different aspects of the application's code. This sets Gamut apart from other systems that have tended to use a single algorithm to infer one form of expression and do not support other expressions at all. One of Gamut's algorithms is borrowed from the Artificial Intelligence literature and is called "decision tree learning" [82]. Gamut uses decision tree learning to classify a behavior when other techniques fail. Gamut's revision algorithm is similar to "plan recognition" [87], but it is structured differently from most plan recognition problems. Gamut uses plan recognition to determine the differences between a behavior and a new example of that behavior. Gamut also uses "heuristic search" [58] which is an algorithm that applies domain specific rules. Gamut uses heuristic search as part of the plan recognition process and to generate new code.

This chapter explains the details of Gamut's inferencing algorithms. Since the overall process of Gamut's inferencing was presented in the previous chapter, this chapter will mostly list the various algorithms and describe their complications. It begins by listing the components of Gamut's internal language and describing their structure. Then, it discusses how a behavior is invoked, followed by a presentation of each stage of inferencing and the algorithms used in that stage. Finally, the chapter ends by describing how Gamut infers geometric constraints.

6.1 Gamut's Language Structure

Gamut's internal language is composed of event, action, and description objects. These objects are based on an object-oriented framework derived from principles in the Amulet system [74]. Although Amulet is written in C++, it defines its own object system which takes its roots from the frame-based style of AI learning [62]. An Amulet object is a dynamically created entity whose data is stored in a list of name-value pairs called properties or slots. These slots can contain either values or methods. Values are data consisting of standard types like integers, floating point numbers, and strings or composite types like other objects, lists, and C++ classes. Methods, on the other hand, are code that defines what the object can do. An Amulet object typically contains a small set of methods that act on the data in the object to achieve whatever effect the object is designed to achieve.

Gamut's three kinds of objects are similar to the language invented by Halbert in his SmallStar system [39]. *Event* objects represent the stimulus that cause behaviors; *action* objects represent the changes to the application's state caused by the behavior's response; and *description* objects are used in action and description parameters to select which objects and properties the actions affect. These objects are similar to the verb-like objects used in semantic networks [98], another AI data structure. The connections between Gamut's code objects and the graphical objects of the application define what parts of an application can cause behavior and what that behavior does.

6.1.1 Events

An *event* object is a specially designated Amulet "command object" [75]. Events are stored in widgets and input objects (called interactors) and are essentially pieces of the application. An event holds methods for causing a behavior to occur as well as performing Undo, Redo, and other system services. The event stores state so that the Undo and Redo methods can restore the information the behavior affected.

Gamut allows the developer to assign behavior to all events including the commands that move and grow objects and set their color. This capability allows the developer to demonstrate "constraint-like" behaviors where graphical objects can respond to changes in other objects similar to behaviors in DEMO [96] and DEMO II [26]. For instance, in a node and line diagram, the developer can demonstrate that any time one of the nodes is moved, the lines attached to the node should follow. This is performed by assigning behavior to the Move/Grow event in the selection handles widget. Though the player never uses the selection handles directly, any behavior the developer demonstrates using the selection handles will incorporate the selection handles' behavior (see Section 6.3.4). Thus, behaviors that the player performs that involve movement will also move the lines keeping them attached.

6.1.1.1 Event Methods and Data

Since events are command objects, they have the same methods that command objects do. Events also have one other significant property that points to its set of actions. The properties of an event are:

Do Method: The procedure that performs the event's activity. In a standard command object, this method is customized to perform the actual command. In an event, this method reads the actions from the event's behavior data and executes them. When it executes the actions, it also makes copies of them for undo.

Undo Method: Called when the system wants to undo the event's activity. Here the event reads its set of actions converted to data and calls the Undo Method on each of them in reverse order.

Redo Method: Called when the system redoes the event's activity. This is the similar to undo except the event calls the data actions' Redo Methods in forward order.

Behavior Data: This holds the set of live actions that contain their descriptions intact. The event uses this property to generate its behavior.

6.1.1.2 Event Types

Gamut categorizes events into these different kinds:

Trigger: This event occurs when a button or similar event-causing widget is activated. This event has no intrinsic parameters.

Mouse Events: These events include mouse up, down, click, and movement events. A mouse event occurs when the developer drops a Mouse Icon into the application window (see Section 4.3.5) and when the system allows the mouse to be used directly in the window. A mouse event's parameter contains the position of the mouse pointer.

Create Object: This event occurs when an object is created by the system. The new object is stored in a parameter. This event occurs when the developer draws a new object from the palette, or duplicates an existing object using the clipboard. It also occurs for grouping in which case, the new object is the created group, and the event contains another parameter that shows what objects were placed inside.

Delete Object: This event means that an object was deleted. The deleted object is stored in a parameter. Normally, this event occurs when the developer uses the Cut command, but it is also used when ungrouping objects similar to the way Create Object is used for grouping. When used for ungrouping, the event uses a second parameter to hold the set of objects that were reparented to the group's window.

Move/Grow: This occurs when an object is moved using the selection handles. If the object is moved from one window to another, the owner is also recorded in the event. Normally, this event is simply used to record movement in an example.

Change Property: This occurs when the developer changes the property of a graphical object. Properties include the colors of the primitive graphical objects like rectangles and lines as well as the values of widgets like the boolean value of a check box and the numeric value of a number field. Note that this means the event attached to a checkbox, number field, and other similar widgets is a Change Property event and not a Trigger. Change Property events are also stored in Gamut's own menus and dialog boxes.

Change Z-Order: This occurs for To Top and To Bottom commands. This event is used when the z-order of a graphical object changes. This event was not fully implemented and is listed here just to be complete.

6.1.2 Actions

Gamut can add *actions* to an event when the developer demonstrates examples. An event with actions is considered a *behavior.* Actions are code objects whose methods change the state of the application.

Each event that can change the application's state implicitly represents some set of actions. So, in some sense, actions can be considered a set of simplified events. Gamut can convert an event into a set of actions that perform the same state change. Each type of action is designed to affect independent graphical properties so that there is no overlap in function and so that actions in a behavior can be independent. Also, whereas events are essentially fixed parts of an application, Gamut can create new actions as needed to represent the effects of demonstrated behaviors.

6.1.2.1 Action Methods

The methods for actions are mostly similar to event methods. Actions contain some extra methods that serve specific roles during inferencing. The methods for actions are:

Do Method: The procedure that performs the action's activity. This method is called by an event's Do Method to perform the behavior's activity. It will store the change it makes to the application into its data object. This data is not only useful for undo but also is essential for the operation of Gamut's inferencing. The inferencing system's examples are constructed from data versions of actions.

Undo Method: This procedure reverses the operation performed by the Do Method. It is called by the event's Undo Method when the system wants to undo an operation. Gamut also can use this method to perform Stop That, and to cancel Response Mode (see Section 4.2.3).

Redo Method: This method re-performs the operation specified by the Do Method. Since the Do Method has already converted its actions into data, the Redo Method only looks at the data to perform its action and does not need to recompute the data from the descriptions. Like undo, this method is called by the event's Redo Method. It is also used to undo a cancel operation out of Response Mode (see Section 4.2.3).

Stop That Method: This method assists during the inferencing algorithm's stage one processing of a Stop That nudge. The method does not actually manipulate the application's state. Instead, it returns a new action that will perform the opposite action of itself. For instance, a Create action will return a Delete action and a Move/Grow will return an action where the old and new positions where the graphical objects were moved are swapped. The graphical objects that the new action affects can be a subset of the objects affected by the original. This method is used when the developer performs Stop That on a subset of the objects that the original action affected (see Section 6.3.3).

Dialog Method: This method returns a list of strings that form a textual description for how the system is trying to revise the action. The text describes what the action did when it was last invoked and what it should be doing when it gets revised. It is called by the inferencing system at the beginning of stage three in order to ask the developer questions (see Section 6.5.5).

6.1.2.2 Action Types

Gamut uses a fixed set of seven action types. Each type was designed so that it would affect state that is as independent as possible from the state affected by other types. The seven actions Gamut defines are:

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

Delete: Removes graphical objects from their window.

Move/Grow: Moves or resizes graphical objects.

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

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

Shuffle: Randomizes one or more decks of cards.

Change Z-Order: Changes the stacking order of one or more graphical objects (like To Top or To Bottom). This action was not implemented.

6.1.3 Descriptions

The objects used to form expressions in the parameters’ other code objects are called *descriptions*. A parameter can hold a constant value like a number or a color, or it can hold a description object that represents an expression. Description objects have parameters which themselves can contain descriptions. For instance, the concept, “the color of the object to which the arrow points” would be represented as a Get Property description whose object parameter is a Connect description which, in turn, refers to the arrow graphical object.

Gamut can form description chains of arbitrary length. Being able to chain descriptions through demonstrated examples is not supported by any previous PBD system. Prior systems such as Chimera [47] only used a fixed set of descriptions with constant parameters that could not be further generalized. Other systems like Marquise [76] and SmallStar [39] permitted descriptions to be nested like Gamut, but did not infer the nesting automatically. Instead, the system would generate a single level of descriptions and if further layers were required, the developer would have to modify the descriptions by hand editing the code.

The descriptions defined in this section are sufficient to demonstrate the example applications that were constructed for this thesis. In general, more types of description would be needed to make Gamut more broadly useful. Section 8.3.1 in the Future Work chapter discusses several ways that Gamut’s descriptions could be expanded.

6.1.3.1 Description Methods

Descriptions are not copied like events and actions. When an action is converted to data for the history list, the descriptions in its parameters are removed and replaced with constant values. As a result, the majority of a description’s methods are used to help the various stages of inferencing and only one is used for evaluation. These are the methods for descriptions:

Evaluation Method: This method will compute the expression that the description represents. It evaluates each of its parameters and uses their values to compute its own value.

Difference Method: This method revises a description’s parameters based on the value it is supposed to return. The Difference Method is basically the opposite of the Evaluation Method. It takes a value and tries to find how the description’s parameters might be changed so it would have returned that value. Each possible change that it finds is stored into the *changes set* data structure (see Section 6.4.4.1) for later processing.

Matching Method: This method tests its own description against another description to see if the two match. Descriptions match if they have compatible types and equal parameters. It is called when the system generates a new description and must be sure that it is not repeating a description that was already created.

Forbidden Method: This method will determine if the description or any description contained within its parameters uses objects or properties that are forbidden in a particular context. This method is necessary to ensure that when a description is discovered through searching, it will be allowed to be placed in the desired parameter.

Dialog Method: This method writes out the expression contained in the description as a list of textual strings and relates how it has been revised by a new example. It is called by the Dialog Method of an action that contains the description as a parameter. Together the action and description's text form a dialog query that asks the developer to highlight parts of the application in order to resolve an issue.

6.1.3.2 Basic Description Types

Gamut uses basic descriptions to generate values and to transform values from one type to another. Gamut defines 13 basic descriptions:

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

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

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

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

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

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

Top Card: Given a deck, returns the object from the list of objects in the deck referenced by the deck's top card index.

Parts Of Object: Given a group-like object that contains other objects (like a deck, card, or group) returns the list of contained objects.

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

Add: Returns the sum of a set of numerical values. Currently only implements adding constants to a numerical property.

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

Virtual Deck: This is an "anticipatory" description (see Section 6.1.3.6). Creates a new deck given a set of objects for the cards and an index for the top item.

Objects From Action: This is another anticipatory description used to fetch a set of objects from an action that can affect the order of objects as a side effect (like shuffling).

6.1.3.3 Attribute Description Types

Gamut defines two special classes of description objects called attributes and variables. These are used under special circumstances. Attribute descriptions are used to generate attributes for decision tree expressions. Attributes are always predicates in Gamut which means that they only return boolean values.

Gamut defines four attribute description objects:

Equal: Tests if two values are equal.

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

Connect Test: Given two locations, return true if they match a given connection criteria.

Min/Max: Returns true if its current value is greater (or less than) all values that it previously computed and false otherwise.

6.1.3.4 Description Variable Types

The other special class of descriptions is called variables which hold values that simply exist within the application and are not computed from parameters. For instance, the Mouse Down description is a variable that represents the location of the mouse. This location is not computed, it is simply the position where the mouse event occurred. The set of variable descriptions that Gamut defines are:

Default Object: In transformation actions like Move/Grow and Change Property, the default object is set to be the object affected by the action.

Mouse Down: The position where the mouse pointer was pressed down.

Mouse Move: The last position where the mouse pointer was moved.

Mouse Up: The position where the mouse pointer was released, or in the case of a clicking event (i.e. not a drag), it is the position of the click.

Created Objects: The set of objects created by a Create Object action.

Current Object: This is used in the Select Object description's attributes. This description represents the current object being examined to determine whether or not it belongs in the set.

Select Index: This is also used in Select Object attributes. This will return the numeric index of the current object that is being examined.

Success Count: One more variable that is used in Select Object attributes. This one returns how many objects in the set have already been determined to belong in the set.

6.1.3.5 Conditions

A *condition* is a special form of description that warrants special mention. Most descriptions form an expression that transforms one or more values into another. For instance, the Add description takes two numerical values and adds them to produce a new value. A condition however forms relationships between different types of data. Gamut defines two conditional descriptions: the Choice description, and the Select Object description. Both descriptions provide operations on a set of values. The Choice description selects and returns one item from a set of values, and the Select Object takes a set of graphical objects and reduces it to a smaller set. Also, both descriptions use *decision tree* expressions (see Section 6.6) to determine which values they return. The Choice description uses the decision tree to pick among its set of branches whereas the Select Object description uses the tree to select the correct objects.

6.1.3.6 Anticipatory Descriptions

Gamut's actions are considered to occur simultaneously. That is, no action is considered to be invoked before or after any other action. Thus, Gamut can eliminate redundant actions in its conversion stage and combine and match actions in its matching stage using fairly simple algorithms because it need not consider the order of actions in the example trace. However, constraints can still occur between the values that are produced by some actions and the descriptions that use those values in another action. When a description refers to a value that is generated by another action (and is not a value in the original state of the application), Gamut uses an anticipatory description to produce the needed value. Though they may seem like a difficult concept, anticipatory actually makes Gamut's algorithms considerably less complicated. Section 9.4.2 describes several of the benefits of Gamut's use of unordered actions and anticipatory descriptions.

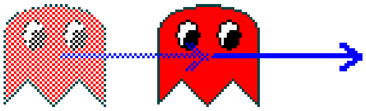


Figure 6.1: The character uses the arrow line in order to move. The character is moved to the end of the arrow and the arrow is corresponding moved back to the center of the character. The ghost objects show the original positions.

Consider the situation shown in Figure 6.1. The developer wants to show how to make the character move in the direction pointed to by the arrow line. The arrow line should also follow along so that the character might be moved again. If Gamut used ordered actions where the result of one action feeds into the next, it might represent this behavior using the code shown in Figure 6.2.

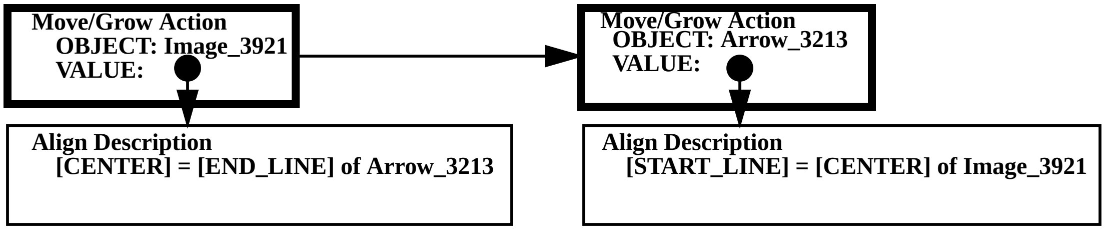


Figure 6.2: If Gamut's actions were ordered the second Move/Grow action's result could depend on the result of the previous Move/Grow as the code shown above does.

In the figure, boxes represent objects. The thick-lined boxes are actions and the thin-lined boxes are descriptions. The parameters to each object are shown inside the box. When a parameter is constant, the value will just be written beside the parameter name. In this case, `Image_3921` is the Pacman ghost character and `Arrow_3213` is the arrow line. Parameters that refer to descriptions contain a pointer to the corresponding object. To represent lists of objects, the objects in the list point to their next element. In this case, the two Move/Grow actions form a list.

In Figure 6.2, the first Move/Grow action places the monster at the end point of the arrow line. This action is fine since it refers to the original state of the arrow line before it moves. However, the second Move/Grow moves the arrow to the position where the character moves. The description in the second Move/Grow does not say that the position of `Image_3921` is actually the new position and not the original position. Thus, there is a conflict. Consider what would happen if the second Move/Grow was supposed to refer to the monster's original position. Since the monster is moved by the first action, the system would have to create a temporary variable to store the monster's position so that it could be used by the later action. This would add a new and quite complicated feature to the language: temporary variables and their associated assignment statements. In Gamut, descriptions in some ways act as temporary variables, but since they are not represented at the level of a behavior's actions (as assignment statements would be), they do not affect how actions are selected or matched against a developer's example.

Requiring actions to be ordered also complicates the process of matching actions in a new example. Assume that for some reason, the developer demonstrates a new example but decides to move the arrow line first before moving the character. If the system were required to maintain the order of actions, it would have to rearrange the developer's example to make it fit the actual behavior. With a large behavior containing conditional segments, rearranging the developer's example in order to recognize becomes challenging. Systems such as Eager [20] have assumed that the developer always performs actions in the same order and fails to recognize the sequence otherwise. By eliminating ordering dependencies such as this, Gamut does not fail when the developer demonstrates an example in a different order.

In constrast to Figure 6.2, Gamut assumes that descriptions always refer to the state of graphical objects *before* they are changed by actions. Values that are produced by an action are expressed by the descriptions in that action. Gamut simply copies the descriptions that produce needed values into the descriptions that need them. Thus, for the example shown in Figure 6.2, Gamut composes the description from the first Move/Grow into the description of the second as shown in Figure 6.3.

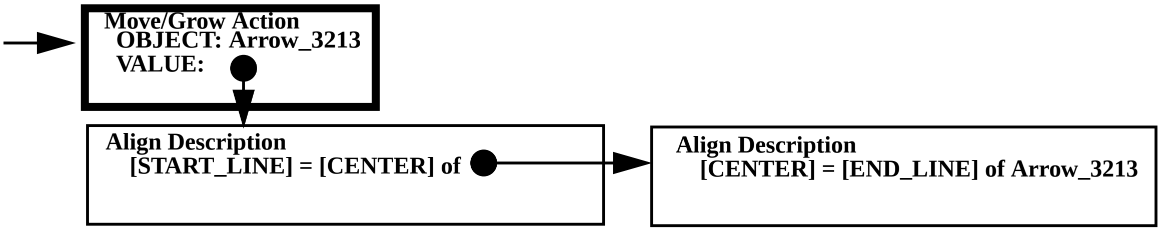


Figure 6.3: The second Align description is made unordered by incorporating the position of the object on which it depends.

The composed Align description returns a position that the other Align description uses to find where the center of the character will be. In this particular case, the constituents of the two Align descriptions can be simplified into a single Align description. Gamut will simplify composed descriptions when it can. The final code is shown in Figure 6.4.

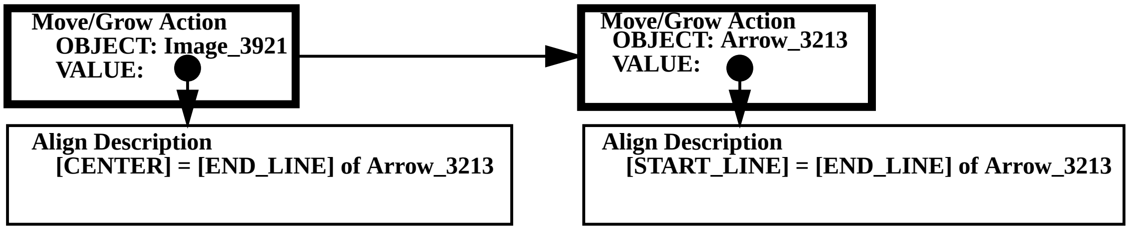


Figure 6.4: Gamut simplifies the composed Align descriptions to form the final actions. The second action has been fixed so that it does not depend on the first.

6.1.3.6.1 Situations Requiring Special Anticipatory Descriptions

In many situations, Gamut can use existing descriptions to anticipate the value for another. In the last example, the Align description from the character's Move/Grow action was copied into the Align description of the arrow line. Sometimes, though, Gamut must use special descriptions in order to anticipate a value. In particular, actions involving operations on created objects must use the Created Objects variable and operations on shuffled objects use the Virtual Deck and Objects From Action descriptions.

When a description depends on a created object, it uses the Create action's Created Objects variable. The Created Objects variable returns the list of objects created for the action, but it also serves a special role. Since Gamut's actions are unordered, it is possible for an action that uses the Created Objects variable to be invoked before the Create Object action that defines the variable. When this occurs, the Created Objects variable will actually create the needed objects ahead of time so that it can return a value. Then, when the Create Object action eventually gets invoked, it will use the objects created by the variable and not create objects again.

When the anticipated value concerns a deck widget, Gamut uses the Virtual Deck description to show how the deck will change. The Virtual Deck description is "virtual" in that it creates a stand-in object for a real deck object that shows how the deck is modified. A deck has two properties, the list of objects it contains and an index that points to the object "on top." Likewise, the Virtual Deck description has two parameters, a list parameter that returns the deck's contents and a numeric parameter that returns the top item's index. When the Virtual Deck is being used to anticipate the value of a Shuffle action, the list parameter contains an Objects From Action description that points to the corresponding Shuffle action. Objects From Action has a generic name because it was intended to be used for other z-order changing actions as well, but this was not implemented. Like the Created Objects variable, Objects From Action can pre-determine the order in which a deck will become shuffled. The list of objects will propagate from the Objects From Action description into the Virtual Deck description where it is combined with the index to become a deck. This is then passed to a description that uses the anticipated value.

6.2 Executing a Behavior

When the developer or player clicks the mouse or uses the keyboard, Gamut retrieves the raw input events and transfers them to the appropriate event object in the application. This is all handled by the underlying Amulet system. For instance, when the player pushes a button, the button widget's interactor looks up its associated command object. That object is one of Gamut's Trigger events. Amulet calls the Trigger event's Do method which will call the Do methods on any actions the event contains. The actions have a nesting structure that controls how if-then-like statements are made. When Gamut executes a behavior it must account for this structure.

6.2.1 Defining a Block of Code

The behavior's actions are stored as a list which is defined as a *block* of code. The list can contain both action objects and Choice descriptions which act as the behavior's conditional statements. When a Choice is used in this way, it no longer functions as just a description, but instead, takes on characteristics of both actions and descriptions. Special consideration occurs in stage two of the inferencing algorithm when Gamut revises these Choice objects. The Choice's branches contain more lists of actions or blocks. Blocks within Choice objects are said to be at a lower *level* than the block where the Choice object itself resides. The structure of Gamut's code is summarized in Figure 6.5. In the figure, the Choice description's VALUES parameter is a list of lists. The parameter contains a list of blocks which are themselves lists of actions and Choice objects. The actions in a block have been connected in the figure with horizontal lines while the list of blocks is connected with vertical lines. Each time it is executed, the Choice will choose a single branch and execute all of its actions. This diagram also shows an event object which is marked by a striped rectangle. Though the figure does not show it, the blocks of a Choice description can contain other Choice descriptions to generate yet lower level blocks.

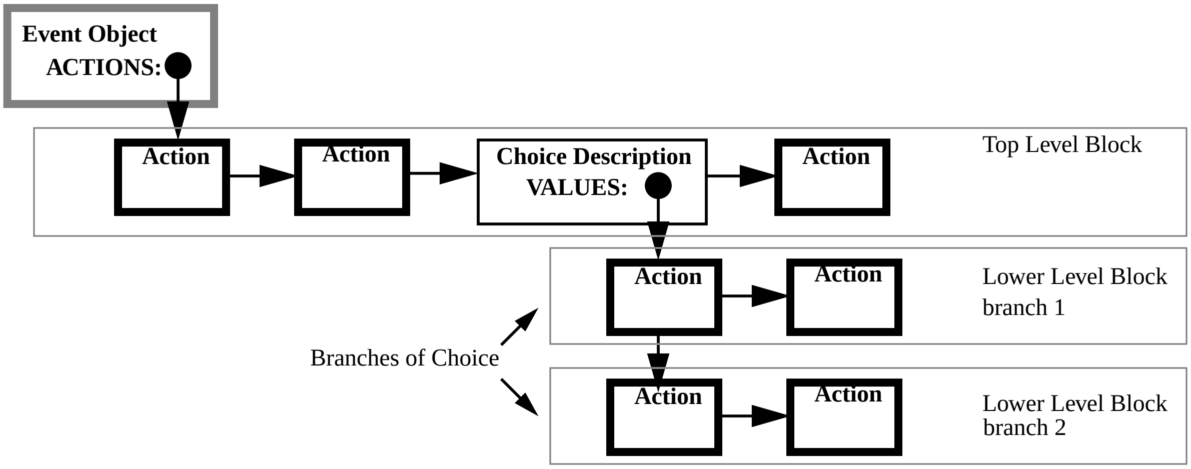


Figure 6.5: Structure of a behavior is Gamut. The event object contains a list of actions called a "block." The list can contain actions as well as Choice descriptions that act as conditional statements. Each time it is executed, the Choice will choose one branch, where each branch can contain more blocks.

6.2.2 Executing a Behavior

When Amulet executes an event, it calls the event's Do method. The Do method, in turn, will read the ACTIONS parameter of the event object and interpret its contents. The interpretation occurs in multiple steps:

- First, top-level Choice descriptions are evaluated so that nested actions can be converted into a single unnested list.
- Next, Gamut separates the Create actions from the other actions in the list. Gamut will evalu-

ate Create actions first so that other actions may have access to the objects they create. The descriptions in the Create actions' parameters are evaluated and replaced with constant values. Then, the Create actions are evaluated by calling their Do methods.

- After the Create actions are computed, the parameters of the remaining actions are evaluated and replaced with constant values and those actions are executed.

Gamut separates the Create actions for historical reasons. Before the Created Objects anticipatory description was added to the system, the system would execute the Create actions first in case other actions that use created objects were somehow rearranged to be executed before their Create actions. Though the introduction of the Created Objects variable eliminated the need to split the actions this way, the code was never changed.

When Amulet executes a command object such as an event, it creates a copy of the object that it stores in the command history list. Thus, each event must store enough state information so that its operation can be undone. To do this, Gamut creates copies of all the executed actions replacing their descriptions with constant parameters. Each action's Do method stores the old values of all the graphical objects that it changes. For instance, if a Set Property action changes a color from red to blue, the Do method will store the value red as well as the object that was changed. The list of copied actions is stored in the event and is called the *history trace* as shown in Figure 6.6. This list is used to implement Undo and Redo and it is used by the inferencing algorithm as the seed that will eventually become a developer's example.

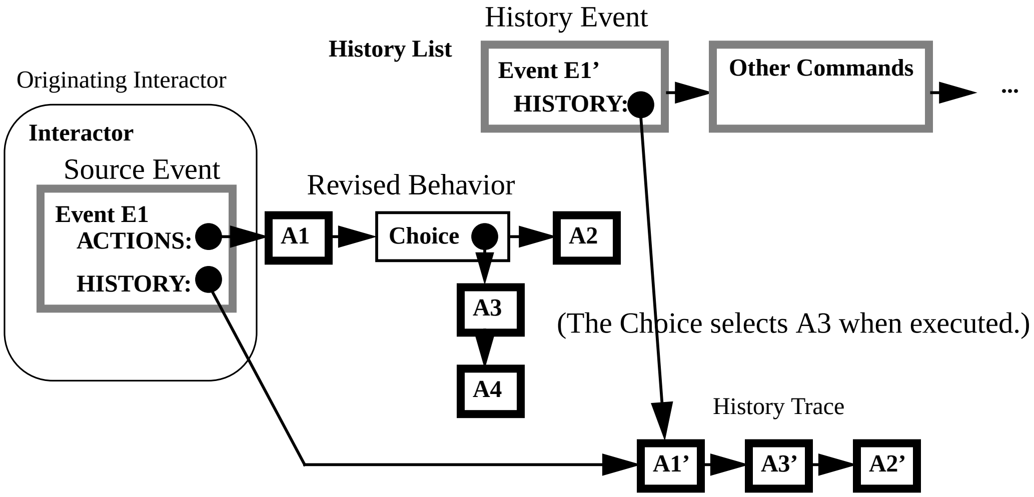


Figure 6.6: When a behavior is executed, it takes its actions, evaluates their parameters and converts them into the history trace which is then stored back into the event. The event, in turn, is copied onto the history list and it, too, will contain a copy of the trace. Copies of objects are denoted by the name of the object ended with an apostrophe.

Because events and actions store copies of themselves in the history list, it is important that Gamut's inferencing keeps track of which object is the original, "live" version of the code object and which is the copied "data" version that was stored in the history. Only the live version contains pointers to the description objects and any changes that Gamut makes to a behavior must occur in the live version. However only the data version of the event and action objects show what actually happened during a demonstration and is used to form example actions.

6.3 Stage One: Converting Events To Actions

As was discussed in Chapter 5, Gamut uses three stages to convert the developer's demonstration into code. In the first stage, Gamut converts the developer's demonstration into a list of actions called the *example trace*. The source of the demonstration is Gamut's undo history list. The latest events and commands in the history determine which behavior the demonstration affects as well as the manner in which the behavior was changed.

6.3.1 The Structure of the Undo History List

Gamut reads the undo history to extract the new example as a list of evaluated events. Figure 6.7 shows how an example looks on the history list before processing. This chapter will use a somewhat different depiction of the history list than the previous chapter to emphasize how it is organized. The list appears backwards because Amulet's history list stores the latest events at the head of the list.

The demonstration begins with the event that the developer wants to modify (which is near the right of the figure). Next is the first nudge's command object. The nudge command objects define the boundaries of example. The nudge commands consist of Do Something, Stop That, Done, Learn, Replace, and other commands that the developer uses to demonstrate an example. A Do Something or Stop That command designates where the behavior's event is located and the point where events that represent how the developer manipulated objects in Response mode began. The command object for the behavior dialog's Done button forms the end of the new example. Between the manipulation events, the developer may also push the Stop button which is in the demonstration dialogs (see Section 4.2.3.1) and acts like Stop That. Stop commands appear as extra nudge commands within the example, but only the Do Something or Stop That command will appear at the beginning.

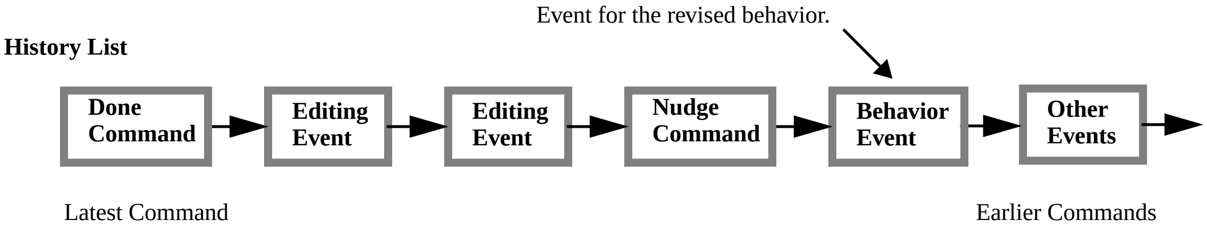


Figure 6.7: This shows how a simple demonstration looks in the history list. At the beginning of the example is a nudge command, either Do Something or Stop That. The behavior's event immediately precedes the nudge. At the end of the demonstration (and the beginning of the list) is a Done command.

The history list can be more complicated if the developer demonstrates two examples back to back for the same behavior. This situation occurs if instead of using or editing the interface, the developer pushes the Do Something or Stop That button immediately after the system completes the analysis of the previous example. Figure 6.8 shows how the history list looks under these circumstances. Between the last Done command and the Done command that completed the previous example, the history list contains another Stop That or Do Something, more Guess and Stop commands, events from the developer’s edits, and also corrective commands generated by the query dialogs. The query dialogs ask questions on behalf of the system that are intended to make the developer highlight objects to resolve ambiguities raised by the new examples. The developer’s responses show up in the history list as Learn, Replace, and Wrong commands. (Commands generated by help dialogs do not change the state of the system hence they are not undoable and never appear in the history list.) This pattern is repeated each time the developer begins a new example without causing a new intervening event. The event command for the entire set of examples begins at the head of the first example.

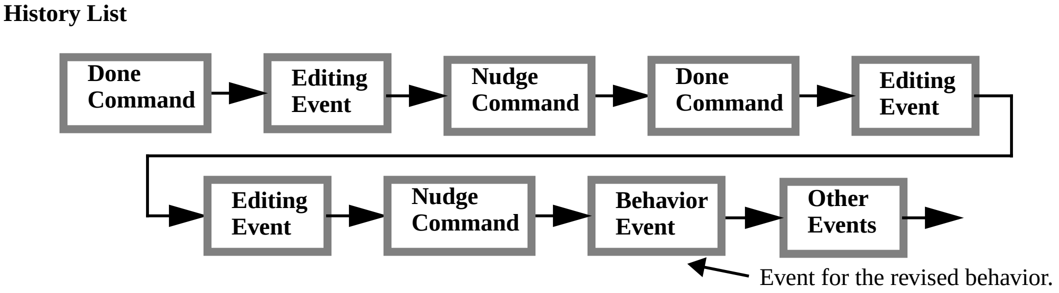


Figure 6.8: In this more complicated demonstration, the developer has performed two nudges in a row without performing an intermediate event. The system must search for the first event in the series to know what behavior to modify.

6.3.2 Actions in Events and Commands from the History List

Events and commands in the history list can contain subordinate actions. First, the revised behavior’s event contains the behavior’s history trace. If more than one example is stored in the history at a time, then the history trace for the behavior up to and including the last example is stored in that example’s Done command. When there are multiple examples in a single demonstration, the inferencing work of the previous examples has already been accomplished. The actions stored in the Done command are simply the results of the previous example’s stage one analysis. These actions summarize all the developer’s actions from the preceding examples including its own actions and the actions from the original behavior.

Actions can also be found in other sources. Events in the Response portion of the demonstration can also contain actions if they are used to define behaviors. Gamut is sometimes able to use the actions from these behaviors to infer descriptions for new actions (see Section 6.3.4 and Section 6.4.3.5). Also, the Stop That and Stop commands also contain the actions that the commands have been used to undo.

6.3.3 Making Stop That Work

Two nudge commands, Stop That and Stop, contain actions that undo parts of a behavior. Hypothetically, the Do Something command could also contain actions. If Do Something were to perform guessing (see Section 8.1.1.1), it would store the guess as new actions, but guessing was not implemented. When the developer selects objects and presses Stop That or Stop, the system determines which actions in the history trace need to be undone. The results of this operation is stored in two lists in the nudge command object: the “Undo” list which records actions that are completely removed from the trace, and the “New Actions” list which records new actions that undo parts of actions in the trace.

When a behavior performs an action that affects multiple objects, the developer sometimes wants to use Stop That on only some of the objects that the action affects. When the set of selected objects covers the entire set of objects that the object affects, Gamut will simply store a pointer to that action in the Undo list. When the selected objects only cover part of the action's objects, Gamut will create a new action that it will store in the New Actions list. The new action will undo the trace action's effect only on the selected objects.

In order to create actions for the New Actions list, the Stop That and Stop commands use each action's Stop That method (see Section 6.1.2.1). The system passes the list of selected objects to the Stop That method. The method, in turn, intersects the selected objects with the objects the action affects. If none of the objects are selected, the method returns and nothing is placed in either list. If all the objects are selected, the method returns a reference to its own action which Gamut will store in the Undo list. When part of the set is selected, the method creates a new action that performs the opposite action as itself. For example, a Create action will return a Delete action and *vice versa*. A Move/Grow or Change Property action will return an action of the same type as itself except the undo and redo value parameters will be reversed. The new action will be set to affect only the selected objects that the original action also affected. The new action is then stored in the New Actions list of the nudge command.

In order for Stop That or Stop to affect the application's interface, Gamut will call the Undo methods of all the actions in the Undo list and the Redo methods of actions in the New Actions list. The developer will see this process as the system undoing the actions on the selected objects. Similarly, when performing undo and redo on the Stop That (or Stop) command, the system refers to the Undo and New Actions lists in order to know what actions the command affected.

6.3.4 Invoking Behaviors Within Response Mode

Gamut incorporates the actions of behaviors executed during Response mode as actions in the revised behavior. For example, in Figure 6.9, the developer uses a button that changes a property during a demonstration. Gamut treats the behavior attached to the button as though the developer had manually performed each of the behavior's actions. If at some later time the developer chooses to modify the button's behavior, it would not affect the behavior that is being currently demonstrated because Gamut does not maintain a link to the button in the revised behavior.

Gamut does not maintain links to sub-behaviors because of the difficulty of representing such links and keeping them up-to-date. It is simply easier to copy sub-behaviors into the behavior being revised than it is to infer behaviors that can potentially possess hierarchies of sub-behaviors. When the developer revises a behavior that happens to be a sub-behavior of another, it is not clear whether the developer wants the sub-behavior to change as well. Gamut assumes that the developer does not want the sub-behavior to change. Researchers building future systems may decide to reverse this assumption.

When Gamut copies the actions of a sub-behavior, it recycles as many of the descriptions contained in the sub-behavior as it can. Each sub-behavior is filtered out of the history list to be presented to the description generation algorithm in stage three of the inferencing process (see Section 6.4.3.5). When the new behavior is assembled, actions from sub-behaviors are filled with their original descriptions. This allows complicated descriptions from one behavior to pass to another without having to be redemonstrated.

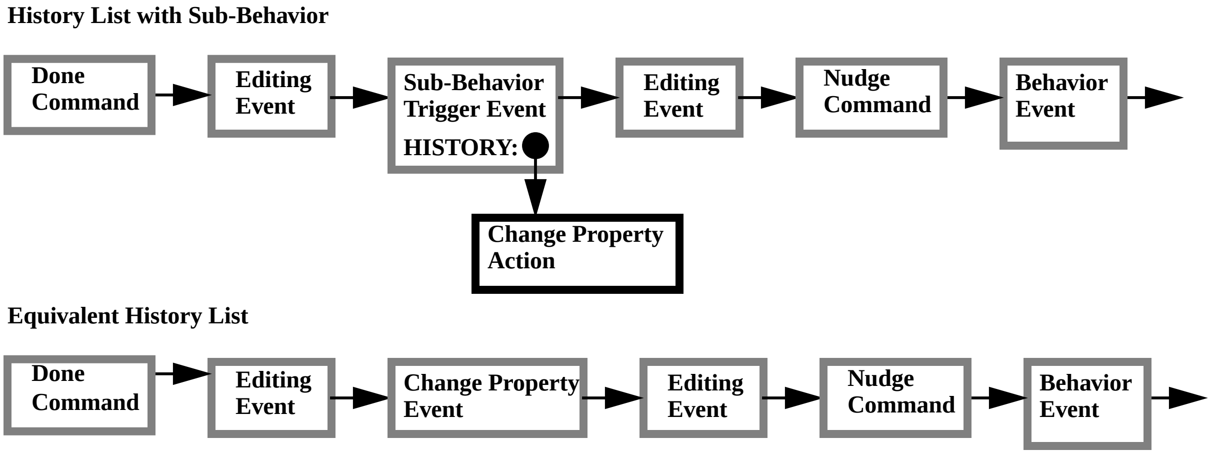


Figure 6.9: The developer has defined a button to perform a Change Property action. During Response mode, the developer pushes the button causing it to execute its behavior as shown in the upper history list. The action in the button's behavior is treated as though the developer performed it manually as in the lower history list.

6.3.5 Creating the Example Trace

As previously mentioned, the goal of stage one is to create the example trace actions for the new behavior. Actions in the example trace are composed from the four sources mentioned above: the history trace from the revised behavior (or the example trace from the Done command of the previous example), editing events generated by the developer, actions from the Undo and New Actions list of Stop That-like commands, and the actions from sub-behaviors invoked within Response mode. Constructing the example trace consists of assembling the various sets of events and actions into a single list, converting events into actions, and eliminating redundant actions.

The algorithm for assembling the list of actions and events follows directly from the structure of the history list. Essentially, the history must first be parsed backwards until the algorithm finds the beginning of the example. The beginning might either be a user input event or it can be an event generated by a previous example being finished. Along the way, the algorithm collects the actions stored within composite events like the nudges and the editing events generated by the developer, also events that cause behaviors to be invoked are discovered and stored in a second list.

6.3.5.1 Converting Events to Actions

Each event that the developer uses in Response mode must be converted into actions. Many events for graphical operations are complicated and require several actions to represent them. For example, grouping objects creates a new group object, changes the selected objects' positions, and reparents the selected objects into the group. Even a simple event like moving an object might be generate multiple actions when the object is moved from one window to another. The process of converting events to actions is mostly a mechanical translation. Gamut knows each event's intrinsic behavior and can convert it into whatever actions are needed.

Sometimes other actions in the history must be examined in order to produce the correct translation for an event. Specifically, the Create Object action must know which properties of the created object are affected by subsequent actions. The Create Object action has two parameters: the prototype object from which new objects are derived, and a count that says how many objects to make. For simplicity, the Create action does not specify expressions that set the initial state of created objects. For instance, if the new object is supposed to be positioned underneath the mouse pointer, then the location is set by a Move/Grow action and not by the Create action. This prevents the Create action from overlapping with any of the state changing actions. Still, the prototype of the Create action needs to be marked to indicate which properties of the prototype are being used. This is needed so that the prototypes of two Create actions can be compared. Properties of the prototypes that are set by other actions must be ignored when they are tested during the matching phase. These properties are marked when the Create action is first created and later reset if new actions are added that affect more of the created object's properties.

6.3.5.2 Eliminating Redundant Actions

Once translation is complete, Gamut removes all redundant actions from the list. It does this by first expanding all actions that affect multiple objects into multiple actions that each affect a single object. Then, it parses the list and combines actions that affect the same property of the same object into a single action. Property changing actions are combined by simply copying the new value field of the second action into the new value field of the first and then destroying the second action. Delete actions will remove all previous actions that affect the destroyed objects. If the destroyed object was also created in the same example, then the delete action is removed as well and no actions will remain that refer to that object.

6.4 Stage Two: Matching Actions and Propagating Differences

In the second stage of inferencing, Gamut revises the behavior's actions comparing them to the newly formed trace actions. Stage two serves two purposes each of which is handled in a distinct step. First, it installs Choice descriptions and rearranges actions in the original behavior to form conditional statements and adds new actions if necessary. The first step is handled by matching the actions in the trace to the actions in the behavior. This will determine how the original actions' parameters have changed as well as which actions were not executed, and which trace actions are new. The process used in this step is similar to the AI technique called "plan recognition" [87]. However, it differs from traditional plan recognition in that there is only one plan against which to match at any one time (though one might consider the various conditions of the behavior to be different plans) and that the plan is not a human-engineered abstraction.

Figure 6.10 shows a pseudo-code description for Gamut's entire inferencing algorithm. The code shows all three of Gamut stages: stage one is lines 1 and 2; stage two is lines 4 through 25; and stage three is lines 27, 28, and 29. Stage two is the most complicated and can be broken roughly into four sections. At the beginning, the trace is matched against the original behavior (lines 4-12). After the matching pass, behavior actions that were judged to be identical or similar to trace actions are used to determine which branches of each Choice description were invoked and which behavior actions were not invoked but exist in invoked branches (lines 14-19). Then, Gamut reconfigures the behavior's actions and adds new Choices and actions (lines 20-22). In the last step of stage two (lines 23-25), Gamut propagates revised values to the descriptions which determines a final set of changes that could be made to the behavior.

The final step of stage two determines how to revise the parameters of active actions. This step propagates the changes found in the original actions' parameters to the description objects con-

```
Get events from command history
Convert events into example trace (see Section 6.3)

Get actions from behavior
Get history trace from behavior
Use history trace to divide behavior’s
      actions into executed and unexecuted actions
Match example trace with executed actions to generate
      similar and identical lists
Add no-op executed actions to identical list
Match example trace with unexecuted actions and add
      results to similar and identical lists

Make copy of behavior
Strip similar and identical actions from copy of behavior
      and example trace
Strip no-op actions from copy of behavior
      // Stripping determines which branch each Choice
      // object takes and produces the set of lost matches
Add new Choice objects and insert new trace actions into code
Promote lost matches to higher level blocks
Restore descriptions from sub-behaviors
Propagate changes to similar actions to parameters
      // This creates the initial Changes set
Add changed conditions to Changes set

Revise Changes set with highlighted objects
Prioritize and form questions from remaining Changes
Use developer’s answers to revise Changes set
```

Figure 6.10: Pseudo-code of Gamut’s inferencing algorithm. This is what Gamut uses to revise actions and build new actions and conditions.

tained within. Each description determines how its parameters change using its *difference* method. This is recursively repeated when the description’s parameters contain descriptions themselves. The final result of this stage is a set of changes one might make to the parameters of the original actions and descriptions in order to make them perform the same operation as the actions in the example. The algorithm Gamut uses in the second step is a heuristic search, and the object-oriented design provides a good framework to apply the heuristics efficiently.

Of course, if there is no original behavior and the invoked event is empty, the action trace simply becomes the new behavior. The new actions are sent directly to stage three (see Section 6.5) where the constant parameters are replaced with descriptions if deemed necessary.

The following sections describe the steps in stage two. The first section will define how Gamut matches actions. Next, the algorithm that uses the matching data to revise the original behavior is shown. Finally, the algorithm that Gamut uses to propagate revision decisions to the descriptions in the code’s parameters is presented.

6.4.1 Matching Actions

In the first step of revision, Gamut matches actions in the example trace to actions in the revised behavior. The goal is to determine which behavior actions were executed, which were not, and which actions in the example are new. The matching process is straightforward for two reasons. First, the number of actions in a behavior is usually small. A behavior contains only enough actions to modify the parts of the interface that change. So, although a monster may move in a complicated manner, its movements would only be represented by a single Move/Grow action. Most of the statements that describe complicated behavior in a typical programming language are assignment statements and procedure calls that compute the various parameters for an object. In Gamut, these expressions are entirely represented by descriptions and do not appear as actions. A behavior with more than five actions is uncommon. The second reason that matching is straightforward is that Gamut's actions are designed to be independent and unordered. The matching algorithm does not need to test cases where one action is not allowed to match against another because they are out of order.

6.4.1.1 Definition of Similarity

The degree to which two actions match is called its *similarity* and is measured by comparing the type of the actions and counting the number of matching parameters. First of all, the types of the actions must match. A Create only matches a Create, and a Move/Grow only matches a Move/Grow, etc. By coincidence, most of Gamut's actions have exactly two parameters. The exceptions are the Delete action and the Shuffle action which only have one parameter (which specifies the object to modify). The Create action's parameters are a prototype and a count, and all the other actions have an object to modify parameter and a value parameter that tells what to store in the object. In the Change Property action, the property to be modified is considered part of its type (and not a parameter) since it is unlikely that multiple properties in a single object can be set to the same type of value. Because there are at most two parameters, there are only three degrees to which actions may match. Two actions are considered *identical* if all of their parameters match. If all but one parameter matches, the actions are considered *similar.* If neither parameter matches, the actions are classified as *no match*.

6.4.1.2 Matching Invoked Actions Versus Uninvoked Actions

Matching occurs in two passes. First, the actions in the revised behavior are divided into two groups: one group contains the actions that were executed when the behavior was invoked, and the other group contains the remaining actions. Gamut determines the invoked actions using history trace from the behavior's event (see Figure 6.6). The trace's actions have backpointers that show from which action the trace was copied. (The backpointers are not shown in Figure 6.6 to avoid clutter.) Gamut follows these backpointers back to actions in the behavior. The trace actions' backpointers can only lead back to the revised behavior.

Gamut matches the example trace against the invoked actions first and then the remaining, uninvoked actions. Matching invoked actions first gives priority to those actions. This can help reduce the amount of revision the system must do. Some of the invoked actions will occur in branches of Choice descriptions. Since the actions were executed, the Choice objects' decision tree expression would already select that branch. If the system were to decide that another branch was preferable, the Choice's decision tree expression would have to be revised. Thus, selecting executed actions tends to reduce the amount of revision that the system must perform on action-level Choice descriptions.

6.4.1.3 Resolving Multiple Match Conflicts

A single example trace action may match multiple actions in the behavior. Gamut gives first priority to identical matches over similar matches. Since identical matches essentially drop out of the inferencing process, it pays to maximize the number of identical matches. The next priority is given to the value parameter over the object parameter. If a trace action matches one behavior action in its value parameter and another in its object parameter, then the value match wins. Through experimentation it was found that expressions in value parameters tend to be more complicated than those found in object parameters. As a result, it is better to maintain descriptions in the value parameters. Matching the value parameter first keeps that parameter fixed while Gamut finds a way to revise the object parameter. This heuristic has been found to work well in the applications tested with Gamut.

6.4.2 Matching Complications

Simply matching a trace action's parameters to the parameters in the behavior's actions is not quite sufficient because the structure of their actions is not the same. Recall that in stage one, trace actions that operate on multiple objects are broken apart in order to eliminate redundant actions more easily. The actions in the behavior are not similarly split apart and the trace actions have to be reunited in order to match against potential behavior actions. Also, Create actions have a slightly altered policy for matching because Create actions in the behavior might not have been invoked (they would exist in an uninvoked branch of a Choice) or the developer may have created new objects that the Create action has to match.

6.4.2.1 Matching Actions With Sets of Objects

The first complication concerns actions that operate on sets of objects. In this case, trace actions that act on single objects must be paired with original actions without attempting to determine similarity until all actions are paired. Gamut ranks whether actions are similar or identical after all the trace actions have been matched. If the objects affected by the trace actions are the same as the objects in the behavior action, the actions are considered identical; otherwise, they are similar.

A trace action may be paired to a behavior action when either the trace action's object matches a member of the behavior action's set or when the trace action's value matches the corresponding value of the behavior action. However, if one trace action matches a behavior action based on its value then all other trace actions assigned to that action must also match values. Trace actions with the same object but different values are allowed to match other behavior actions but are not allowed to mix with trace actions that match with their values. This keeps the system from trying to revise both of the behavior action's parameters at once.

When multiple actions match an action's object parameter but not the value, the system will only use the value of one of the trace actions to revise the behavior action's value. This might leave the values of the other trace actions undescribed. It is assumed if such a discrepancy should occur that the developer would fix the problem with more examples. This was never tested in practice.

6.4.2.2 Matching Created Objects

A second complication concerns matching Create Object actions. If the developer demonstrates creating graphical objects, those objects will not be the same as objects created by a behavior's Create Object actions. For instance, the developer might delete the objects created by the earlier behavior and supplant them with new objects. Also, the Create Object actions in the behavior might not have been invoked if they reside in conditional structures. The system has to match created objects with the Create Object actions regardless of the source of the created objects.

Instead of matching the parameters of Create Object actions, the system examines the properties of the created graphical objects to see if they "look the same" as the prototype objects of the behavior's Create Object actions. When Create actions are reduced in stage one, any properties of the created object that are set by subsequent actions are noted (see Section 6.3.5). So, when a Create action in the trace is compared against a Create action in the behavior, the prototypes are considered identical if the remaining properties of each object match. Only properties that are not set by other actions (and are derived solely from the prototype) are tested. If a match is found, Gamut will union together the sets of properties from each prototype and revise the behavior's Create Object action to contain a superset of all.

When the developer deletes created objects and replaces them with substitutes, the actions in the behavior that affect created objects will still refer to the original objects. When Gamut pairs a new Create action with one from the behavior, it also defines a pairing between objects that were actually created and objects that were supposed to be created but may have been deleted. Gamut uses this pairing to make a table and to substitute the objects that other actions use. When two Create actions are matched, the objects created by each action are stored in the table. To test the object parameter of other actions, Gamut first exchanges the action's objects with objects from the table.

In order to build the table of paired created objects, Gamut processes all Create actions before any other actions. Gamut uses a two-pass system to first process all the Create actions that builds the object mapping table before it processes any of the other actions. Note that this two-pass process occurs during revision and not execution. In Section 6.2.2, it was mentioned that Create actions do not really need to be *invoked* before other actions because of anticipatory descriptions. However, Create actions do need to be *matched* before other actions.

6.4.3 Assembling Actions and Creating Choice Descriptions

After matching the example trace with the actions from the behavior, Gamut is ready to begin rearranging and augmenting the behavior. In order to determine which code should be modified, Gamut must determine which branches contain actions that were invoked or should have been invoked. Gamut determines the branches by making a copy of the behavior and stripping out all the actions that were similar or identical to actions in the example trace. In general, the branches with the highest number of matches are considered to be the ones invoked, but there are some complications that are mentioned below. When Gamut selects branches, the remaining uninvoked branches might also contain actions that matched the example trace. These are called the *lost actions* because they represent invoked actions that exist in uninvoked branches. After the copy of the behavior is stripped, the remaining actions are those that do not appear in the example trace but exist in executed branches. These are enclosed in new Choice descriptions to convert them into conditional statements.

Though Gamut uses Choice descriptions as if-then or case statements, Gamut lacks an explicit “loop” structure such as the common for-next or while loops found in most programming languages. Some of Gamut’s descriptions do contain explicit loops. For instance, the Create action has a parameter for the number of objects to create, and the Chain description has a parameter for the number of links to follow. Gamut’s descriptions can also contain implicit loops since a single action can act on a set of objects, not just one at a time. Gamut also supplies a timer widget (see Section 4.3.3) that can invoke a behavior multiple times. These various looping devices are adequate to support repeated behavior, but it might still be useful if Gamut eventually supported a general purpose loop structure.

6.4.3.1 Stripping Actions to Determine Executed Branches

In the stripping phase of stage two (lines 14-19 in Figure 6.10), the behavior actions that were paired with example actions in the previous step are stripped out of a copy of the revised behavior. The matching phase of the algorithm produces two lists as its result: the list of identical actions in the behavior and the list of similar actions in the behavior. When stripping the behavior, similar and identical matches are treated the same way. The stripping algorithm is a recursive procedure (see Figure 6.12) that walks down the structure of the behavior and removes the similar and identical actions. What remains in the copy of the behavior after stripping are actions and Choice descriptions that were not evident in the example trace. For instance in the behavior shown in Figure 6.11, action A1 matched an action in the example trace. The match could be either identical or similar. However, A2 and A3 were not matched with the trace actions. When Gamut strips the behavior, it removes A1 but leaves A2 and A3 behind.

A Choice description is stripped whenever a trace action matches at least one action in one of its branches. If no trace action matches any action in the Choice, then the Choice remains in the copy of the behavior. For instance in Figure 6.11, Choice3 has no matching actions so it is not removed. When a Choice is stripped, it gets replaced by the branch that has the most matching actions. Choice1 in Figure 6.11 has two matching actions in its first branch and one match in the other, so Gamut selects the first branch. When Gamut strips Choice1, its first branch is inserted into the position of the Choice description and all the matching actions are removed. The result is that only the unmatched action in the selected branch, A6, remains in the code. Gamut counts all the matched actions in each branch including actions that occur within subordinate Choice descriptions. Ties are resolved by picking the branch that the Choice’s decision tree expression would choose or else the algorithm will pick the first one. Since stripping a Choice eliminates the actions from its other branches, any matching actions from those branches become “lost.” In Figure 6.11, A8 was in the unselected branch of Choice1. When Choice1 was stripped, A8 became a lost action.

Stripping a Choice description determines which of its branches was invoked. When a Choice becomes stripped, Gamut records the selected branch. The list of selected branches becomes part of the set of “changes that must be satisfied” or the “Changes” set. The Changes set factors prominently in the next part of stage two where values are propagated to descriptions and will be described in more detail later.

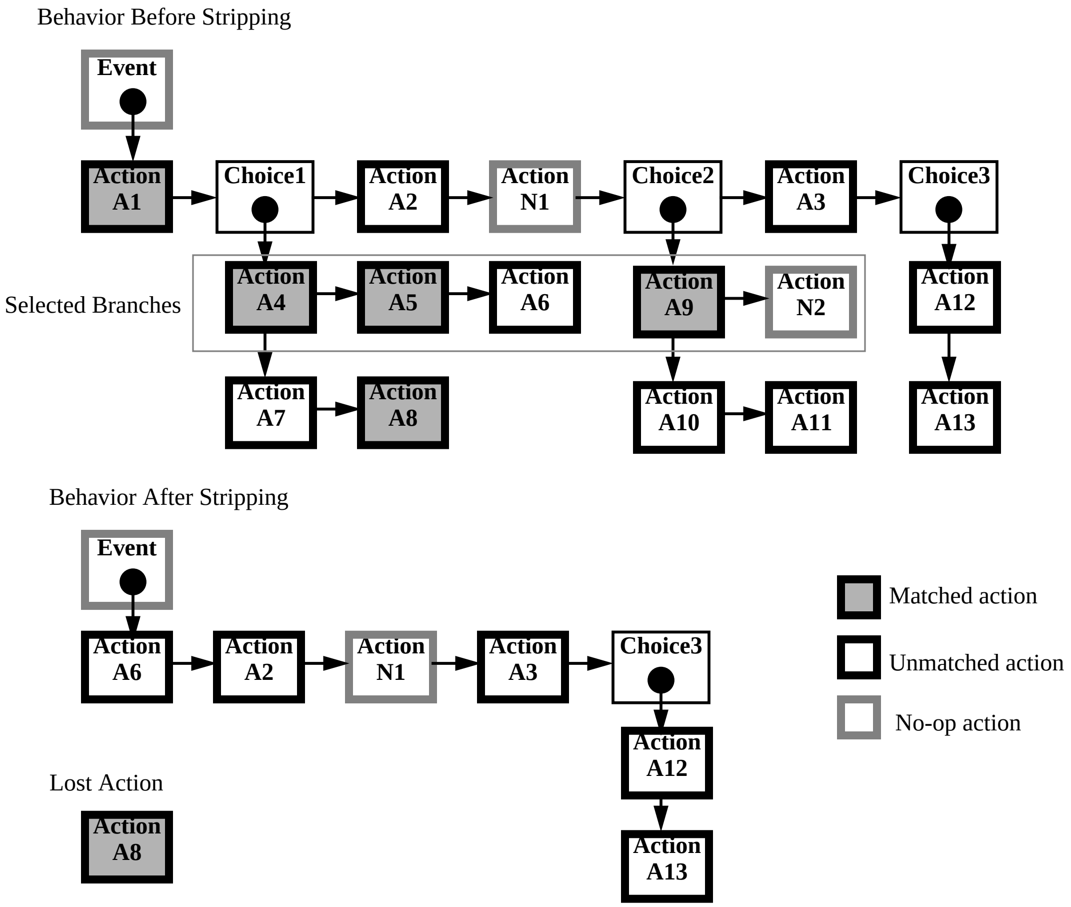


Figure 6.11: To determine unmatched and lost actions, Gamut strips away all the actions from the behavior that match actions in the example trace. During this phase Gamut determines which branch of Choice descriptions were invoked. Matched actions are shaded and no-op actions are marked with a special border.

6.4.3.2 No-Op Actions

Sometimes the value produced by an action will be identical to the value that the modified object had before. In Gamut, an action that exhibits this phenomena is called a “no-op” action because even though it runs, it appears to have no effect. An action can become a no-op in two ways. First, the expression in the action’s value parameter might simply return the identical value. For instance, consider implementing a Reset button for a game like Chess. Pressing the button causes all the pieces to move to their starting position. Since some of the objects may already be in their starting position, the action that moves those objects will sometimes do nothing. The other means through which an action can become a no-op are error situations where one or more of the actions parameters cannot be evaluated. For instance, say an object is supposed to move to the center of the rectangle overlapping a particular arrow line as in Figure 6.13. When the arrow line does not overlap a rectangle, Gamut handles the error by not executing the action.

Testing an action to see if it is a no-op is relatively straightforward. If the action is no-op because of an error, one or more of the parameters will be invalid. Similarly, the history of an affected object’s property can be tested to see if the action has set the property to the identical value.

```
int Strip_Behavior (behavior, matching_actions)
  int num_matches = 0;
  if behavior is a Choice then
    choice = behavior
    int max_branch = current_branch
    num_matches =
      Strip_Behavior (choice[current_branch],
                      matching_actions)
    for branch = 1 to num_branches do
      int matches =
        Strip_Behavior (choice[branch],
                        matching_actions)
      if matches > num_matches then
        num_matches = matches
        max_branch = branch
      end if
    end for
    if num_matches > 0 then
      replace behavior with choice[max_branch]
      note in choice which branch was chosen
    end if
  else
    list of actions = behavior
    foreach action in list of actions do
      if action is a Choice then
        num_matches +=
          Strip_Behavior (action,
                          matching_actions)
      else if action is member of
               matching_actions then
        ++num_matches
        remove action from list of actions
      end if
    end for
    replace behavior with list of actions
  end if
  return num_matches
end function
```

Figure 6.12: Pseudo-code for the stripping algorithm which recursively scans the blocks of the code and removes the identical and similar actions. The variable `matching_actions` is a list of both similar and identical actions. The `current_branch` variable is the branch to which the Choice object's decision tree expression evaluates. Branches of the Choice object are being indexed as though they were an array.

The main question concerning a no-op action is whether or not the action should be considered to have occurred or not. If it did occur, it should be stripped out of the behavior like matched actions. Since a no-op action does not affect the state of the application, there will be no record of it in the action trace so there will be nothing in the trace to match against. As shown above, sometimes proper behavior includes times when an action will be a no-op in which case, the action should be considered to have occurred even though it does not appear in the trace.

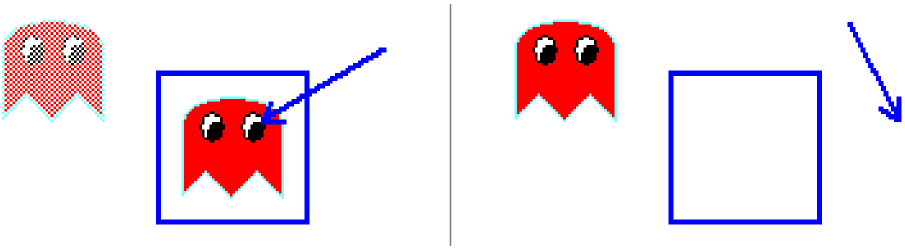


Figure 6.13: A behavior is defined so that the character moves to the center of the square that the arrow line points to. When the arrow line does not point to a square, the description for moving the character becomes invalid and the character does not move.

Gamut's rule is that if the set of unmatched actions are all no-op, then those actions will be considered to have occurred. In Figure 6.11, N2 is a no-op action in the selected branch of Choice2. Since it is the only unmatched action, it is assumed to have occured so it is removed along with A9 when the Choice description is stripped. When all the unmatched actions are no-op, there is no need to add a condition to prevent them from happening. However, if one of the actions in the unmatched set is not a no-op, then Gamut will have to create a Choice description anyway. In Figure 6.11, the block containing N1 also contains other unmatched actions, A2 and A3. Thus, N1 is not removed and it will be placed in the same Choice branch as A2 and A3.

One exception to this rule concerns Choice descriptions that are no-op. A Choice can be a no-op if any one of its branches is empty or contains only no-op actions. A no-op Choice is always stripped even if one of the actions in the block level containing the Choice description is not a no-op.

6.4.3.3 Inserting New Conditions and Actions

The same style of recursive procedure that is used to strip actions from a behavior is also used to insert conditions and new actions into a behavior. Figure 6.15 shows pseudo-code for the algorithm that takes the remaining actions in the stripped behavior and trace and uses them to revise the original behavior. The basic procedure is to take all actions found in the stripped behavior and enclose them within a condition and to take the actions found in the stripped trace and append them to the end of the block. However, Gamut can detect a couple of situations that help to make the code cleaner.

When multiple actions in the same block level have all been left unmatched, Gamut does not create a condition for each action, but instead combines the set of actions into a new block and stores them within a single condition. For instance in Figure 6.11, A2, N1, A3, and Choice3 can all be placed in a single new condition as shown in Figure 6.14. A6 must be placed in a different Choice description because it came from a different block. Gamut assumes that all unmatched actions in the same block have been switched off for the same reason and can safely be placed in a single condition. Another simple case occurs when a condition is the only unmatched action in a block. Instead of embedding the existing Choice description inside a new one, Gamut adds a new branch to the existing Choice. When Gamut adds a branch to a condition, that condition is recorded in the Changes set.

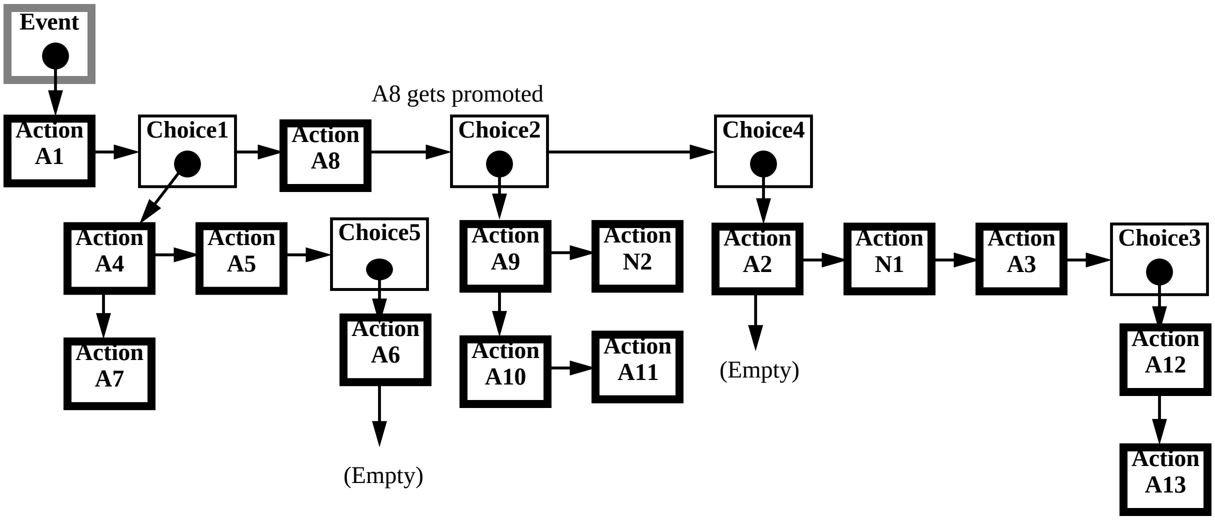


Figure 6.14: Gamut adds new Choice descriptions (Choice4 and Choice5) to hold the unmatched actions from Figure 6.11.

The remaining actions in the example trace become new actions in the behavior which Gamut must insert somewhere in the code. Gamut tries to insert new code in the most specific branch of a Choice description that it can find. When a block contains no Choice descriptions, Gamut appends the actions to the end of the block. When a block contains one Choice description, the actions are inserted into the executed branch of that condition (applied recursively). If there are two or more conditions in the block then Gamut cannot know in which one to place the new actions so it just appends the actions to that block. Gamut assumes that new actions represent a new feature to the existing behavior and does not enclose the actions within a Choice description. If it turns out that the new actions are conditional, then later examples will show when they are not supposed to be executed and Gamut can create a Choice description then. In the example from Figure 6.14, there is more than one Choice description in the top-level block so new actions are appended to the end.

6.4.3.4 Promoting Lost Actions

When Gamut selects the invoked branches of each Choice description, any matched actions in the unused branches become *lost actions*. Lost actions are actions that are considered to have been invoked but do not exist in an invoked branch of the behavior. Gamut interprets lost actions as though they were placed at the wrong level block. Since Gamut places new actions at the lowest unique block level it can find, it is possible that some actions are placed too low. So, Gamut “promotes” lost actions by moving them in the Choice description hierarchy and placing them in the next lowest block that was executed. Figure 6.16 shows an example of how lost actions are promoted.

6.4.3.5 Restoring Behaviors Invoked Within Response Mode

As mentioned in Section 6.3.4, when the developer invokes a behavior while demonstrating an example for another behavior, the sub-behavior’s actions are incorporated directly into the revised behavior. Gamut acts as though the sub-behavior’s actions were manually generated by the devel-

```
Assemble_Behavior (behavior, stripped_behavior,
                   stripped_trace)
  var
    list of choices
    list of unmatched_actions
    list of result_actions
  foreach action in behavior do
    if action is a member of
             stripped_behavior then
      add action to list of unmatched_actions
    else
      add action to list of result_actions
      if action is a Choice then
        add action to list of choices
    end if
  end for
  if length of list of unmatched_actions != 0 then
    var choice
    if length of list of unmatched_actions == 1 and
       unmatched_actions[0] is a Choice then
      // reuse the existing choice
      choice = unmatched_actions[0]
    else
      choice = create a new Choice description
      choice[0] = list of unmatched_actions
    end if
    if length of list of choices == 0 then
      add_branch (choice, stripped_trace)
      stripped_trace = empty list
    else if branch of choice is already empty then
      redirect choice to use empty branch
    else
      add_branch (choice, empty list)
    end if
    add choice to result_actions
  end if
  if length of list of choices != 1 then
    append stripped_trace to result_actions
    stripped_trace = empty list
  end if
  foreach choice in list of choices
    branch = executed branch of choice
    Assemble_Behavior (choice[branch], stripped_trace)
  end if
  replace behavior with result_actions
end procedure
```

Figure 6.15: Pseudo-code for algorithm that adds new conditions into the code. The actions from the stripped behavior are compared to actions in a block of the revised behavior. The actions are grouped into several sets. Unmatched actions are either placed into a new Choice descriptions or added to an existing Choice. Actions from the stripped example trace are placed into the lowest unique block that was executed.

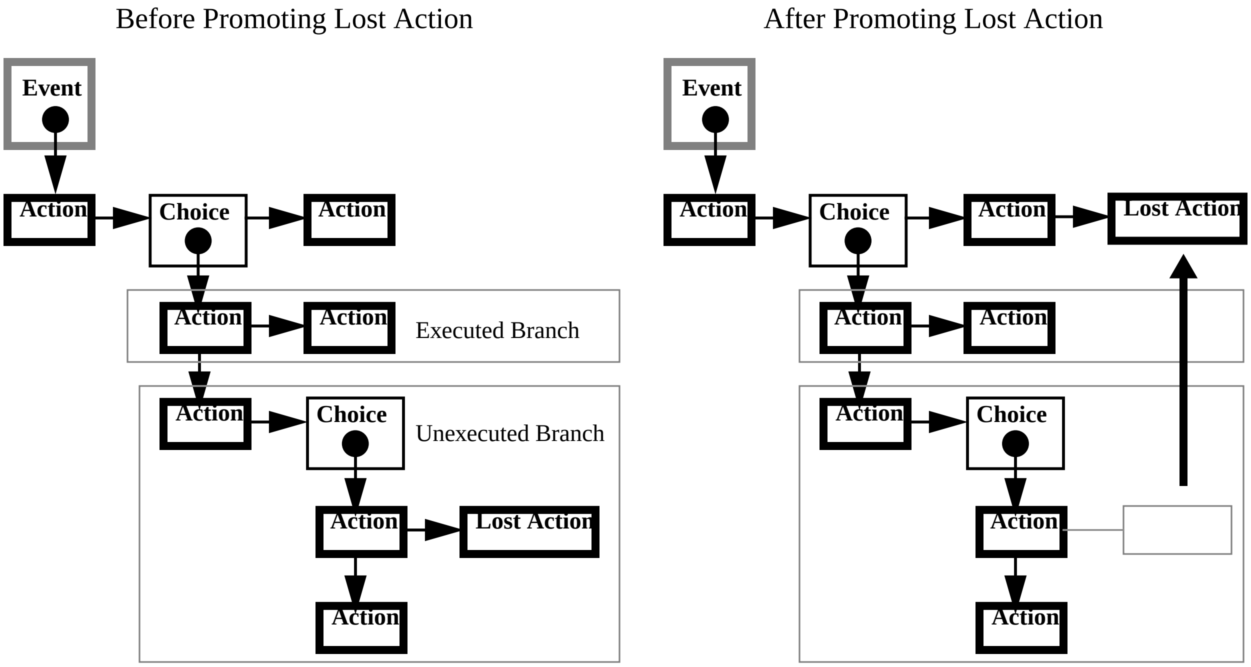


Figure 6.16: In this example, the lost action exists in a Choice that is also stored in the branch of another Choice. The higher-level Choice is the last one to exist in executed code, so the lost action is promoted to that Choice's block.

oper and allows the actions to pass through stage one and most of stage two normally. However, it is probably true that if the developer does not modify the results of the sub-behavior while demonstrating the new example, that the developer probably wants the new behavior to take on all the characteristics of the sub-behavior and not have to relearn the sub-behavior's descriptions. Therefore, Gamut restores the conditions and descriptions found in the sub-behavior.

Gamut's rule is that a sub-behavior can be restored only if it has not been subsequently affected by the developer's example. In other words, if the sub-behavior affects an object, that object must still be in the same state the sub-behavior placed it in at the end of the example. To do this, Gamut takes the list of new actions that it just installed into the revised behavior and determines which ones were derived from the sub-behavior. All invoked actions contain backpointers to their prototypes. Gamut follows these backpointers to see how many end in the sub-behavior.

To restore an action, Gamut replaces actions from the example trace with their corresponding action from the sub-behavior. If the sub-behavior's action is enclosed within a Choice description, then Gamut will not restore the action unless all the other actions in that Choice description's branch also match. In this case, Gamut replaces the actions with the whole Choice description. Since Gamut does not try to understand the contents of the sub-behavior, it tries to keep as much of the code together that it can. Gamut cannot determine what effects splitting up a Choice description would have on the code so only the top-level block of the sub-behavior is allowed to be split.

A major caveat for restoring sub-behaviors is whether or not to incorporate no-op actions from the sub-behavior. Since these actions have no affect on the state of the application, they would not appear in the list of new actions. And since they do not affect the application, the developer has no way to tell the system if a particular no-op action should not be copied. Gamut's rule is that if all normal actions in the top-level of the sub-behavior match in the new behavior then the top-level no-op actions are added to the new behavior as well. However, if any of the top-level actions in the sub-behavior do not match, then no-op actions are not added. This is a fairly arbitrary decision. To date, few developers have actually used sub-behaviors in their demonstrations so there has been no evidence on which to base a decision.

6.4.4 Propagating Changes To Descriptions

At the end of the matching step, Gamut has several pieces of data: the newly revised behavior, the list of new actions, the list of similar and identical actions, and the list of modified Choice descriptions. At this point, the actions of the revised behavior are in place and they can be stored in the behavior's event. However, the parameters of the similar actions are not the same as the parameters of the trace actions, and the decision tree expressions of the revised and newly created Choice descriptions still need to be updated. Determining the ways in which the parameters of the similar actions can be made to match the trace is discussed in this section. This data along with the changes to the Choice descriptions is then passed to stage three which will assign descriptions to various parameters and revise decision tree expressions.

Gamut's method for propagating new values into the descriptions begins with the list of similar actions. A similar action is an action from the behavior that matches all but one parameter of an action in the trace. Gamut must determine how to revise the mismatched parameter from the original value to the trace value. To do this, Gamut propagates both the original and new value to the "difference" method (see Section 6.1.3.1) of the description stored in that parameter of the behavior's action. The difference method takes the new value and finds a way to modify the description to generate that value.

Each type of description has its own custom difference method, but most simply apply a technique in Gamut called the "one change" rule. Using the one change rule, an algorithm only tries to change one thing. This is similar to Winston's concept of a "near miss" for training examples [95]. The difference method tries changing one of the description's parameters at a time leaving the others the same. For instance, the Connect description finds an object in the application that is connected to a reference object in a certain way. It has two parameters: the reference object and the way that the objects would connect. Using the "one change" rule, the difference method can choose either to change the means of connection or the reference object. The method will first search the window for different objects that happen to be connected in the needed way. Also, it will compare the reference object to the new object that it is supposed to return. If the two form a connection that is a subset of the previous connection, the method will record that option as well. All matches that the method finds are added to the *Changes* set (see below). The new values are then recursively propagated to the Connect description's parameters. In this case, only the reference object parameter can hold a description so the discovered objects are passed to it. Other difference methods works similarly. For instance, a Get Property description will search for objects that exhibit the same property value as its original object. An Add description will search for objects with the desired numerical properties.

Propagating new values through the original behavior is common in AI techniques such as reinforcement learning [86] and neural networks [63] but has not been previously applied to a PBD tool. The majority of PBD systems only attempt to generate new expressions from scratch similar to the processes that Gamut uses in stage three. Even PBD systems that learn from multiple examples have required the developer to provide all examples at once and do not allow the developer to revisit previously inferred behavior and make further revisions (see Section 2.1.1.1).

6.4.4.1 The Changes Set Data Structure

Potential parameter modifications are stored in the “changes that must be satisfied” data structure, or the “Changes set” for short. The overall format of the set is an And-Or tree. And-nodes indicate a set of changes that must occur together (or not at all), and Or-nodes indicate that any one of a set of changes will satisfy the parameter. The top node of the set is an And-node that connects nodes that resolve the parameter differences for each pair of similar actions and nodes that resolve the modified Choice descriptions. The top node is always an And-node because Gamut must resolve all the parameter differences and Choice descriptions in order to revise the behavior.

A leaf node of the Changes set represents a potential change that could be made to a given description or action. Gamut provides four kinds of leaf nodes called the Change, Replace, Branch, and Select leaf nodes. The Change leaf is the most common. It represents a change made to the parameter of an action or description and consists of an old value and a new value. Its semantics are that if the system finds a way to describe the new value, it can use that description to replace the old value and satisfy the parameter. The Replace leaf is similar to the Change leaf except that it represents a change to a constant parameter. Some descriptions have parameters that cannot be filled with a non-constant description. For instance, the property value of a Get Property description is always constant since Gamut does not contain a description that returns a property. Since the parameter cannot be described, Gamut is only allowed to replace the parameter with a new value. Normally, a Replace leaf is accompanied by other Change leaf nodes within an And node so the system does not simply replace all the original values without consideration.

The Branch and Select leaf nodes have a special purpose in Gamut. These nodes represent changes to a decision tree database. As discussed below in Section 6.6, Gamut uses decision trees to represent conditional structures like the ones found in a Choice or Select Object description. The Branch leaf represents changes to a Choice description, and the Select leaf represents changes in a Select Object description. The Branch leaf’s data consists of the indices of the old and new branches. Since multiple branches might hold the same value as the one passed into a Choice’s difference method, the new branch value is actually a list of branches. The Select leaf holds three values, the old set of objects that the Select Object used to return, the new set of objects it was supposed to return, and the set of objects that the object description inside the Select Object returns. The way that these values are used are described in Section 6.6.6.

6.4.4.2 Forbidden Objects and Properties

Gamut’s descriptions must avoid applying the same parameter twice in a single chain of expressions. For instance, to describe the arrow line labelled ‘A’ in Figure 6.17, the system might pick arrow B and describe arrow A’s start point as connected to the end point of arrow B. When describing arrow B, the system must be prevented from describing B by saying that it is the object whose end point connects to the starting point of A. If this were to happen, A would end up describing itself and not produce any useful information. In fact, all descriptions that are used in the parameters of a description that describes object A, no matter how deep they are in the hierarchy, must be prevented from reusing object A as part of their description. When an object like

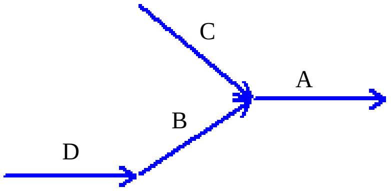


Figure 6.17: Arrow lines A, B, C, and D are connected as a fork. If B is used to describe A, then B must be prevented from using A to describe itself. Furthermore, B cannot use C to describe itself, either, because C is connected to B using the same point that B is being used to describe A. However, it is okay to describe B using arrow D.

arrow A must be prevented from being reused, it is called a *forbidden object*. As the chain of descriptions grow deeper, more objects become forbidden.

A more complicated case of forbidden reuse occurs by adding a third arrow line, C. Once again, Gamut has described arrow A as the object whose start point connects to the end point of arrow B and now wants to describe object B. The system cannot choose to describe object B as the object whose end point connects to the end point of arrow C. The reason is that it is the end point of B that is being used to describe A. To redescribe the end point of B is to describe the same thing that is used to describe A making the intermediate description useless. Hence, the property that the current description uses to provide its value to its parent description must be remembered in order to prevent a parameter from describing the same property. This is called a *forbidden property*. Note that the whole location of B is not prevented from describing B. Object D, which is connected to the starting point of B, is a perfectly valid way to describe B. Hence, the system actually records location properties in terms of their parts. The method Gamut uses to describe portions of a location is described in Section 6.7.2 about geometry and alignments. Other properties such as color are not as complicated and can be represented just by their object property name.

6.4.4.3 Other Caveats During Propagation

The “one change” rule usually provides a sufficient criteria for creating a useful difference method. A few descriptions such as Align (which computes the location of graphical objects) use more complicated heuristics (see Section 6.7). There are a number of decisions that a description designer faces as well as a number of pitfalls that must be avoided.

Most difference methods do not require input from the developer. Difference methods generally do not use hinted objects because they are sufficiently constrained by the “one change” rule not to produce too many results. Furthermore, eliciting special information from the developer requires that the system generate dialogs and interaction techniques that can ask for that kind of information. Therefore, most difference methods quietly search the application for pertinent objects without querying the developer. An exception to this would be descriptions like Count Objects which returns the number of graphical objects in a set. The Count Objects description cannot be computed in reverse since knowing that the description is supposed to return 5 instead of 3 does not tell the description which 5 objects to count. Instead, it would have to ask the developer to highlight the objects or use some other elicitation to find the objects the developer wants to specify. The Count Objects description needs to ask questions because its operation throws away most of the data that it uses to compute its value. Currently, Count Objects has not been implemented because it has not been needed for any of the test applications or the usability experiment. As a result, the needed feedback mechanisms have not been explored.

It is important that difference methods do not select modified objects or properties to describe new values. Consider an Align description that describes the location of the rectangle shown in Figure 6.18. As the ghost object in the figure shows, the arrow line that attaches to the center of the rectangle was moved as well. If the position of the rectangle depends on the final location of the arrow, Gamut must create an "anticipatory" description (see Section 6.1.3.6) to avoid an ordering dependency. However, if the arrow's position is undescribed, it might be that its location depends on the rectangle. If both the Align description for the rectangle and the Align for the arrow both "anticipate" that they need to use each other's location, Gamut would have no way to resolve this issue in stage three and would be deadlocked. The rule therefore, is not to use modified properties during value propagation. While Gamut is propagating values, it is only allowed to use values that have not been changed in the example trace. Algorithms in stage three are used to resolve situations where anticipated values are needed.

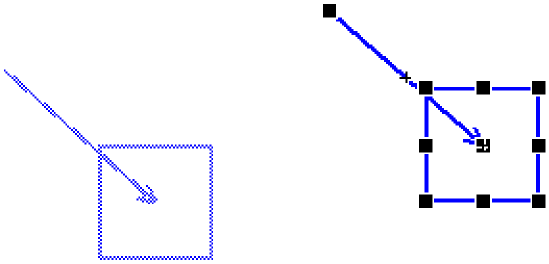


Figure 6.18: Both the rectangle and arrow line are moved simultaneously. Since the system does not know whether the rectangle's position depends on the arrow or vice versa, Gamut must wait until stage three to make a decision.

6.4.4.4 Tracking the Revision Context

Gamut uses a single data structure called the Context object to maintain state information while parameter values are propagated to successive descriptions. The Context holds several pieces of information. It holds the list of forbidden objects and properties that was described above. It also holds the history of the revised behavior, as well as the position where the description is stored in the behavior's code.

Gamut stores the behavior's history as a "constraint context," which is an Amulet formalism that allows Get and Set calls on the properties of Amulet objects to be redirected to custom methods. Normally, this feature is used by Amulet's constraint system, but Gamut uses it to make Get calls on an object return the historical value of the property when desired. The history is important to a difference method since the method must usually search the original state of all the graphical objects to find the proper dependencies. Furthermore, a difference method uses the historical values whenever it evaluates a subordinate description in one of its parameters.

Gamut must also store backpointers to the description and parameter that contains the description and difference method that are currently being evaluated. Gamut needs to record this data in the Context because parameters of different actions and descriptions are allowed to share the same description. Hence, descriptions do not contain backpointers of their own and do not know the context in which they will be applied. If a description is used multiple times in the same behavior, its difference method can also be called multiple times: once for each parameter that refers to it. When the difference method creates a leaf for the changes set, it stores the Context's backpointer in the leaf. This makes the code reference for each difference method call unique even when the actual description object is not.

6.5 Stage Three: Resolving Differences

Stage three is the last part of Gamut's inferencing algorithm. Its job is to make the parameters of the behavior's actions match the values in the example trace. Stage three resolves the parameter differences that were found by stage two, manages changes in existing Choice and Select descriptions, generates expressions for new Choice and Select descriptions, and describes the parameters of newly created actions and descriptions. Stage three assembles this data into two structures: the Changes set which holds changes to parameters and manages expressions for Choice and Select, and the new parameters queue which holds the parameters of new code objects.

Highlighted graphical objects factor heavily in stage three. Stage three uses the highlighted objects to create new descriptions. When the developer does not highlight objects that produce descriptions for items in the Changes set, the system must ask questions that will hopefully prompt the developer to highlight the needed objects.

6.5.1 Sorting The Changes Set

The modifications in the system's Changes set could hypothetically be resolved in any order. However, when there are multiple ways to resolve a set of changes (when there exists an Or-node with more than one branch), Gamut applies a sorting criteria in order to choose which change is more likely to be the one the developer expects. This helps prevent Gamut from generating unusual questions for the query dialog (see Section 6.5.5).

Gamut uses the revision context to pick which leaf of the Changes set was generated by the deepest description. As discussed in the previous chapter, Gamut assumes that code that matches deeper in the description hierarchy is more likely to be what the developer wanted to change. Gamut counts the chain of descriptions starting from the context's action and ending at the parameter that is being changed.

When Gamut determines the depth of Select Object and Choice leaf nodes, it adds a bonus of one. Through experimentation, it was determined that Select Object leaves seem to be most likely nodes that the developer wants the system to update. Similarly, in a Choice description, it was more likely that the Choice's decision tree expression needed to be updated rather than having the system add a whole new description to the Choice's set.

After Gamut sorts the Changes set, the system attempts to resolve them in order. Resolving a leaf node is not guaranteed. The objects that the developer highlights may not be useful for that leaf node or it might generate a redundant description. Whether or not a change succeeds is propagated to the node to which the leaf is attached. When a branch of an Or-node succeeds, Gamut will use that branch to resolve the node. When an Or-node branch fails, Gamut will try the next branch. When a branch of an And-node succeeds, Gamut will continue resolving the other branches. If an And-node branch fails, the other branches are not attempted and the entire node fails.

6.5.2 Describing a Parameter

Gamut uses three methods for revising a parameter from one value to another. Gamut can search for an existing description that evaluates to the new value, it can create a new description that evaluates to the new value, or Gamut can store the old and new values for the parameter into a Choice description and use decision tree learning.

6.5.2.1 Searching for Existing Descriptions

Searching the behavior for an existing description is relatively straightforward. Gamut scans the actions and descriptions in the revised behavior for descriptions that return exactly the same value as the one desired. To reduce the search space, Gamut only looks at descriptions in executed branches of Choice descriptions. Other branches are ignored because they usually contain “dead code” (see Section 6.5.6) that hold descriptions that are not being used. When a matching description is found, it is put through a more rigorous test to make sure it can be used in the new context. In order for an existing description to be integrated into a new context, the description must satisfy two criteria: the description cannot contain undescribed parameters, and it cannot incorporate forbidden objects or properties.

A description’s parameters are considered “undescribed” when the description is newly created. Undescribed parameters contain constant values where they might eventually contain descriptions. For instance, a description is not allowed to refer to a newly created graphical object as a constant. Since created objects do not exist in the previous state of the application, referring to one as a constant value is meaningless. Since undescribed parameters might be expanded later, a incomplete description can return a different value from the one it would return if its parameters were fully expanded. As a result, new descriptions are considered off-limits to the search process.

Forbidden objects and parameters were mentioned in Section 6.4.4.2. When a difference method scans objects in the application for ways to revise its description, certain objects are not permitted to be used. If a description that is found through search uses one of these objects, it is also forbidden for the same reasons.

6.5.2.2 Creating a New Description

Gamut can also create a new description to revise a changed parameter. Gamut uses a heuristic search algorithm similar to the one in Marquise [76] to accomplish this task. However, instead of using only heuristics to find matching descriptions, Gamut focuses attention on the highlighted objects.

Gamut divides the task of creating new descriptions based on the type of the value desired. Gamut currently defines five types: geometric locations, numbers, objects, sets of objects, and other properties. In theory, Gamut could use a mechanism like the difference methods to assign description generating heuristics to each description, but for historical reasons, Gamut uses monolithic procedures to generate descriptions for each type.

Generating a new description is usually straightforward. Each potential description is examined in turn. For each description, the list of highlighted objects is scanned and tested. If a highlighted object contains properties that a description can use to generate the needed value, Gamut will create that description and use the highlighted object as one of its parameters. The description’s other parameters are filled to reflect the relationship between the property of the highlighted object and the needed value.

Normally, only one highlighted object is needed to create a description. For example, in Figure 6.19 a highlighted arrow line points to the center of a rectangle. One of the descriptions for describing objects is the Connect description. When the Connect description is tested, it scans the list of highlighted objects and finds the arrow and discovers that the end point of the arrow meets the center of the rectangle. This is enough evidence to produce a Connect description. The procedure creates new Connect description and stores the arrow line object in the reference object parameter. The fact that the connection happened between the end point and center is stored in the other parameters.

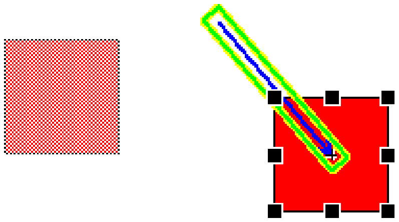


Figure 6.19: Here a highlighted arrow line points to the center of a rectangle. When Gamut describes the rectangle, it can use the arrow line by creating a Connect description.

Just as in the revision and search process, forbidden objects and properties must be avoided. Gamut first removes forbidden objects from the list of highlighted objects. Also, when a relationship is discovered, Gamut tests the property of the relationship to see that it is also not forbidden.

Once Gamut generates a new description, the description is stored in the parameter using the backpointers provided by the Changes set. The parameters of the new description are added to new parameters queue. Resolving a parameter difference from the changes set clears that entry from the set. If the entry was part of an Or-node, then the whole node can be eliminated, but if it is part of an And-node then Gamut will have to continue to try to resolve more changes.

6.5.2.2.1 Adding Select Object to Object Descriptions

Gamut currently uses two descriptions for generating sets of objects: Connect and Chain. Both of these descriptions use geometry to select objects from a window or card. The actual value that the application needs might rely on other properties of the objects besides geometry and might only want a single object at a time. To provide these services, Gamut adds new object descriptions into a Select Object description before it is placed in the behavior.

The Select Object description uses a decision tree expression to pick objects based on their various properties. The method that Gamut uses decision tree learning is discussed in Section 6.6. The decision tree expression can either be used to pick a single object from a set to return a single object or to remove undesired objects from a set and return a set of objects.

In several examples, such as in the previous chapter, object descriptions have appeared without showing the accompanying Select Object descriptions. The material was presented this way to eliminate the need to describe a potentially confusing concept where it was not needed. Also, since all object descriptions are embedded in a Select Object, the Select Object might as well be considered part of the object description. In examples where Gamut's code is displayed without showing Select Object descriptions, it can be assumed that a Select Object exists on any description returning an object.

6.5.2.2.2 Describing Sets of Objects

Gamut has a special procedure for describing sets of objects because sets generally require more evidence to describe than single objects. Currently, Gamut only supports a geometric description to describe sets of objects, specifically, the Connect description. A significant complication when describing a set of objects is that there is more opportunity for finding spurious relationships between the objects and highlighted objects.

To describe a set, Gamut first describes each object in the set individually. In Figure 6.20, suppose the system needs to describe a set of four arrow lines. The arrows might be used in another description or action. For instance, perhaps the color of these arrows might be all changed to blue.

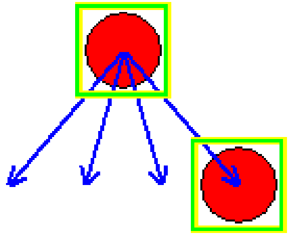


Figure 6.20: The system needs to describe the four arrow lines. The developer has highlighted both circles, only one of which could be used to describe all four lines. The system must cull the spurious relationship with the second circle.

To describe the arrows, the developer has highlighted the circle that joins all lines at their start point. The developer has also highlighted another circle that happens to be at the end of one the lines. Since the developer is allowed to highlight any object, it is possible for the system to find spurious relationships such as this. Gamut's algorithm for describing sets must be able to select among many potential descriptions.

Gamut records all connections between each desired object and highlighted object by creating a Connect description for each. Gamut then evaluates each Connect description to see which ones happen to produce the most objects in the desired set. In this case, the Connect description for the top circle will produce all four arrow lines and will be the best match. Gamut can also combine multiple Connect descriptions that use the same highlighted object by forming the intersection of the geometry properties that they use (see Section 6.7.4). This can help Gamut find a description that fits all the desired objects. Gamut can also join multiple Connect descriptions when more than one highlighted object is needed to describe the entire set. Objects that cannot be described are maintained as a constant list.

6.5.2.3 Creating a Choice Description

The final method that Gamut can use to resolve a parameter change is to create a Choice description. When Gamut is unable to find a description to generate a given value, it is assumed that the value is a constant and that other criteria are being used to determine which value to use. Gamut stores the old value that the parameter once held and the new parameter value as a list in a new Choice description. Gamut then uses the methods described in Section 6.6 to create a decision tree expression that will select which value to use.

6.5.3 Revising Decision Tree Expressions

Some portions of the revised behavior may already contain decision tree expressions in the form of Choice and Select Object descriptions. When Gamut revises these descriptions, it creates Choice and Select leaf nodes in the Changes set. Gamut resolves these items by adding new entries to the existing decision tree database in those description or by creating new attributes for the decision tree expression (see Section 6.6.6).

6.5.4 Automatically Highlighted Objects

Through the use of a paper prototype study (see Section 7.1) and some preliminary user testing, it was discovered that developers do not highlight certain kinds of objects. In some sense, such objects were considered “too obvious” to highlight. Mostly these objects were temporal ghosts (see Section 4.2.2.2) where the developer would highlight an object connecting the before and after state of another object that had been moved but would not highlight the ghost object. In order to generate a new description, Gamut needs the important objects highlighted. Otherwise, there would be no list from which to test relationships. To alleviate this problem, Gamut automatically highlights one class of object, the ghosts of objects referred to by the object parameter of an action.

Autohighlighting only applies to transformation actions like Move/Grow and Set Property that have an object parameter. Gamut will not automatically highlight anything for a Create Object or Delete action. Furthermore, automatically highlighted objects have a second-class status compared to objects the developer highlights. Autohighlights can only be used to infer some relationships, mostly geometric relationships. Relationships between colors, numeric values, and booleans are restricted to only using objects highlighted by the developer.

As an example of autohighlighting, in Figure 6.21, the developer has just moved a character to the right as the example and highlights the line between the character and the ghost of the character to say that the position of the arrow matters. Gamut will not only infer that the end of the arrow matches the center of the character, but it will also autohighlight the ghost of the character and note that the center of the ghost matches the start point of the arrow. Note that the regular character is not autohighlighted. In fact, the object that is modified in a transformation action is always forbidden and cannot be used to describe the value of the action. (Using the object to describe its own position would be circular.)

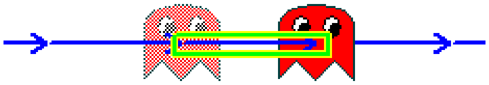


Figure 6.21: The developer moves a character to follow a chain of arrow lines. The developer highlights the arrow but does not highlight the ghost of the character. Gamut will highlight the ghost automatically and still discover the relationship.

In general, allowing Gamut to highlight objects automatically can be dangerous. For instance, what if in the previous example, the alignment of the ghost and arrow line was just coincidental? The developer would be forced to demonstrate a second example because Gamut inferred more than was required. Geometric relationships are restrictive enough so that accidentally highlighting a ghost object is not likely to generate spurious inferences. Other kinds of values like numbers are vulnerable to misinterpretation, though. Basically, one can add a constant to any integer to produce a given integer value. Thus, Gamut would only want properties of objects the developer highlighted to generate integer descriptions. This same issue holds for the attributes of decision trees (see Section 6.6.3) that are similarly unconstrained. Essentially, if Gamut uses automatically highlighted objects to generate descriptions for a highly unconstrained value, the system may spend a long time learning that it has made a mistake before asking the developer to highlight what Gamut should have used all along.

6.5.5 Generating a Question

When Gamut fails to find an appropriate description, it asks the developer a question. The question asks the developer to highlight objects in the application that Gamut could use to revise an item in the Changes set. It is hoped that the objects the developer highlights can be used to create a description for the revision. If not, then the highlighted objects are used to create attributes when Gamut creates a Choice description.

An item in the Changes set contains three pieces of information: an old value, a new value, and the context where the value is located in the behavior. From the context, Gamut distills two pieces of information: the description whose parameter is being modified and the top-most action that contains the description. Note that there might be several layers of descriptions between the top-most action and the description that holds the parameter. There also might be no description at all if the modified parameter is for an action. One more situation occurs when the context is a Choice description that acts at the action level of the behavior. In this case, there is not even an action to provide context, just an old and new branch value.

Gamut uses methods to generate the questions for the developer. Each kind of action and description contains a “dialog method” that when called returns a string message that Gamut puts into a dialog box. Gamut first calls the dialog method on the action of the context. (For the one case where there is no action, Gamut uses a special procedure.) The action’s dialog method looks at the description context and the old and new value and decides whether it must further delegate work to the dialog method on the description. Each action and description contains information about the kinds of values it uses. They use that information to read the old and new values and generate a text string pertinent to the expression that the action or description computes. Gamut does not generate text for the intermediate layers of descriptions because it would likely generate dialogs that were too wordy and potentially confusing.

When Gamut calls the dialog methods, it also sets up a sets of marks in the application window to point to objects referred to by the text. If the old and new values contain objects or locations, then Gamut draws markers around each of those positions. The dialog procedures assume that Gamut will draw the markers, thus they produce text accordingly. Gamut also marks the objects modified by the action part of the context. To do this, Gamut evaluates the object parameter of the action to recover the affected objects.

For example, consider the question the system asks the developer in the application from Chapter 5. The developer has already demonstrated that the piece object follows a chain of arrow lines starting from the piece's initial position, and has just given an example that shows that the length of the chain is equal to the value shown on the die in the center of the board. The currently inferred behavior is shown schematically in Figure 6.22a. When Gamut analyzed the example it found that "Length" parameter of the Chain command could be changed from 2 to 3 to correctly resolve the behavior. The Change leaf Gamut uses to represent this difference is shown in Figure 6.22b. Gamut could not decide why the value changed from 2 to 3 on its own because the developer did not highlight any object, so it will ask a question.

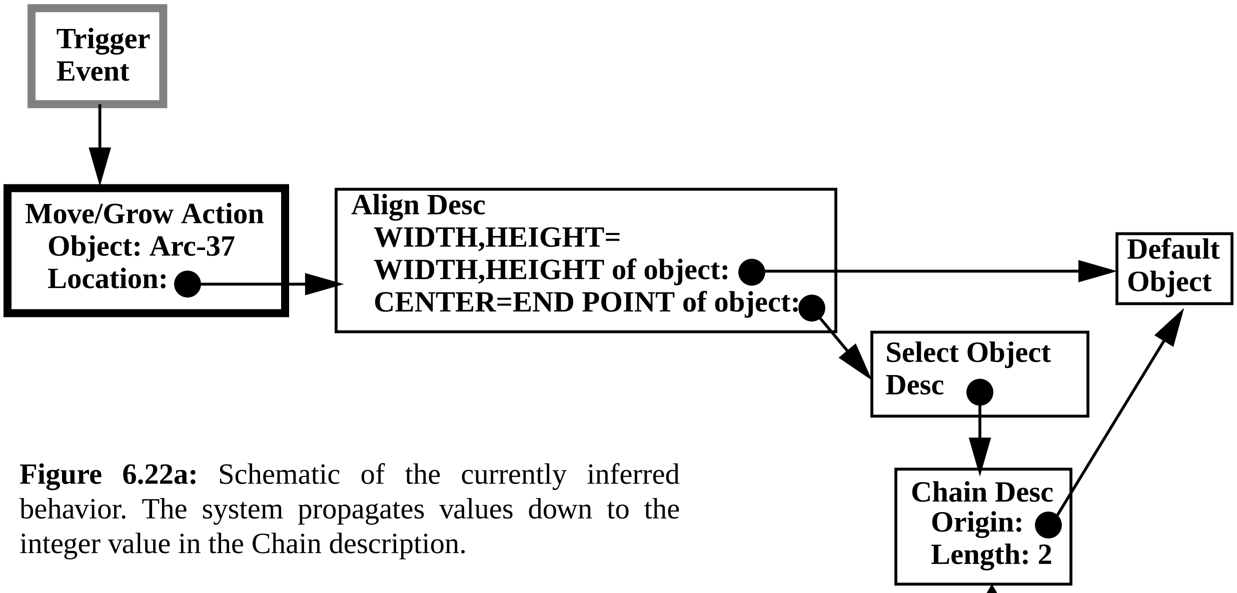


Figure 6.22a: Schematic of the currently inferred behavior. The system propagates values down to the integer value in the Chain description.

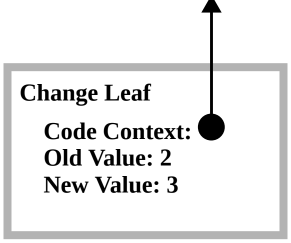


Figure 6.22b: The Change leaf in the changes set that shows that the length has changed from 2 to 3. The context points back to the behavior's code.

Taking the context of the leaf, Gamut finds the Chain description where the number parameter is stored and the Move/Grow action that moves the piece in the behavior. The intermediate Align and Select Object descriptions are ignored. Gamut calls the dialog method on the Move/Grow action and passes it the Chain description as well as the values 2 and 3. Gamut also reads the object parameter of the Move/Grow action and finds the reference to the piece object which it marks with a purple outline. The 2 and 3 values are neither objects nor locations so Gamut cannot mark them.

The Move/Grow action’s dialog method finds that the description context is valid so it knows that the description will provide most of the context. The Move/Grow dialog method recursively calls the dialog method on the Chain description passing it the old and new values. Dialog methods in descriptions are always called from an action so the description’s method knows to mold its text so that its caller can append more detail to the end. The method takes the old and new values and returns, “The length of the path has changed from 2 to 3.” The action’s method takes the description’s sentence and appends its own context, “for the action that moves the object outlined in purple.” It also adds the tagline, “Please highlight everything that the new value depends on and press Learn.” And, “To replace the original value with the new one press Replace.” The last two sentences are common for most dialog queries and are repeated for consistency.

Gamut’s dialog mechanism is actually too simplistic. It often fails to convey a sense of meaning to the developer who has a different notion about how Gamut’s behaviors are organized. For possible ways to enhance Gamut’s feedback, see Future Work, Section 8.1.5.

6.5.6 Storing New Descriptions Into the Behavior

Gamut stores a new description into the behavior by using the backpointer it stored in each leaf of the Changes set. When Gamut replaces the original description, it creates a Choice description and stores both the new and old description together. By not referring to the old value’s branch in the Choice’s boolean expression, it remains uncomputed, but is available for comparison with new descriptions. Gamut uses these “dead” branches to prevent itself from repeating incorrect inferences, and to create conditional expressions.

Gamut can only create Choice descriptions at points where they are allowed in the code. For instance in an Align description, the geometric constraint values are kept constant so Gamut is only allowed to put a Choice in the parameters that hold objects. When the difference method for the Align description creates backpointers for its parameters, it notes whether or not each parameter allows Choice descriptions. When Gamut stores a result in a parameter, it checks the mark. If the parameter allows Choice descriptions, Gamut creates a Choice using the new and old values and stores it in that parameter. If a Choice already exists there, Gamut adds the new value to it. If the parameter does not allow Choice descriptions, Gamut must still install a new Choice description but it must be placed further up the hierarchy. Gamut follows the backpointers in the context to find where the code object that disallowed the Choice is stored. If that parameter allows Choice descriptions, Gamut will place the new Choice there (or reuse an existing one), otherwise it will continue following backpointers until it reaches the action which is guaranteed to allow Choice descriptions. Once Gamut creates or adds a branch to a Choice, it no longer needs to create Choice descriptions further up the line of backpointers.

6.5.6.1 Matching New Descriptions

In deciding whether to add a new branch to a Choice description, Gamut compares the new description to all other descriptions in the Choice. Gamut is not allowed to use a description twice in a given parameter. Gamut actually uses a looser criteria than strict matching because often the complexities inherent in some descriptions can hide when two perform exactly the same calculation. Gamut’s first test is to see which descriptions already present in the Choice return the same value as the new description. Descriptions that return different values cannot possibly match. Gamut then calls the matching method of the new description passing to it the descriptions whose return values matched.

For example, the matching method for the Get Property description tests for a matching type and whether its two parameters match those of the incoming description. If the matched description is also a Get Property and its object and property value match, then the two descriptions match and the matching method returns true. Normally, a match can be discovered by examining only the values of the description's parameters. The criteria for matching some descriptions can be complicated, though. For instance, a Connect description can match a Chain description if they use the same reference object, have identical geometric constraints, and the Chain description's length is one.

If a new description matches one in the Choice description, the new description is not stored because it is essentially already present and does not need to be stored again. Resolution for the Change leaf that produced the new description is considered to have failed. When a node fails, Gamut attempts to resolve a different leaf from a common Or-node in the Changes set. For instance, if the Choice description in question were being used as an if-then expression, then the developer would want the system to learn to switch between the branches and not perpetually create new branches.

6.5.6.2 Marking New Branches

Gamut marks newly created branches to indicate which descriptions were revised to create the new ones. Gamut treats a new branch as a successor to the branch that the Choice invoked in the original behavior. The original branch and its successor are considered to be from the same "family." Since Gamut uses heuristics to revise descriptions, it is possible that a newly revised description is less correct than the description from which it originated. The Choice description's difference method handles this situation by propagating new values not just to the currently active branch, but to all branches in the same family, see Figure 6.23. This way, if a revision turns out to be less correct than the original, the original is still given a chance to be revised again when the developer gives another example.

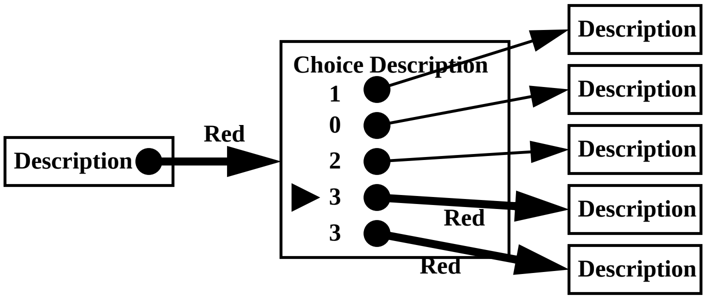


Figure 6.23: When Gamut propagates a value into a Choice description, the new value is propagated to all descriptions that are in the same "family" as the currently active description. Here the value "red" is passed to two item in the Choice. The triangle marks the currently active description. The numbers indicate in which family each branch is located. The zero family is used for "dead" code.

Gamut creates a new family of branches whenever the value of an inactive branch matches the value needed for the Choice. If the inactive branch was formerly a dead branch, then all dead branches that return the needed value become members of the new family. If the inactive branch used to be a member of an existing family, the family is split in two with members who return the needed value being placed in the new family.

6.5.7 Describing New Parameters

Every new action and every new description that Gamut creates has new parameters that must be filled. At first, Gamut stores constant values in these parameters. For many parameters, having a constant value is suitable. However, it is useful at times to infer ahead and replace the constant parameters with expressions based on the objects highlighted by the developer. This can save the developer the need to show further examples when the parameters are easily inferred. Also, some constant values may not be legal. For instance, referring to an object created by the same behavior is not possible since the object did not exist when the behavior was first invoked.

6.5.7.1 New Parameters Queue

Gamut stores the list of all new parameters in a queue. When all the items in the Changes set are resolved, Gamut proceeds to examine items in the queue and infer descriptions for their constant values. Each queue item contains three pieces of information: the value of the parameter, the context where the parameter is stored, and the list of objects that were highlighted at the time the new action or description was created. As parameters from the queue are described, the parameters of the newly created descriptions are added to the end of the queue.

Inferring a description for a new parameter works similarly to inferring a description to resolve a parameter difference. Descriptions are generated based on the type of value needed and which objects are highlighted. When Gamut describes a new parameter, it does not have to match the new description against other descriptions because there are no prior descriptions in the parameter to replace. Also, the lack of prior values means that Gamut will never need to install a Choice description for a new parameter. Most other caveats for creating a description still hold. For instance, the new description must not use forbidden objects and properties (see Section 6.4.4.3).

There are two kinds of parameters on the queue. One is used for standard values like colors, numbers, and locations. These kinds of values can be inferred using the standard methods. The other kind of item is strictly for objects. Since the developer may choose to highlight an object that has been modified, it is possible that Gamut will need to generate an anticipatory description for it (see Section 6.5.7.2). Gamut uses a search to find what action causes the object to change and replace it with the required description.

6.5.7.2 Describing Anticipated Values

Gamut uses searching to generate anticipated values. Given a graphical object that the behavior modifies, Gamut can find the action that modifies a given property of that object. When the needed action is discovered, Gamut then copies the description from the action’s value parameter to be used in the new context. In two cases, Gamut uses special descriptions to describe an anticipated value. If the anticipated value is a newly created object then Gamut uses the Create Object description, and if the value refers to a shuffled set of objects, then Gamut uses the Objects From Action description.

Sometimes the description for which Gamut must search will not be completely described at the time it is needed. Normally, when Gamut fails to create a description for a new parameter, the parameter stays constant. However, when the needed value is derived from a modified object, Gamut cannot leave the object as a constant because a constant object always refers to the historical value of the object’s parameters (not the modified value). If Gamut finds that the needed description is not fully described, Gamut defers describing the anticipated object by putting its new parameter item back into the new parameters queue.

Re-queueing an unfinished anticipated value allows other parameters to be processed before the anticipated value appears again. However, if Gamut reaches a situation where all leaf nodes expect an anticipatory result, the algorithm reaches an impasse. This occurs when the developer demonstrates a situation where all objects depend on the future values of themselves. For instance in Figure 6.24, the developer has moved three arrow lines at once and has highlighted all three as if to say that each arrow's position depends on the others. In this case, Gamut will take the first queue item and break the impasse by storing a constant value in its parameter. Gamut does not refer to the modified object but instead reads the object's property value and stores that. If there are more impasses, Gamut will repeat this procedure until all queue items are processed.

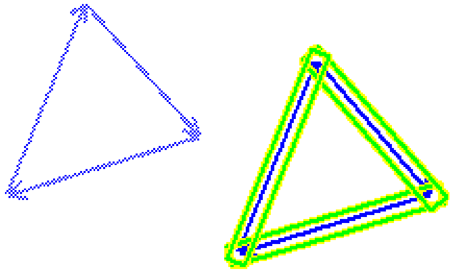


Figure 6.24: The developer has moved all three arrow lines at once and has tried to explain each arrow's position by highlighting the others. Since all arrows have moved, the system has no basis from which to create descriptions for these objects.

6.5.7.3 Eliminating Items from the New Parameters Queue

Sometimes a new description will be created but is later destroyed because it did not fulfill some criteria. For instance, a description may have been found to match with another description in the same context (see Section 6.5.6.1). When a description is destroyed, Gamut must eliminate its parameters from the queue, but it is difficult to know which parameters are affected. Some descriptions are created with parameters that are initially set with other descriptions. For instance, the attribute that tests whether an object's property is equal to some value is an Equal description with a Get Property description embedded in its first parameter.

Gamut removes all unneeded parameters from the queue before it starts so that it does not perform unneeded work. To do this, Gamut uses a mark-and-sweep algorithm such as in garbage collectors. Gamut first parses the behavior and marks items in the queue that refer to each description. Gamut them removes unmarked items from the queue.

6.6 Decision Tree Learning

One of Gamut's most significant improvements over previous PBD systems is the ability to learn complex, if-then-like, expressions strictly through demonstration. To do this, Gamut uses a inductive learning algorithm from the Artificial Intelligence community called decision tree learning [82]. In inductive learning, the system is given a body of data (usually a very large set of data) and tries to find relationships in the data called "concepts." As mentioned in the introduction to this chapter, programming-by-demonstration is itself a form of inductive learning. A PBD system like

Gamut transforms the data gathered by the developer's examples into behavior-like concepts. By using decision tree learning, one part of Gamut's behavior inferencing has been reduced to a task that can be accomplished using a standard AI algorithm.

Though the decision tree algorithm is well-known, Gamut uses it in a novel way. Though the algorithm itself is not significantly modified, Gamut automatically provides an environment where the algorithm's concepts, attributes, and data are created and collected. This section will show how Gamut produces new descriptions to form attributes that a decision tree recognizes including some clever attributes Gamut uses to make a decision tree perform tasks not performed in prior AI work. It will also show how Gamut uses demonstrated examples to build a table of data for the decision tree algorithm.

6.6.1 The Decision Tree Algorithm

The decision tree algorithm was invented by Quinlan [82]. Gamut uses the generic ID3 version of the algorithm [65]. The data from which a decision tree learns has a table-like structure as in Table 6.1. Each row of the table provides an example of the "concept" to be learned, and each column represents an "attribute" of the concept. For instance, the concepts might be whether the monster in a game of Pacman should move toward the Pacman object, move farther away, or move to the center "home" area of the board. Attributes for this behavior include whether or not the Pacman has eaten the larger "power pellet" dot to make the monsters run away and whether the monster has been eaten by the Pacman. These attributes are relatively easy to derive from the game. The Pacman game uses many graphical indications to show when different states are active, plus the developer would likely create flag-like checkbox objects in the off-screen area to keep track of such state. The diagram in Figure 6.25 shows a possible result of converting a database containing these data into a decision tree.

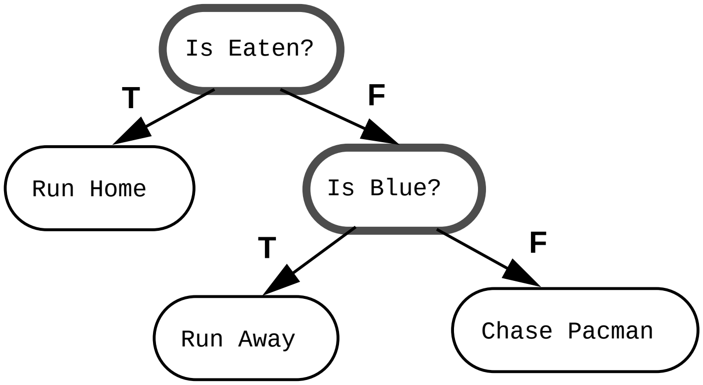


Figure 6.25: Decision tree to control a Pacman monster.

The decision tree table is filled with entries that correlate the attribute values for all the concepts. Note that the same concept can be repeated many times in the table because it is often true that many attribute-value combinations produce the same result. However, the same set of attributes cannot produce two different concepts. Table 6.1 shows an example database for the Pacman situation above. Here the attributes are boolean typed. (In fact, Gamut only uses boolean attributes.)

The “Run Home” concept appears two times in the table. It shows that the “Is Blue” attribute can take on either true or false, and as long as the ghost has been eaten, the ghost must “Run Home.” This table is trivial because it only has two attributes. Typically, Gamut will generate many attributes for a single table though only one or two attributes might actually matter for a given concept.

	Is Eaten?	Is Blue?
Run Home	true	true
Run Home	true	false
Run Away	false	true
Chase Pacman	false	false

Table 6.1: Database of examples for the Pacman decision tree. Each row represents the various actions. Along the top are the attributes worded as attributes.

The decision tree algorithm uses a statistical measure called “entropy” that relates how well an attribute corresponds to a concept. The algorithm chooses the attribute with the best entropy value to be the root of the tree, eliminates that attribute from the database, and recurses for each branch of the tree. The formula for entropy is shown below:

$$Entropy(S) = \sum_{i=1}^{c} -p_i \log_2 p_i$$

The entropy formula says the entropy of a collection, S, is related to a sum of equations based on the number of categories in that collection. The variable, c, is the number of categories, and p_i is the proportion of the elements in the collection that have the value of the given category.

The entropy is further combined to find the information gain for each attribute’s column. The formula for information gain is:

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

Once again the equation adds up a series of values based on the entropy of each value in the column. In Gamut’s case, $Values(A)$ is always the set $\{true, false\}$.

The decision tree algorithm computes the information gain for each column in the table. The column with the best value indicates which attribute is the best correlated with the concepts. For instance, the best correlated column in Table 6.1 is the one for the “Is Eaten?” attribute that has a Gain value of 1, the Is Blue column’s value is $\frac{1}{2}$. Since the “Is Eaten?” attribute has the best Gain, it is used in the top node of the tree. The algorithm then splits the table grouping all the concepts where the attribute is true and the ones where it is false into separate tables. The algorithm then recurses on each smaller table. If ever a table only contains one concept, then the algorithm returns a leaf node containing the value of the concept. As Gamut adds more entries with more concepts to the table, the generated decision tree grows more diverse.

6.6.2 Where Gamut Uses Decision Trees

Gamut uses decision trees to compute the Choice and Select Object descriptions. These descriptions have two things in common: both use boolean expressions to determine their value, and both pick their values out of a set of possible values. The Choice description uses a boolean expression to select which branch out of a set to return. The Select Object description uses a boolean expression to pick objects out of a set of possible objects. The Choice description uses a decision tree in the most obvious way. Since there is a constant set of branches, each branch is assigned a concept in the decision tree table. The predicates of the boolean expression form the attributes of the table and the decision tree selects and combines the attributes to form the final expression.

The Select Object description uses decision trees in a less standard way. The set of objects that the Select Object uses is not constant. Objects in the list change depending on the state of the application. Also, objects can be created and deleted from the application making the set of objects variable. Since the set of objects from which the description must choose is not constant, the decision tree cannot assign each object to be a concept. Instead, Gamut uses a decision tree to group predicates into an aggregate predicate. Gamut trains the decision tree to learn whether or not an object belongs in the set. Gamut iterates through each object in the set and presents each, one at a time, to the decision tree. The tree is trained to return true or false depending on whether the object belongs in the set or not. To do this, Gamut creates parameterized attributes that evaluate to different values as each object in the set is presented. The way that Gamut parameterizes attributes is described in Section 6.6.5.

Gamut uses decision trees differently from the standard practice. For instance, each Choice and Select Object description keeps a separate table of data to support its own small, local decision tree. Normally in AI systems, there is only one large table of data, and one decision tree comprises the entire application. Also, Gamut generates the set of concepts and attributes dynamically, incrementally adding new ones as the developer presents new examples. Typically, decision trees use hand-crafted attributes and hundreds (even thousands) of examples to learn a complicated expression. Gamut’s automatically generated attributes tend to be more numerous than a standard decision tree’s set and the number of examples tend to be few since the developer has to demonstrate each one manually. Gamut’s decision trees tend to be small, maybe consisting of two or three concepts joined by one or two attributes. Though Gamut generates many attributes, the decision tree will only need to use a small number of them. Essentially, Gamut uses the entropy measure as a good way to pick out the one or two attributes that matter. In many AI systems, decision trees tend to use all their attributes, which usually number in the dozens and sometimes number in the hundreds, to describe their concepts.

6.6.3 Attribute Descriptions

The purpose of the attributes is to gather information from the application in order to make a decision. Gamut creates attributes in a similar way as creating descriptions for parameters (see Section 6.5.2.2). The main difference is that Gamut does not have a value that it wants to satisfy. Normally, if Gamut needs to describe a color, it has heuristics for searching the application for the desired value and drawing together a relationship using a description. Here, Gamut only knows that a situation has changed and that the developer has highlighted objects that hopefully can be used to distinguish one situation from another.

Gamut's attributes can recognize four kinds of graphical object properties: the values of properties like color or font, type related information such as the number of objects that have the same description, geometric relationships between pairs of objects, and numerical tests such as testing closeness.

6.6.4 Creating Attributes

There are two situations where Gamut can create attributes. The first occurs when a decision tree expression is first created. Here Gamut creates an initial set of attributes focusing on the properties of objects, specifically singleton property attributes, geometric attributes, and type attributes. The second situation occurs when Gamut finds a contradiction while adding an example to the decision tree's database (see Section 6.6.6). In this situation, Gamut adds new attributes and is allowed to add attributes for gauging proximity and relative values.

Gamut defines four attribute description types: Equal, Number Test, Connect Test, and Min/Max. Equal is used to test single properties and the type of graphical objects, and tests the equality of two parameters. To test properties, Gamut stores a Get Property description in the first parameter and a value in the second. To test types, Gamut puts a Get Prototype and a type value into the parameters. Number Test, which compares two numeric values, is currently used to count objects. Gamut stores a Count Objects description with the object description to be counted in the first parameter and a number in the second parameter. The Number Test attribute can be set to return true if the number of objects meets or exceeds the number in the second parameter. The Connect Test description relates the positions of two overlapping objects. The Connect Test stores a set of geometric constraints. It returns true if the objects meet the constraints and false otherwise. Min/Max is used to measure proximity and relative values. Min/Max can test a number of numeric values and can tell which one is the minimum or maximum value. This is used, for instance, to make a monster chase the player's character by choosing a path that is closest to the character.

The parameters of attribute descriptions are like the parameters of any other description and are allowed to be further described. When Gamut creates a new attribute, its parameters are added to the new parameters queue. If highlighted objects can be used to describe the values of the parameters, Gamut will create descriptions and store them in the attribute's parameters. For instance, this is how Gamut can compare the properties of two objects.

As an example, suppose the developer has highlighted two objects: a blue rectangle and a blue circle. When Gamut creates an attribute for the blue rectangle's color it uses an Equal description with a Get Property in the first parameter that reads the rectangle's color property and places the color blue in the second parameter. However, the second parameter's value, blue, can be further described. When the second parameter is read from the new parameter's queue (see Section

6.5.7), Gamut describes the color by creating a Get Property description for the circle and placing in the parameter. The final result is an Equal description with a Get Property in both parameters, one for the rectangle and the other for the circle.

However in the same example, one would not want to use the rectangle to describe attributes created for the circle. Gamut would also create an attribute for the circle that tests its color property at the same time it created one for the rectangle. Having two attributes comparing the rectangle and circle's color would be redundant. To prevent the circle attribute's parameter from being described by the rectangle, Gamut adds the rectangle to the list of forbidden objects (see Section 6.4.4.2). As Gamut creates attributes for each object, the object is made forbidden to attributes created for all subsequent objects.

Gamut can only generate "true" relationships. That is, Gamut cannot generate an attribute that initially returns false. This means that in order for an attribute to appear in an expression, the developer must demonstrate a case where the attribute is true. This can be a problem if the developer only highlights an object that would generate a needed attribute only when the attribute would be false. For instance in Figure 6.26, the monster object has two behaviors A and B (the actual behavior does not matter in this case). It performs A when the rectangle in the scene is red, but does B if the rectangle is any other color. The developer first demonstrates A while the rectangle is red. The developer does not need to highlight the rectangle while demonstrating unless A uses it for some other purpose which is assumed not to be the case. Then, the developer wants to demonstrate B. First, he or she changes the rectangle to blue and then demonstrates B in the usual way. Gamut will know that the developer is creating a new branch to the inferred behavior because the actions in A do not match B and so it asks the developer to highlight the objects that will go into the new Choice's boolean expression. Here, the developer highlights the rectangle (which is now blue, not red). Gamut creates an Equal description for the rectangle's color saying that it is equal to blue. Gamut did not see the color red and would not know to include it as an attribute. Searching the history of past demonstrations for other rectangle values is a poor idea because Gamut has no way to know which relationships are important. The history of all changes would create far too many attributes at once and would not likely produce anything more useful than what Gamut has already generated.

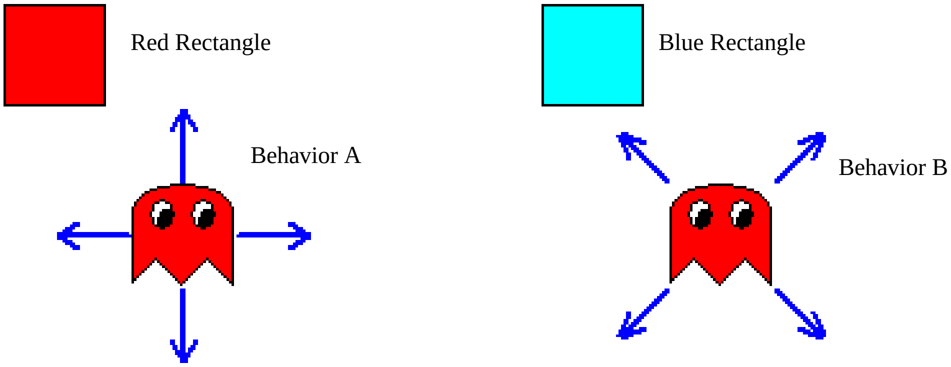


Figure 6.26: This monster's behavior is controlled by the color of the rectangle. The monster does one thing if the rectangle is red but does another action, otherwise. In order for Gamut to generate this relationship, the rectangle has to be red for at least one example where the developer highlights it.

In the example, if the rectangle only changes between two colors, red and blue, then there would be no problem. An attribute that tests for blue would be equivalent to one that tests for red. However, if the developer changes the rectangle to some third color, green, Gamut would likely switch to performing the A part of the behavior which is wrong. The developer would have to redemonstrate a case where the rectangle is red in order to make Gamut generate an attribute that knows about red.

6.6.5 Attributes For Select Object Descriptions

The Select Object description is used to select one or more graphical objects from a set. The attributes for the Select Object description are generated the same way as attributes for the Choice description; however, some Select Object attributes must construct relationships with the objects being selected. For instance, if Gamut needed to construct a description that picked out red rectangles, it would need an attribute to test a selected object's color and another to test its type. To create such attributes, Gamut treats the objects in the set as though they were highlighted by the developer, and uses special descriptions to parameterize the Select Object's attributes.

Gamut automatically highlights the objects in the Select Object's set by appending them to the list of highlighted objects. Highlighting all the objects in the set guarantees that Gamut will create attributes for all their properties. Gamut treats all the objects from the set as essentially a single object. For example, when Gamut adds a selected object to the list of forbidden objects, it actually adds all selected objects to the list. The reason for this special treatment is that the attributes are only used test one object in the set at a time. Gamut uses special attributes like Min/Max to test attributes between multiple selected objects.

Gamut uses the Current Object description to refer to an object from the Select Object's set. The Select Object description evaluates objects from its set one at a time. The Current Object description is a variable that returns the object that the Select Object is currently examining. For instance, an attribute that tests whether an object is red is shown in Figure 6.27. It consists of an Equal description with a Get Property description in one parameter and the color red in the other. The object of the Get Property description is the Current Object description. Thus, for each object in the Select Object's set, this attribute will return true if the object is red and false otherwise.

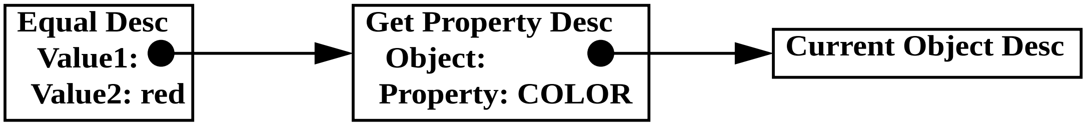


Figure 6.27: This attribute for a Select Object description would test whether or not an object is red.

The Select Index description returns the numeric position of the current object in the set. This attribute is for cases where one object out of a set of objects is being used in an action. Consider a behavior that creates four objects and puts each one in a constant location along the bottom of the application as in Figure 6.28. Since all four objects are identical, Gamut uses only one Create Object action to create all four. However, each object requires a separate Move/Grow action because the object's positions do not match. The Create Object action creates a set of objects but the Move/Grow actions each need a different element from the set. When Gamut describes the object parameter of each Move/Grow action it uses the Select Index attribute to pick an element out of the set. The attribute tests the number returned by the Select Index against a constant value and thus returns true for only one of the objects. Each object description for each Move/Grow compares against a different constant so they each get a different object. This mechanism might be further expanded to describe more complicated expressions (see Section 8.3.4).

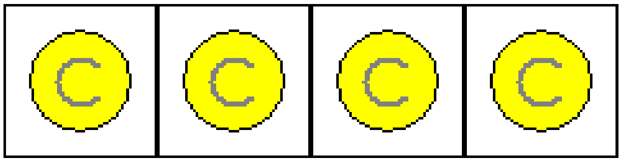


Figure 6.28: These objects are being placed in a row. Since the objects all look the same, they would be created by a single Create Object action, but since their locations are different, they have separate Move/Grow actions. The Select Index description can pick a single object from the group for the Move/Grow actions.

The Success Count description is designed for situations where only a fixed number of objects are desired in a set but there might be more that fit the criteria. However, this ability has not been implemented. The Success Count description returns the number of objects that have already successfully passed the decision tree's test. For instance, if the developer wants the set to return three of the red rectangles in the window, the Select Object description for this expression will contain a comparison (using Number Test) between the Success Count description and the number three. When the number of successes becomes equal to three, the comparison will fail and the Select Object can be trained not to allow more objects in the set.

6.6.5.1 Min/Max Attributes

The attributes for the standard decision tree algorithm are designed to be independent; the result of one decision tree calculation would not carry over into another. Gamut, however, relaxes this rule in order to make a new class of expressions possible using "Min/Max attributes." This class of attribute is only associated with the Select Object description and only for ones that return a single object. This is useful when Gamut needs to pick an object from the set that has a feature that maximizes or minimizes some value. For instance, if the monster in a Pacman-like game wants to move toward the PacMan, then Gamut must pick the *shortest* path that connects the two.

Normal attributes are not able to pick greatest and least objects in the object set because they only operate on one object at a time. Instead, Gamut introduces the Min/Max description which keeps a state variable throughout the object testing loop. At the beginning of the Select Object loop, all Min/Max descriptions are initialized. The first time a Min/Max is evaluated, it always returns true, but it also stores the value of the description it contains in a special variable that it retains. The next time the Min/Max in evaluated, it takes the new value that its parameter generates and compares it against the stored value. If the description is a Min description, it will return true if the new value is less than the retained value and then store the new, lower value in its variable. The opposite holds for a Max description.

To make Min/Max attributes work, a Select Object description always tests all objects in the set. Furthermore, the selected object is the *last* object that was successful, not the first. This ensures that any Min/Max attribute has found the minimum or maximum value that it needed and returned false for all values that appeared later.

Gamut does not create Min/Max attributes when attributes are first created. Instead, it creates them only when the current set of attributes fail and need to be revised. This allows the other properties of the objects to establish their attributes before the Min/Max attributes take effect. Gamut currently only makes Min/Max attributes for testing the distance between objects. Gamut creates Min/Max attributes by performing a heuristic search similar to the way Gamut searches for graphically constrained properties. Gamut first calculates the distance between the centers of each object in the Select Object's set with each highlighted object. If the object that was selected in the example possesses the minimum or maximum distance from a given highlighted object, then Gamut adds a Min/Max attribute to the attribute list that tests selected objects against that highlighted object.

Min/Max attributes have a number of interesting properties. For instance, a given expression will not normally use more than one Min/Max attribute at a time. To combine the effect of two Min/Max's means that only objects that minimize or maximize both values are selected. More often than not, what is really sought is to apply Min/Max-like expressions in a hierarchy, first one value is maximized (minimized) and then from the resulting set of objects that share the same value, another value is maximized (minimized). The problem with this hierarchy is that the first pass is not really a case of applying a Min/Max attribute. It has more in common with sorting or partitioning the set. Gamut currently does not have a description process that can partition or sort a set of objects. For this same reason, Gamut cannot express concepts like "pick the top three elements." These kinds of expressions are discussed in the Future Work, Section 8.3.4.

6.6.6 Adding Entries to the Decision Tree Table

Rows in the decision tree table are called "examples" in the AI literature, and in Gamut, there is nearly a one-to-one relationship between an example in a table and an example as demonstrated by the developer. The revising processes in stage two of Gamut inferencing algorithm (see Section 6.4) determine which branch is selected for a Choice and the proper set of objects for a Select Object. Each time a Select Object or Choice description is revised, the system will first try to add a new example row to the decision tree table. If the new row matches another row in the table, Gamut will need to create new attributes. New examples are formed by evaluating the set of attributes and pairing the result with the new concept value indicated by the revision. If the new example is different from the other examples, a new decision tree will be generated that will handle the new situation. The decision tree algorithm does not revise its tree structure, but creates a whole new tree when the database is changed.

6.6.6.1 Creating Examples for Select Object

A Select Object description can often generate multiple rows for its decision tree table from one demonstration because each object in the set can be treated as a separate example. Objects that the description returns are positive examples and objects not returned are negative examples. When the system revises a Select Object, it provides three pieces of information: the objects that the description was supposed to select (the selected set), the objects the description selected the last time that were wrong (the original set), and the whole set of objects from which the description had to choose (the whole set). There is also the situation where the Select Object description is created for the first time where the system only knows the objects from which it must choose (the whole set) and the portion of that set to be selected (the selected set). The three sets of objects in the revision situation and the two sets in the creation situation overlap to form Venn diagrams as shown in Figure 6.29. (The two set situation can be considered a special case of the three set situation where the set of objects selected last time is empty.)

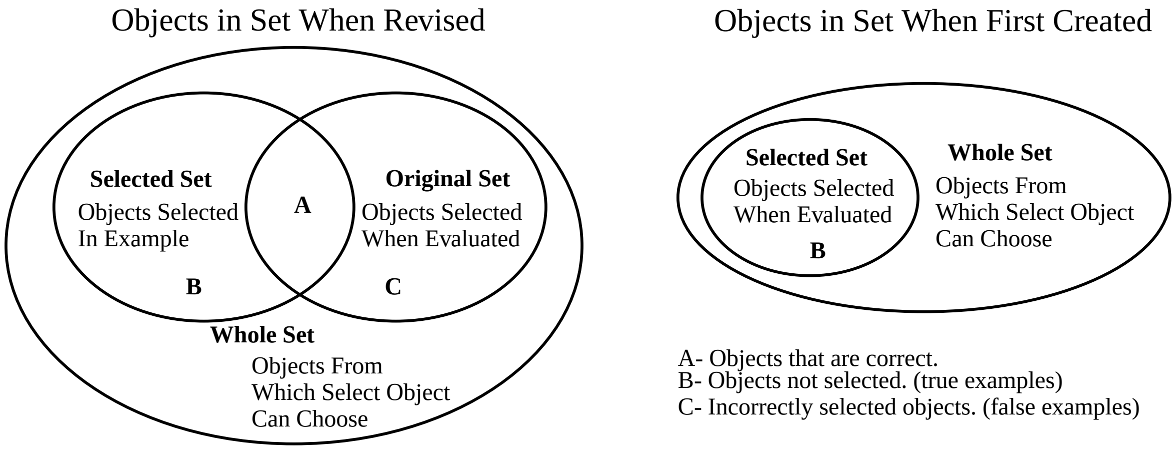

Objects in Set When Revised
Selected Set
Objects Selected In Example
A
Original Set
Objects Selected When Evaluated
B
C
Whole Set
Objects From Which Select Object Can Choose
Objects in Set When First Created
Selected Set
Objects Selected When Evaluated
B
Whole Set
Objects From Which Select Object Can Choose
A- Objects that are correct.
B- Objects not selected. (true examples)
C- Incorrectly selected objects. (false examples)

Figure 6.29: These Venn diagrams show the kinds of objects that Gamut must handle when revising a Select Object description. There is the set to return, the set that was rejected, and the set that the description has to choose from. When a Select Object description is first created, the rejected set does not exist.

The overlapping portion of the diagram between the selected set of objects and the original set of objects represents the set of objects that Gamut chose correctly. These objects do not contribute to new examples because the Select Object's decision tree already supports them. The objects that appear in neither the selected set or the original set also do not contribute to new examples because Gamut has no information that either supports or denies that these objects could be used in future sets. (Gamut may only be selecting a fixed portion of objects out of a larger set of potential objects.) Objects that create new positive examples are found in the set difference between the selected set and the original set. These objects were not selected by Gamut but have been selected in the new example, so they need to be added to the table. Negative examples are generated by objects in the set difference between the original set and the selected set. These objects were selected by Gamut when the description was invoked but are wrong, therefore the decision tree must be revised to discover what has changed.

6.6.7 Adding New Attributes

When Gamut attempts to add a new example row to a decision tree's table but its evaluated attributes exactly match another example's row it causes a contradiction. This situation is explained as "noisy data" by the decision tree literature [64], but this assumes that the table's set of attributes is correct and that it is the data collection process that is faulty. Since Gamut's attributes are machine-generated, it is more likely that the attributes are faulty and not the examples. If the conflict is due to faulty examples, Gamut assumes that the developer will correct the fault by using Gamut's Replace feature (see Section 4.2.3.2). If the developer uses Replace instead of Learn to respond to Gamut's dialog concerning a decision tree modification, then Gamut will simply remove the conflicting row from the table and replace it with the new example.

However, if the developer highlights objects in the application and presses Learn, then Gamut responds by creating new attributes.

Normally, Gamut creates new attributes by forming them from the objects that the developer highlights, but Gamut may also attempt to revise attributes that it has already created. Revising attributes is a much less directed process than creating descriptions for other situations because Gamut does not have a value that it wants to satisfy. Gamut has two methods for revising attributes, both of which are implemented but only the first is actively used. Neither mechanism modifies the attribute that they revise. Instead, the revised attribute is added to the list of attributes leaving the original unaffected. The first method uses the difference propagation methods to attempt to convert attributes that evaluate to false into attributes that return true and the second method scans the attribute for constant parameters and attempts to find descriptions for those parameters.

6.6.7.1 Creating New Attributes Using Difference Propagation

In general, Gamut does not know whether a given attribute was supposed to return true or false for an example. If it did, then it could propagate the correct value through the descriptions in the attribute's parameters as it does for other descriptions. However, since an attribute only produces two values, Gamut can still use difference propagation by assuming that whatever value the attribute returns, act like it ought to return the opposite value and see if that produces a better attribute.

However, Gamut only attempts to revise attributes that return false and try to make them return true. Virtually any combination of values can make an attribute return false, but an attribute's parameters are more restricted when it returns true. Currently, only the Equal and Connect Test descriptions are implemented to use difference propagation for attribute revision.

As an example, an attribute that tested the color of an object and failed can be revised to succeed. First, the value, true, is passed to the Equal description at the head of the attribute. The Equal difference method, in turn, looks at the values it is comparing and passes the opposite value to each parameter it is testing. Tests continue to find objects that match the desired color and a color that matches the object and so forth. The results are captured in a Changes set (see Section 6.4.4.1) as occurs normally during difference propagation. Gamut then tries to resolve the differences in the Changes set using objects highlighted by the developer and other standard methods (see Section 6.5.2). If Gamut finds a way to revise the attribute, the new parameters are stored back into a new copy of the attribute which is added to the end of the attribute list.

6.6.7.2 Expanding Leaf Parameters

Another method to revise attributes is to describe parameters that have constant values. Using this technique, Gamut does not change the value of the attribute but only redescribes parts of it with deeper expressions. The purpose for this technique is to help developers who forget to highlight key objects in the application. The system may need to have an object highlighted in order to describe a value used in an attribute but when the developer forgets to highlight, the system must use a constant value instead. Allowing the system to expand constant values into expressions overcomes this problem.

Replacing constant parameter values with descriptions is similar to using the difference propagation methods. The original attribute is left alone and the system generates a new attribute that holds the new expressions. When Gamut creates a new attribute by expanding a parameter of an existing attribute, it cannot use the usual matching process to make sure it differs from all other attributes. In fact, the new attribute will match perfectly with the attribute that served as a prototype. Therefore, new attributes created using this technique must be stored directly into the attribute list bypassing the matching process.

Though this technique was implemented in Gamut, it was deactivated because it tended to cause problems. Because the new attribute was stored without matching it against others in the attribute list, Gamut could not tell when it was creating the same attribute twice. Though the mechanism that replaces the constant parameters with new descriptions should have prevented the system from repeating itself, this part of the algorithm was difficult to make work correctly and eventually the entire mechanism was disabled.

6.6.7.3 Missing Entries in the Decision Tree Table

When Gamut adds attributes to the decision tree database, it creates gaps in the examples that were already there. These gaps occur at the end of old rows where the new attribute columns are added to the attributes that existed at the time the example was recorded. If in a later demonstration, the developer provides a new example that has the same prefix of evaluated attributes as another row in the table, Gamut will replace that row with the new example. This does not change the original values of that example but only extends it to fill in missing information.

Gamut’s decision tree algorithm had to be modified slightly to be able to handle examples that do not have a full set of attributes. Normally, when the algorithm picks an attribute to be a new node in the decision tree, it will divide the rows into sets based on the number of different values in that attribute’s column. The rows that have a given value go into the set whose sub-tree is connected to the branch with the same value. Gamut only supports boolean attributes, hence there are two sets of rows and two branches from every decision tree node. However, having unknown gaps for some of the rows creates a third kind of value. Gamut treats the unknown entries as a single unit and must decide whether to put them into the true or false category. When the database is split, Gamut puts the unknowns into the category with the fewest entries. So, if the selected attribute has four examples where its value is true and no examples where the value is false, Gamut will group the unknown examples with the false entries.

From experience, it was found to be fairly common for the decision tree table in Choice descriptions to split such that there are no entries in one of the categories. This is especially true for relatively new Choice descriptions which have not collected many examples. The decision tree algorithm creates an attribute node with an empty branch when one of the categories is empty for that node. Furthermore, many attributes tend to become strung together in one long chain when they are all missing data for that category. By grouping the unknowns with the empty category, Gamut balances the branches of its decision trees and reduces the number of long chains.

6.6.8 The Size of a Decision Tree Table

The total number of entries in a decision tree’s table depends on how well the attributes match the intended expression. Quite often, the needed expression is small. A behavior may depend on a single property of a single object. In these cases, Gamut needs only two entries in its table.

The number of table entries needed for more complicated expressions can be much greater. The decision tree algorithm is not perfect and can be hampered by unrelated attributes and the order in which the attributes and examples were presented to the system. If the developer always highlights relevant objects, and Gamut has descriptions that can express the desired concept and puts them into the attributes, then the decision tree algorithm is guaranteed to produce the correct expression eventually. Experience with using Gamut to build more complicated decision trees such as the one used to distinguish the winning conditions of Tic-Tac-Toe suggests that large tables are not too difficult to build. The decision tree for Tic-Tac-Toe requires five attributes which are picked out of a set of around twenty and can be demonstrated in approximately six examples depending on which board configurations the developer shows first.

6.7 Inferring Geometry

Of all the concepts that Gamut infers, expressions based on graphical relationships are the most complicated. Most PBD systems in the past have found inferring geometry equally difficult, and many expend their entire inferencing repertory just to infer how the developer wants an object to be placed on the screen. By comparison, Gamut's geometric inferencing is much less sophisticated than that in some other systems. Since several other systems have concentrated on learning geometry, it was decided that other avenues needed to be explored such as learning conditional expressions. As a result, Gamut's geometric learning heuristics are only adequate for relatively simple tasks. When the developer uses appropriate guide objects to help Gamut learn geometric tasks, the system's expressive power improves greatly, but it still does not cover the entire range of possible geometric expressions. Adding features from other systems such as Chimera [47] would expand Gamut's geometric capabilities as discussed in Section 8.3.1.

Though Gamut only strives to be adequate at learning geometry, still a significant portion of the code is devoted to geometric heuristics. The code size for the methods in the Align description alone is comparable to the entire section of code which implements the decision tree algorithm and all the methods for all descriptions that use decision trees. Geometry is difficult to infer for a number of reasons. The human eye is very good at seeing complicated relationships in visual scenes. A developer who wants to build a graphical application therefore assumes that the system will be able to see a relationship that any person would see. But computer systems have not yet caught up to the sophistication that the human eye possesses, so the system makes due with only a limited set of heuristics.

Gamut must be able to perform two kinds of geometric inferencing that are similar but need to be handled separately. The first involves examining a graphical relationship between two objects and noting whether it meets a given criteria. For instance, the application might need to know whether a bullet object overlaps a space ship object in order to make the two objects become deleted. Another variation for the first form of inferencing provides one object and a relationship and asks the system to find all other objects that satisfy that relationship. An example for this is finding all the objects that are inside a rectangle.

The second form of inferencing involves finding a relationship that describes where to place an object on the screen. Basically, any object that the system moves must use this set of heuristics. While the first task involves a single graphical relationship, the second is more difficult because it can involve many relationships with several objects. Furthermore, the relationships must be able to generate an actual location for where to place the moving object.

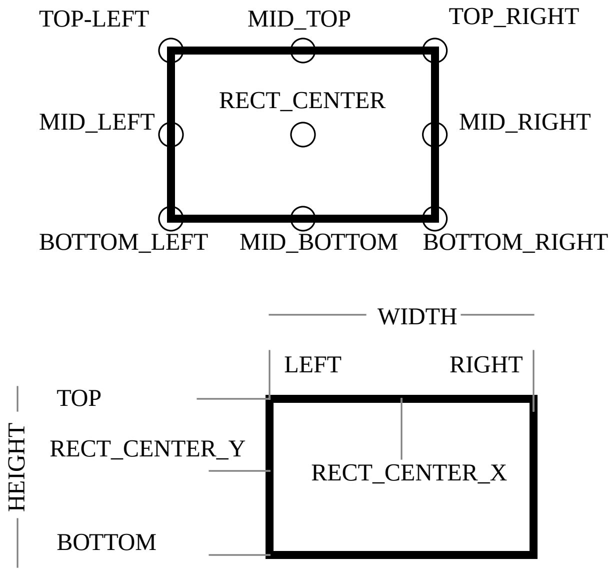
6.7.1 Defining Constraint Keywords

Gamut performs a limited analysis on objects to determine graphical relationships. Gamut uses a pair of keyword lists to represent graphical constraints. Gamut defines three kinds of keywords: points, dimensions, and areas.

When comparing two objects, Gamut only matches a limited set of *points* between the two. Gamut recognizes two kinds of graphical objects, rectangle-like objects and line-like objects. Rectangle-like objects include rectangles, arcs, ellipses, images, widgets, and text objects. A rectangle must be oriented to be square with the Cartesian axes of the window. Gamut recognizes the nine points in a rectangle shown in Figure 6.30a. The points include the four corners as well as the center points of the four sides and the center point of the rectangle. A few rectangle-like objects like text and buttons have a fixed width and height, and some like the text-input widget have a fixed height but variable width. For these objects, special consideration must be made to prevent Gamut from trying to change one of the fixed properties. Another special case for rectangles are point-like objects like the mouse pointers which only have one distinguished point and are defined to have a zero width and height. Line-like objects include the line and the arrow line. Lines are only divided into three points as shown in Figure 6.30b. Lines have the special distinction that they may be directed or undirected. Arrow lines are directed because the developer can tell which direction the arrow is pointing whereas regular lines are undirected because both ends are visually identical. Gamut does not support objects with a variable number of points like polygons or splines.

The various *dimensions* of rectangles and lines can be summarized in a set of eight keywords. These are `LEFT`, `RECT_CENTER_X`, `RIGHT`, `WIDTH`, `TOP`, `RECT_CENTER_Y`, `BOTTOM`, `HEIGHT` for rectangles and `X1`, `LINE_CENTER_X`, `X2`, `DELTA_X`, `Y1`, `LINE_CENTER_Y`, `Y2`, `DELTA_Y` for lines. These attributes are illustrated in Figure 6.30a and b. There is an obvious correlation between lines and rectangles when their properties are expressed this way as well as the obvious symmetry between the horizontal and vertical dimensions. The rectangle's coordinates are constrained so that the width and height parameters are always positive. Negative widths and heights are made positive by recomputing the other coordinates to match the rectangle's flipped location. The advantage of this notation is that Gamut can specify any location by describing any two horizontal coordinates and any two vertical coordinates. The disadvantage of the notation is that it omits angular or warping information. For instance, it is not possible to specify the length and angle of a line segment which makes inferring the motions of objects in some physical simulations outside Gamut's domain.

Gamut can also detect two graphical conditions that involve *area*. Gamut can tell when two objects overlap and when one object is contained within the boundaries of another. Containment and overlapping can actually be combined into four kinds of constraints shown in Figure 6.31. Given two objects A and B, A and B might just intersect, B might be inside A, A might be inside B, or A and B might have identical positions. These possibilities form a kind of Venn diagram where A being inside B is a special case of A overlapping B. Gamut can handle all these constraints and can generalize one kind of constraint into a broader category.



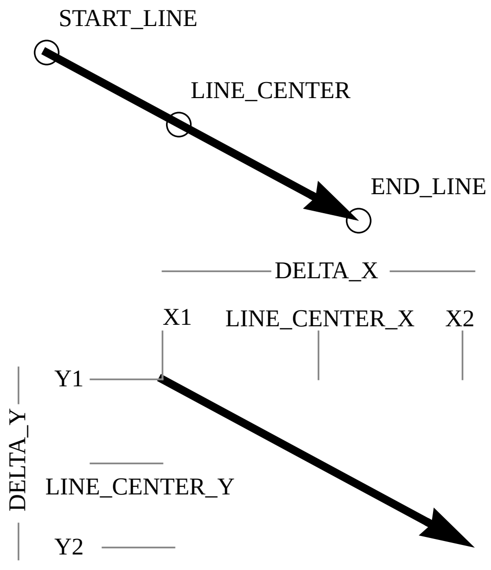


Figure 6.30a: The nine points and eight dimensions on rectangle-like objects that Gamut recognizes.

Figure 6.30b: The three points and eight dimensions on line-like objects that Gamut recognizes.

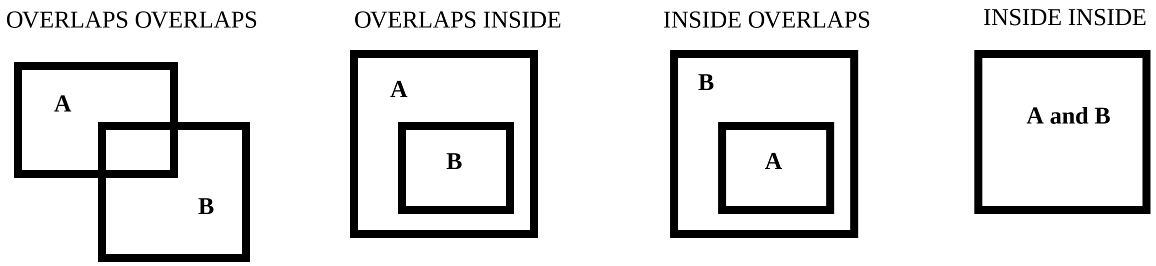


Figure 6.31: The kinds of overlapping-area conditions that Gamut can detect. Gamut forms area constraints from the keywords `OVERLAP` and `INSIDE`. The first keyword in each diagram refers to object A and the second refers to B.

Since Gamut uses a fixed notation to represent constraints, it cannot infer certain relationships without the help of guide objects. The most common involve relationships where two objects are “nearby” or spaced by an offset. The reason Gamut cannot infer when objects are near one another is because it does not know how close “near” is. Likewise, if an object is to be placed next to another with a constant offset, Gamut does not know how big to make the offset. The numeric value of a proximity relationship can be likened to application state. Gamut cannot infer when unseen state exists (see Section 5.6) so the unseen variable from where the relationship’s numeric value is derived cannot be inferred. However, if the developer makes the offset available in the form of a guide object, then Gamut can learn the geometry quite readily. For instance, to determine if an object is nearby another, the objects might be fitted with a guide object rectangle that extends the boundary of the objects. Gamut can detect when the guide rectangles overlap so it can tell when the visible objects are nearby.

It was previously mentioned that Gamut does not infer angles or use polar notation, though line-like objects could certainly benefit by their addition. Gamut's notation is also not sophisticated enough to infer graphical relationships like constraining the center of one object to always be in the boundaries of a second. Though Gamut can tell when such a relationship exists between existing locations, it cannot generate a location based on an imprecise geometric relationship. For instance, the application in Figure 6.32 would like for the player to be allowed to move the small dark circle anywhere on the screen and for the large rectangle to be moved so that the circle is kept inside the rectangle. Gamut cannot form an imprecise constraint like "somewhere around the circle," but instead has to be more precise such as, "put the center of the rectangle at the center of the circle."

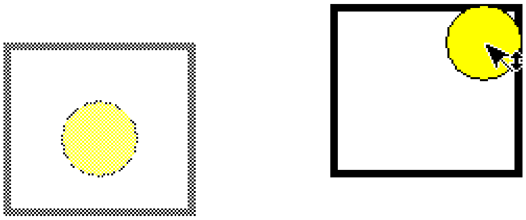


Figure 6.32: The rectangle is supposed to move so that the circle is always within its boundaries, but otherwise unconstrained. Gamut cannot handle this form of imprecise constraint. A more specific constraint like, "put the center of the center of the rectangle on the center of the circle" is possible, though.

6.7.2 Representing a Graphical Constraint

Gamut only searches for constraints between pairs of objects. Therefore, a graphical constraint consists of two lists of points, one for each object in the pair. The points in each list are the ones that both objects have in common. For example, consider a circle that is placed in the center of a square. The simplest and most specific form for Gamut to express this relationship is `[RECT_CENTER]`, `[RECT_CENTER]`. This simply says that the circle's center matches the rectangle's center. For the pair of objects in Figure 6.33, the relationship can be written as `[RECT_CENTER, BOTTOM_RIGHT, RECT_SIZE]`, `[TOP_LEFT, RECT_CENTER, RECT_SIZE]` which says that the corners of each object are placed in the center of the other object and that the two have the same width and heights. The `RECT_SIZE` (and the similar `LINE_SIZE`) keyword is unusual in that it represents a pair of coordinate values just like all the other points but it is not specific enough to represent a position on the screen.

To represent individual dimensions of objects, Gamut can use the dimension keywords in a relationship. Representing the relationship in Figure 6.33 becomes `[RECT_CENTER_X, RECT_CENTER_Y, RIGHT, BOTTOM, WIDTH, HEIGHT]`, `[LEFT, TOP, RECT_CENTER_X, RECT_CENTER_Y, WIDTH, HEIGHT]`. Gamut can use the dimension keywords to express relationships that do not involve full points. For instance in Figure 6.34, the relationship is `[WIDTH, TOP, RECT_CENTER_Y]`, `[WIDTH, TOP, BOTTOM]` even though no two points of either object aligns with another. Naturally, property elements can only be paired with other property elements. Also, Gamut keeps horizontal and vertical properties distinct. Gamut will not pair a horizontal property with a vertical one. In principle, this dis-

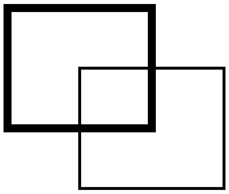


Figure 6.33: These two objects' graphical relationship is [CENTER, BOTTOM_RIGHT, RECT_SIZE], [TOP_LEFT, CENTER, RECT_SIZE].

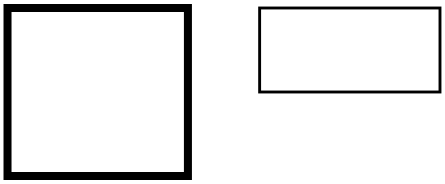


Figure 6.34: A relationship that uses property elements and not point elements.

tinction is artificial and only was added later because Gamut was making too many superfluous matches while it is was being tested.

Gamut uses the area keywords to represent connections between areas of the objects. When Gamut uses area elements to describe the relationship in Figure 6.32 the description is [INSIDE], [OVERLAPS]. The first list for the circle contains the keyword INSIDE which means that its entire area is within the bounds of whatever is indicated in the corresponding element of the other list. In this case, the corresponding keyword is OVERLAPS which says the some portion of the rectangle's area overlaps the corresponding element in the first list. Area elements can also be paired with points such as [OVERLAPS], [CENTER] which says that the center point of one object is located somewhere inside the boundaries of another. This relationship is more specific that saying that the two objects overlap and is more general than saying that one object is inside the other. It is also possible to match the INSIDE element with a point, but situations where this occurs is rare. Basically, what such a constraint would be saying is that the entirety of some object has been squeezed down to one point that is co-located with a point on the related object. Neither area element can be paired with a property element such as LEFT, or HEIGHT because only entirely specified locations can be compared with an area and property elements represent single dimensions.

6.7.3 Descriptions That Use Geometric Relationships

There are four descriptions that use geometric relationships in Gamut: the Connect, Chain, Connect Test, and Align descriptions. Of the four, only Align uses the relationships to generate a position. The other three use the geometric expression to either test the location of two existing objects (Connect Test) or to find a set of objects that satisfy a relationship with a given location (Connect and Chain).

The geometric expressions in all four descriptions are always constant values. Gamut has no descriptions that can calculate and return a geometric relationship. Furthermore, Gamut does not allow geometry parameters to contain Choice descriptions. Instead, when Gamut wants to modify a geometry parameter, the changed description gets propagated to a higher level (see Section 6.5.6).

6.7.3.1 Connect, Chain, and Connect Test

The Connect, Chain, and Connect Test descriptions have a fixed set of parameters. Connect has an object and geometry parameter, and Connect Test has a geometry and two object parameters. The Chain description is a little odd since it has an object, a number to count chain links, and one and a half geometry parameters. Recall that a geometry parameter has two parts, one for each object being compared. A Chain description needs to represent two geometric relationships, the first relates how the initial object is converted into the first link of the chain, and the second shows how given a link finds the next link in the chain. It turns out that the second half of both these relationships are always the same so Gamut only stores it once.

Evaluating the Connect, Chain, and Connect Test descriptions is straightforward. Gamut has a single function that when given two locations and a geometric relationship will return true or false depending on whether the locations satisfy the relationship. Connect Test uses this function directly. Connect and Chain use their given object as one parameter to this function and then iterate through the set of objects in the window one at a time to provide the other. Connect does this one time and returns the list of all objects that match. Chain uses its link count and repeats the test for each link in its chain.

Gamut places a restriction on the geometric parameters of the Connect and Chain descriptions that forces them to only use point and area elements in their relationships. This means that the objects tested in the Connect and Chain must actually touch one another to match. This restriction is used to speed up the searching process to find matching objects. The only objects that can match are ones that overlap the reference object. Thus, Connect and Chain can use an efficient test to filter out objects whose boundaries do not overlap the reference boundary before testing the geometric relationship which can be expensive.

6.7.3.2 The Align Description

Align has the distinction of being the only description in Gamut to have a variable number of parameters. An Align is like a list of Connects all chained together to form a single relationship. The geometric relationship of each item has a slightly different interpretation than usual. Instead of testing two locations, the relationship picks out the parts of the reference object’s location that are pertinent to the location being generated. The second list of each relation tells which part of the generated location that portion is assigned. For example, an Align description that moves the center of an object to the end point of an arrow line would consist of two constraints. The first constraint would relate the location of the object’s center to the end point of the arrow line. The second constraint would show that the remainder of the location, the width and height of the object, are derived from the object itself. A diagram of this description is shown in Figure 6.35.

6.7.3.2.1 Creating a New Align Description

To infer a new set of relationships for Align, Gamut uses a set notation to represent portions of a location so Gamut can track the portion that it needs to describe. The goal of inferencing is to match the locations of highlighted objects to the undescribed portions of the location. Whatever portion of the location is left undescribed is set to be a constant value. When Gamut adds a new

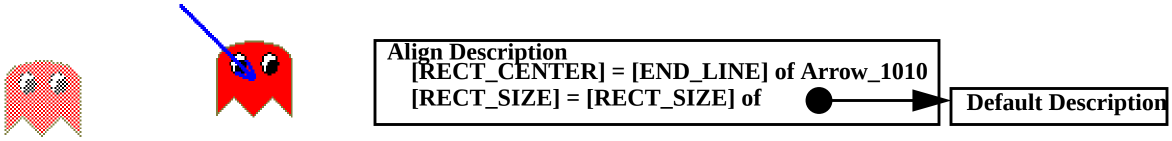

Align Description
[RECT_CENTER] = [END_LINE] of Arrow_1010
[RECT_SIZE] = [RECT_SIZE] of
Default Description

Figure 6.35: An example Align description that creates a location that centers an object on the end point of a specific arrow line.

item to the list of relationships, it records all geometric constraints that the object has in common with the desired location. This includes constraints that may refer to parts of the location that are already described. Gamut incorporates these constraints because the described portions of the location may be only accidental. The location may be described by a coincidental alignment of an object, or it may just be that the developer has not shown an example that exercises all of the locations' various dimensions.

6.7.3.2.2 Evaluating an Align Description

When Gamut creates a new Align, it does not try to separate the constraint contributions of the highlighted objects that it uses. This means that multiple relationships in an Align may specify the same dimension. Gamut uses a priority scheme to resolve conflicts in overconstrained Aligns. First priority goes to connections involving point-like keywords. Gamut considers a match between points to be very relevant and will try to maintain these connections unless it is contradicted by the developer. The `RECT_SIZE` and `LINE_SIZE` keywords, which represent both the width and height of an object together, are considered to be points. The second priority goes to dimension keywords. These connections fill in the gaps left behind by the points. Constraints are applied in the order they appear in the list of relationships. It is assumed that if a relationship is wrong, the developer will fix the problem by demonstrating a new example. Area-like elements such as `OVERLAPS` and `INSIDE` are not permitted in Align descriptions because the algorithm that converts the relationships into a location is not strong enough to handle them. In principle, there is no reason these elements should be restricted. The system would just require a better constraint solving algorithm.

6.7.3.2.3 Revising an Align Description

Revising an Align description is complicated. Gamut must track each dimension of the new location and test each item in its set of relations to determine whether or not to modify it. Gamut must make sure that it only affects those geometric relationships that are actually wrong and leave the others alone. Once again Gamut uses a geometric set expression to track the portion of the location that needs to be revised as well as the relationships that it has already finished revising.

Another complication concerns the direction of lines. When the developer corrects the position of a line, the line may be pointed backwards from the way it was placed originally. In order to generate a correct location, though, Gamut must distinguish between the line's end points. To solve this problem, Gamut propagates undirected line-like positions to Align descriptions twice: once in the direction as presented by the developer and once in the opposite direction. Experience has shown that when the line is tested in the wrong direction (whichever one that is), the difference methods will find less matches and be less likely to propagate changes as far into the chain of descriptions as the correct orientation.

Gamut simultaneously uses two schemes to revise an Align description: element removal and object swapping. In element removal, Gamut only modifies the graphical constraints of the relationships. Here Gamut goes through each constraint and eliminates elements that are no longer true. Because Gamut records extra constraints when it generates new Align relationships and because the priority scheme can potentially hide the correct constraint and keep it from being used, the hope is that the remaining constraints will still be sufficient to describe the location. If the resulting set of elements is sufficient, Gamut can return the revised Align without needing to use highlighted objects from the developer.

For instance in Figure 6.36a, An arrow line points to the center of a rectangle. Also, the arrow line's length happens to be equal to the rectangle's width. When Gamut first records this configuration, it creates the constraint `[END_PT, X2, Y2, DELTA_X]`, `[CENTER, CENTER_X, CENTER_Y, WIDTH]`. If in another example, the line and rectangle form a different configuration such as the one shown in Figure 6.36b, Gamut can reuse the constraint by eliminating the parts that are no longer true. In this case, the end point still points to the center of the rectangle but now the arrow's length matches the rectangle's height instead of the width. Gamut eliminates the `DELTA_X`, `WIDTH` pair of elements from the constraint resulting in the constraint `[END_PT, X2, Y2]`, `[CENTER, CENTER_X, CENTER_Y]`.

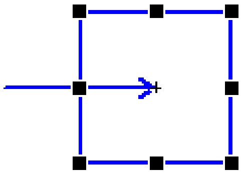


Figure 6.36a: Here an arrow line and a rectangle are graphically related. The end point of the arrow points to the center of the rectangle and the length of the line (delta-x) equals the width of the rectangle.

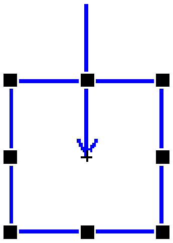


Figure 6.36b: In this configuration, the end of the arrow still points to the center of the rectangle, but the widths are now different.

When Gamut uses object swapping, it leaves the constraints alone and tries to replace the reference objects that the constraints use. For example, the description in Figure 6.35 refers to an `Arrow_1010` object in one of its constraints. When Gamut revises the description, it may find that a different object's end point matches the center of the object, in which case Gamut will create an entry in the Changes set that changes `Arrow_1010` to the other object. Gamut uses the priority mechanism to filter out elements in the constraints that do not contribute toward the value of the location. If the constraint for `Arrow_1010` contained extraneous elements, they would be filtered out. Once it finds the minimum set of constraints, it uses them to search for objects in the scene that fulfill the needed relationship. These objects are then propagated to the reference object parameter of that relationship. If Gamut can find a replacement object, it considers that relationship revised and notes that the corresponding portion of the location has been described. Of course, propagating the object to the reference object parameter does not guarantee that it will be described. The developer may have to highlight something in the scene to describe how the original reference object was converted into the new one.

6.7.3.2.3 The Origin of Tracking Redundant Constraints and Prioritizing

The original implementation of the Align description did not create redundant constraints. When Gamut used the location of a highlighted object to create each constraint of the Align, it would take care not to repeat any property that was already specified by previous constraints. Since the constraints were not redundant, there was no need to prioritize one kind of keyword over another. However, this created a problem because Gamut could not learn simple “follow-the-path” connections when the pathway was rectangular as in the example in Figure 6.37.

The problem occurs because each relationship was kept independent. In the example, when the developer moves the circle horizontally, Gamut only searches for a connection to the circle’s horizontal position. The circle did not move vertically so Gamut assumes that the vertical position is constant. When the developer shows the second example, the circle moves vertically but not horizontally, this makes the second translation appear to be a different kind of movement from the first. Gamut did not record any vertical translation in the first example so it does not know how to join the second example with the first. Thus, Gamut generates a Choice description for the two different kinds of movement and must learn how to branch between the two. Of course, there is no criteria that Gamut could use to pick a branch. Having the circle in any position is more or less the same as any other.

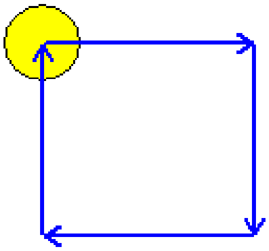


Figure 6.37: Before Gamut created redundant constraints for its Align description, Gamut could not infer an object following a rectangular path.

The solution was to record the redundant geometrical relationships as well as the ones for which Gamut was searching. But this revealed the second problem. It was first thought that the number of extraneous elements of a geometric relationship would disappear quickly. It seemed unlikely that two demonstrations in a row would hold a relationship that the developer did not intend to exist so those elements in the geometric relationships would quickly be contradicted. However, the unfortunate truth is that even when some relationships would be removed, new extraneous relationships would exist in the new example and tended to be added back into the description. This made winnowing down a set of relationships to the correct set very troublesome and it was difficult to understand why Gamut could not learn the correct relationship quickly. To remedy this situation, the priority scheme was added so that point-like elements would have priority over the dimension-like elements. The dimension-like elements were found to be more likely to cause extraneous relationships so they were given less priority so that some might become hidden.

6.7.4 Geometry Sets

Gamut uses a simplified list of dimension keywords which is called a “geometry set” to represent portions of a location. One way to think of the geometry set is that it is the minimum set of keywords that are needed to describe a portion of a location. When Gamut creates a new Align or revises a portion of an Align’s constraints, it has to track which portions of the location it is changing. It must know both what dimensions of the location are valid and which ones it has to modify. Gamut uses geometry sets to represent portions of locations.

The positions of graphical objects in Gamut, like rectangles, are set using properties like `LEFT`, `TOP`, `WIDTH`, and `HEIGHT`. The complication is that there are more than one set of properties that the developer may choose to use, for instance a rectangle’s position may be set using `LEFT`, `TOP`, `RIGHT`, and `BOTTOM`, or it may be set using `RECT_CENTER_X`, `RECT_CENTER_Y`, `WIDTH`, and `HEIGHT`. Location properties are not independent. Given any two properties in one dimension (horizontal or vertical) the other properties of that dimension are also known. The geometry set collects the properties that a given location specifies. There can be no more that two properties per dimension so the maximum size of a geometry set is four properties when it specifies an entire location. Gamut uses the same dimension keywords in geometry sets that it uses to specify geometric constraints. However, point-like and area-like keywords are not permitted.

The geometry set is called a “set” because Gamut can perform set operations on the structure such as union, intersection, and difference. In a union operation, two geometry sets are merged into one and any overlapping keywords are eliminated so that there remains a maximum of two keywords per dimension. In order to perform the other set operations, Gamut must be able to invert a geometry set. Inverting a geometry set shows which properties a constraint does not describe. The undescribed properties can then be filled by using highlighted objects or by searching for objects with corresponding locations.

To represent an inverted geometry set, Gamut includes a representation for *negative* dimension keywords. Along with `LEFT`, `WIDTH`, and `RECT_CENTER_X`, Gamut defines `NOT_LEFT`, `NOT_WIDTH`, and `NOT_RECT_CENTER_X`. Inverting a single keyword simply replaces that keyword with its counterpart. A dimension with two keywords inverts to having no keywords and a dimension with no keywords inverts to having two. (When Gamut creates a keyword pair from nothing, it uses the inverted properties, `NOT_LEFT`, `NOT_WIDTH` for rectangles, `NOT_X1`, `NOT_X2` for lines, and similar values for vertical keywords.) When Gamut unions an inverted keyword with a non-inverted keyword, the inverted keyword can subsume the other if they do not refer to the same property. For instance, `{NOT_LEFT}` union `{WIDTH}` equals `{NOT_LEFT}` (one property); whereas `{NOT_LEFT}` union `{LEFT}` equals `{NOT_LEFT, LEFT}` (fully described horizontal dimension).

The geometry set notation is sufficient for Gamut’s purposes. Gamut only uses two-dimensional graphics and does not support rotation. However, if Gamut were to be extended, it is likely the geometry set notation would have to be redesigned. The main advantage for the set is that portions of a location can be represented succinctly and that multiple portions can be combined easily. Adding rotation would make the union operation much more difficult to implement. To add rotation to Gamut, first, the set of rectangular keywords would increase to five (the usual set of four properties plus a rotation angle). Also, the meaning of `WIDTH` and `HEIGHT` becomes more complicated since the developer might mean the width and height of the rectangle itself or the width and height of the region the rectangle’s bounding box (see Figure 6.38a). Furthermore in lines, angle

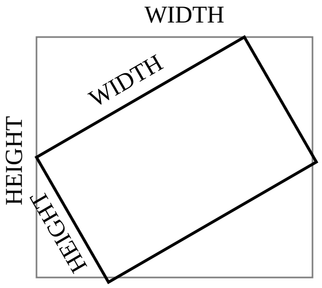


Figure 6.38a: With rotation, the height and width properties becomes confused with the rectangle's bounding box.

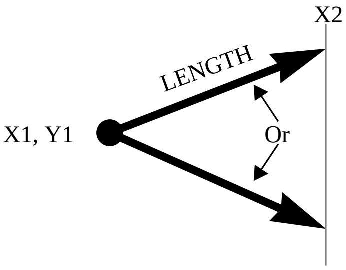


Figure 6.38b: Specifying four properties of a line can still lead to ambiguous situations.

and length properties can be either horizontal or vertical properties. For instance, say that the developer specifies a line by giving its length, its starting point, and the X position for its end point. The system would still need to disambiguate between two possible positions as shown in Figure 6.38b. Also, it is unclear whether could not use the NOT_ properties to represent inverted sets with rotation. Not having inversion would make finding intersections and differences of geometry set more difficult.

6.8 Speed and Complexity of Gamut's Algorithms

Gamut runs with reasonable speed on a standard 90MHz personal computer. A typical example requires less than one second to complete and the longest time to wait is about 30 seconds. In fact, with more modern computers, Gamut operates very quickly. This indicates that Gamut's inferencing algorithms are suitable for direct manipulation interfaces. Most of Gamut's algorithms run in time linear to the number of actions in the developer's example. This section will list the algorithms that have a higher complexity and take the most time.

6.8.1 Matching Complexity and the Number of Examples

Matching occurs during stage two, when the developer's new example is compared to the actions in the revised behavior. In general, the cost of analyzing a single example in the matching step is insignificant. The matching algorithm is $O(n^2)$ where n is the maximum of the number of actions in the behavior and the number of actions in the trace. The cost of matching any two actions is essentially constant. Since both the number of actions is typically very small (maybe five), the fact that it is an $O(n^2)$ algorithm is inconsequential. The per example revision cost is much higher during the propagation step of stage two and the searching phases of stage three.

The matching step determines the number of examples required to achieve a given behavior structure. In general, it takes at least $O(n)$ examples to create a behavior with n conditions. However, it is possible to formulate a series of examples such that it takes $O(2^n)$ examples to create the same n conditions. The developer can achieve this kind of behavior by purposely showing examples that activate and deactivate actions in a specific order that depends on the structure of the code being demonstrated. In practice, it appears that such exponential orderings are rare. It is also the case that behaviors usually have very few conditions, on the order of two or less. So, the fact that the developer may have to show four examples for a behavior with two conditions is probably not a problem.

6.8.2 Propagation Complexity

Propagation occurs after matching is completed and is the algorithm that sends new parameter values into the descriptions for those parameters. If a description's difference method is poorly designed, there is a potential for combinatorial explosion. The depth of the search is fixed to be the depth of the description hierarchy, but the number of changes that each description can find could be high. The structure of the application will normally restrict the search space adequately. However, difference methods normally restrict the number of changes they find to only those most likely to be the ones the developer means. Because each description has its own difference method, it can be tailored to meet the specific needs of that expression.

Typically, a difference method will restrict the number of changes it finds by eliminating potentially matching objects that do not meet some secondary criteria. For instance, in descriptions that use an object parameter such as Get Property, Connect, Chain, and Top Card, the difference method will only search for replacement objects with the same owner as the original object in that parameter. (An owner object is the object that contains a given graphical object like a window, card, or deck.) Objects in other owners may contain the property for which the description is searching, but to search the whole application would likely generate too many results.

6.8.3 Creating New Attributes

The process that requires the most time actually occurs in stage three when Gamut creates new descriptions. When Gamut creates or revises a Choice or Select Object description, it creates attribute descriptions for its decision tree expressions. Gamut does not know which attributes to create so it creates many at a time. The attributes Gamut creates for a description will also contain parameters and some of those parameters will be used to hold objects. When Gamut describes the objects of an attribute's parameters it will create Select Object descriptions. These Select Object descriptions will, in turn, contain still more attributes; thus, there is a cycle. The number of attributes Gamut can create given a set of highlighted objects is linearly proportional to the number of objects. However, since the attributes can contain still more attributes, the actual number of attributes created can be exponential (it is actually factorial).

Gamut uses aggressive searching and reuse to reduce the number of attributes it creates. Before Gamut creates an attribute, it first describes all the objects the developer has highlighted but without enclosing those descriptions in a Select Object so that Gamut will not create attributes for those descriptions. Then, Gamut adds a Select Object to each object description one at a time but does not allow the attributes of those Select Object descriptions to be further described. Finally, Gamut allows the attributes to be described including the attributes Gamut set out to create originally. Whenever one of those attributes refers to an object, it is forced to use an object description that Gamut has already created. This reduces the number of attributes to $O(n^2)$ where n is the number of objects highlighted. Though the number of attributes is still large, they can be created in a reasonable amount of time for the applications that Gamut was used to demonstrate.

6.9 Summary

The Gamut inferencing algorithms have been very successful. Gamut can infer a significantly broader domain of applications than has been possible previously. Whereas most other systems have resorted to the developer adding statements to the application's code, Gamut can create complex, conditional code with only demonstrated examples and hints gathered through highlighted objects.

Much of Gamut’s success can be attributed to a good underlying object design. The structure of Gamut’s language was inspired by the command object architecture in Amulet. This architecture merges methods and data and eliminates many of the burdens typical object-oriented languages require such as unneeded class hierarchies. Plus, other features in Amulet, such as automatically handling events, refreshing the screen, and providing undo history recording, makes a system like Gamut much easier to build.

Gamut’s language along with the hint highlighting interaction technique makes it possible for the system to infer conditional expressions between arbitrary objects in the application. Normally, PBD systems are limited to the immediate context of objects that are modified by the behavior. Gamut can use data from any source to affect the outcome of a behavior. (Of course, Gamut is still limited in how it can combine and interpret the data from those sources.) This ability eliminates the major reason why most PBD systems require the developer to annotate their code. Most annotations are used to select a portion of the behavior’s code and assign a conditional expression to it to guard its execution. Instead, Gamut creates conditional structures and guard actions on its own using the differences between the examples as its guide. Gamut can generate the attributes that determine which branch of a condition to take by creating decision trees using objects that the developer highlights as hints.

Through the recursive difference methods, Gamut can revise behavior at a finer level of detail than has been possible in prior PBD systems. Descriptions can form arbitrarily long chains where one description’s parameter is filled by another description. Gamut propagates revisions along these description chains using each description’s difference method that converts the desired result into changes in its own parameters. Gamut uses highlighted objects to resolve the parameter adjustments discovered using this method. This technique allows Gamut to find minor changes in the descriptions in the behavior that can resolve seemingly big differences in how one example looks from the next. It is also naturally incremental in how the descriptions can both be generalized and specialized by applying a new description into one or more of its parameters.

Gamut’s inferencing has proven to be powerful and useful in actual practice. Though it is a prototype system and lacks many features, it can still generate real-world applications using only demonstration. It also functions with reasonable speed and runs on home computers as well as high-performance workstations. Gamut can learn concepts with surprisingly few examples. Gamut’s inferencing techniques show that programming-by-demonstration can be used to create powerful and useful behaviors. These techniques will aid nonprogrammers and programmers alike in building complicated applications in a short amount of time.

Chapter 7: Usability Experiments

In order to test Gamut's interaction techniques and inferencing, two usability studies were performed. The first study was conducted as a "paper-based prototype" (or paper prototype) [85] where Gamut's interface was mocked-up with paper and cardboard. Here the experimenter played the role of the computer and manipulated the paper interface as though it were responding to the user. The paper-prototype experiment helped to identify large-grain interface problems in Gamut so they could be eliminated before any code was written. The second study was a more typical user study that involved the actual system. Volunteers were seated before a computer running Gamut and were asked to use it to perform tasks. The second study provided validation that a system like Gamut can work and showed what future systems would need to improve to achieve even more. Of course, Gamut was also tested informally during the course of implementation. As techniques were designed, they were tested by whomever was available.

The tests took place in a laboratory setting. A major consideration for conducting user studies in a lab is making good use of a limited amount of time. Session times can only last a few hours because a participant's time is valuable. Gamut has no manuals or help system, so instructions for using the system had to be easy to learn. Also, the experimental tasks had to be relatively short so they could be constructed in an hour. On the other hand, the tasks had to be thorough in order to validate most of Gamut's techniques.

Since Gamut's primary new interaction technique is *nudges* (see Section 4.2.1), the experiments tended to focus on this technique. The experiments also examined Gamut's other techniques such as cards and decks (see Section 4.3.1.1) and temporal ghosts (see Section 4.2.2.2) to see if they were understandable. For instance, it was interesting to see if developers could use the deck widget to demonstrate random events (via shuffling). It was also interesting to see if the study's participants would fall into any common patterns that would be useful for fixing interface problems or that required special heuristics in the inferencing system to learn. Since many of Gamut's techniques are optional, a developer may not use a particular technique during the course of an experimental session.

Gamut's two "formal" usability studies would be considered informal in the standard terminology [12]. Since the basic test was to verify whether nonprogrammers could use the system, it was not necessary to compare Gamut with any other system. Hence, neither study provides statistical results of any kind. However, it is still informative to test a system in a more controlled setting to provide observations that would be difficult to get under less controlled conditions. These observations, though anecdotal, can be helpful when other systems like Gamut are built.

7.1 Paper Prototype Experiment

Paradoxically, the user interfaces for some programming-by-demonstration projects in the past have been notoriously difficult to use. Seeking to correct that trend, Gamut's interface was tested from the very beginning. Writing code is time consuming and it is difficult to correct major user interface problems once the code is complete. Furthermore, it was important to know what sort of concepts Gamut's inferencing would have to learn and which concepts the developer would be able to represent explicitly. To determine these issues, a paper prototype study was conducted.

Appendix B contains a copy of the materials used in Gamut's paper prototype study. It contains all the pictures and materials that were used to produce the paper prototype as well as the forms and questionnaires the participants filled out. Though this reproduction is in black and white, the original materials used color.

7.1.1 Purpose

The purpose of the initial study was to evaluate the prototype interface that was initially proposed for Gamut. The design seemed efficient but no one knew whether users would be able to use the interface effectively. The original design contained most of the interactions that exist in the present interface including the nudges Do Something and Stop That, hint highlighting, and the card and deck widgets. It also contained several other interactions including the timeline, behavior icons, and mode switches (see Section 8.1.1). The study helped determine which techniques would be the most useful and which could be removed.

The study would also test whether a tool like Gamut was feasible. For instance, it might be that users would expect far more intelligence from the system than could be implemented. The interaction techniques of a programming-by-demonstration system are basically a kind of programming language. The study would help determine how users would use the language and whether they could communicate their intentions to the computer. We also needed to know how the computer should communicate with the user in order to resolve conflicts. The wording of a question often implies the form of the answer. Gamut's questions had to be neutral so as not to favor one answer over another, and yet still had to be direct so as not to be vague.

Finally, the study would help formulate the system's overall interaction. Gamut's original design implied a basic strategy for demonstrating an example, but it did not explicitly state how its various modes were entered and exited. For instance, most application building systems have a Run/Build mode switch that controls whether widgets are selectable and can be edited, or whether widgets are active and can be used as the player would [92]. The study would see how users expected the system to operate.

7.1.2 Experimental Design

It was decided early on that Gamut's first test should be a mock-up. However, there was an issue on whether to implement the test on a computer instead of using paper. In a "Wizard of Oz" mock-up [56], the participant is seated in front of a computer in much the same way as would take place if the program were implemented. However, the interface is only half-complete. The code that would run the interface is not available, and an experimenter uses a second computer to actually perform the operations behind the interface. Using a computer-based mock-up gives the participant the impression that the computer is working and the participant is more likely to act as he or she would if the interface were actually complete. However, building a computer-based mock-up requires that the modeled interface be fairly well understood. Even writing code for just the interface is time-consuming. Furthermore, there is the issue of networking another computer to act as the experimenter’s interface which will not even be a part of the final system. On the other hand, drawing the computer’s interface on paper is trivial as well as easy to modify. Though the participant knows that the paper is not a real computer, he or she can still behave as though it were a computer.

A paper-based prototype is sometimes whimsically called a “low-fidelity” mock-up. In a paper prototype, the computer is put aside and the system’s interface is drawn and acted out completely in paper. The method is described in Rettig’s report [85]. The most advanced version of a paper prototype session requires three people to pretend to be a computer along with the participant of the study. One person is the facilitator who talks to the participant and answers questions while another quickly prepares paper facsimiles of the computer interface. The person playing the computer quickly rummages through a stack of pre-made printouts and presents the interface to the participant. The third person sits off to the side and takes notes. An optional fourth person is the “greeter” who first talks to the participant and asks the participant to fill out surveys. This design is meant to ensure that results are impartial and very similar to the results one might achieve if an actual computer were used. For Gamut, a less involved paper prototype technique was implemented using only one experimenter. The experimenter plays both the facilitator and computer roles and uses a video camera to record the session and take notes later.

Paper prototype sessions are not a formal process. With only one experimenter, the procedure is even less formal. Also, because there was only one experimenter, the paper interface had to be made simple enough that one person could handle it. Therefore, graphical interactions like using selection handles and tool palettes were left to the participant’s imagination. The participant was given a background scene like the one in Figure 7.1. Pens and markers served as the way to create new objects by drawing them on the paper. The participant had a blank sheet of paper to act as the offscreen area. For objects that could be created, deleted, or moved, the participant was given cardboard cutouts of the objects. To move an object, the participant would simply pick up the piece and move it. No one was allowed to draw on anything made of cardboard because those items would be reused for the next participant. Dialog boxes and parts of the system like buttons and palettes were drawn on cardboard and given a simplified look. To build card and deck widgets, the participant had index cards. The participant could draw what was desired on the card. To make the cards into a deck, the participant stacked them. Shuffling the deck was handled by having the participant actually shuffle the cards.

The test also needed ways to simulate onscreen guide objects and hint highlighting (see Section 4.1.2.1 and Section 4.2.2.1). An onscreen guide object is a normally drawn object that can be made invisible on demand (so that the player cannot see it). To make the guide objects distinct, the participant drew them using highlighting markers. The bright color of the highlighting marker made those objects stand out and be recognizably different. Normal objects that are always visible were drawn using black pens. Hint highlighting was a more difficult interaction to represent. Highlighting is a temporary action that goes away once the question is answered, therefore the participant should not draw on the paper because that would be permanent. Instead, the participant used pennies to highlight objects. On one side of the interface was a stack of pennies. To highlight an object, the participant placed a penny on it. After accepting the response, the experimenter would remove the pennies and put them back in the pile. The whole set-up looked like the

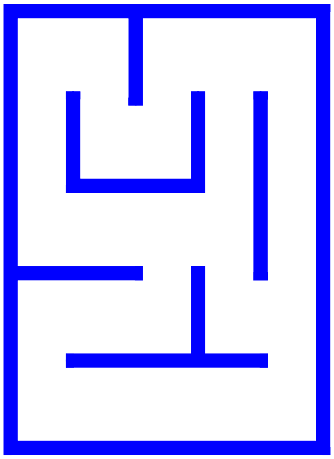


Figure 7.1: Background scene from one of the paper prototype tasks. This task was based on Pacman. The original image was enlarged to fit on a single 8.5x11 sheet of paper, and the large spacing of the maze made it easier to see what the participant was doing.

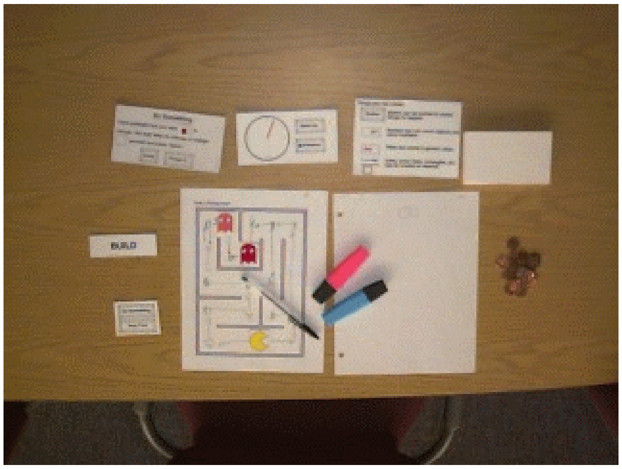


Figure 7.2: Photo the actual paper prototype materials. The participant would use the various pens to draw on the interface. The blank page was for the offscreen area. The pennies on the right are for hint highlighting.

picture in Figure 7.2. In all, the paper interface looked much more like a board game than a computer, but this effect seemed to work well.

7.1.2.1 Participants

The participants of the paper prototype study were drawn from the student population of Carnegie Mellon University (CMU). The main goal was to test nonprogrammers since this is Gamut's target audience. Since college students tend to be exposed to a wide range of fields and since Carnegie Mellon is known for its top-level Computer Science Department, the definition for nonprogrammer was not strict. People who had taken one or two programming classes to round out their education were permitted in the study. However, people who were taking Computer Science or Computer Engineering as a major and those who had programmed a computer to earn money were not allowed.

CMU has a sizeable liberal arts college so finding nonprogrammers was not difficult. University students are actually good candidates for people who would likely use Gamut in real life. Building applications is a creative process and not all people have the impetus to create something original. A university is a good place to find a collection of intelligent, creative people of which many will have enough time to be part of an experiment. Using students meant that the tests could be held locally at the University and there would be no need to set up special rooms and equipment besides a camera. The study was held in a typical conference room that contained only a large table and chairs.

There was some consideration about having children participate in Gamut’s test. Children are more difficult to study because they require special permission and are not readily available. For instance in order to use children, an experiment might be conducted at a public school. This would require permission not only from CMU but also from the children’s school as well as needing contacts within the school to help facilitate the activity. Since I did not have such contacts, and since the experiment did not need children in order to obtain useful information, it was decided that children would not be studied.

The invitation to participate in the study was distributed though the electronic bulletin board system. By using only the bboard and email to solicit and accept volunteers, it was guaranteed that the participants would have at least a basic level of computer skills. For the paper study, the participants did not require much computer skill, so this was not as great an issue as it was in the later study. A total of five people were tested. It ended up that all the paper prototype study’s participants were undergraduate level students with a median age of 21.

7.1.2.2 Procedure

The experimental procedure was straightforward. Participants first had to fill out a permission form and survey (Appendix B.1 and B.2). While the person was writing his or her answers, the experimenter would set up the camera and prepare the paper prototype. The camera was set to record only the paper interface and the participant’s hands. The camera could also record the sound. The participant’s face was not on video.

The survey was designed to gauge how likely it would be that the participant would want to use a system like Gamut to build actual games. The survey asked whether the participants played board games and video games, and whether the person would be inclined to create his or her own games if it were only easy enough. Surprisingly, some participants wrote that they did not like games at all. Also surprisingly, the results of the survey did not correlate with how well the participant performed using Gamut.

After the participant finished the survey, the experimenter turned on the camera and provided the study’s instructions. The participant was told about paper-based prototypes and how to use the materials (see Appendix B.4). For instance, the participant was told to use his or her finger as the mouse. To provide verbal information, we had the participants follow a “think-aloud” or “verbal” protocol [34] where the subject articulates his or her thoughts while interacting with the interface. Most subjects did not maintain the think-aloud protocol for very long. Most would drift into silence after a short while. At these times, the experimenter would try to talk to the participant and ask what he or she was thinking.

Gamut's interface would be modified after each subject in order to improve the interaction techniques. As a result, the findings for any one participant would not necessarily carry over to the others. Since the goal was to make Gamut's interface easier to use, it did not matter whether quantitative results could be generated from the experiments. As a result, the first participant had great difficulty performing the tasks whereas the later participants had significantly greater success.

Before the participants were allowed to begin the tasks, the experimenter first had them demonstrate simple behaviors as a tutorial (see Appendix B.6). Each participant was asked to draw lines and demonstrate having objects move around the screen. This prepared the participant to be able to demonstrate behavior for the actual tasks and it also served to get the participant in the habit of drawing objects on the paper screen. The tasks that the participant would later be asked to demonstrate all had a background scene already drawn for them. There was no tutorial in the first session, and it was found that the participant would not draw guide objects in the interface. Forcing later participants to draw during the tutorial helped inform them that drawing was necessary. The tutorial also introduced cards and decks as well as Gamut's other widgets and techniques.

The length of a session was to be no longer than two hours. Remarkably, the tutorial portion of the session could take as much as an hour and a half, yet even then all subjects finished at least the first task. The study used three tasks that are described in the next section. After the participant finished the tutorial he or she would be presented with one task at a time and allowed to work on it until it was finished or the participant gave up. Each task was designed to take about 30 minutes, though the actual amount of time people used varied widely.

For their two hours of work the participant would receive $5 whether they finished the tasks or not. Though offering a cash incentive for people who finished more tasks was considered, there was no fair criteria on which to base the incentive. Since a paper prototype is so informal, offering some people more money did not seem appropriate.

After the tasks were completed or time ran out, the participant would be given a questionnaire designed to elicit new ideas for Gamut (see Appendix B.3). It contained questions like "What would you do to improve Gamut?" or "What aspect of Gamut was the most difficult to use?" As it turned out, the responses to the questionnaire were completely useless and provided no interesting data. Most participants responded with short, vague statements that sometimes would even contradict what occurred during the tasks. When the survey was completed, the participant would be debriefed with a short statement (see Appendix B.5).

7.1.2.3 Tasks

The three tasks for the paper prototype study (see Appendix B.7 and B.8) were designed to be short enough to be implemented quickly yet be indicative of tasks that actual developers may want to create. The first task was based on a board game like Parcheesi or Sorry and consists of a pair of pieces that move around a board. The other two tasks were based on video games; specifically, Pacman and Space Invaders. Even though modern games use much more involved graphics and special effects, the essential gameplay has not changed much from these early games. Players still shoot the bad guys, pick up items for points, and dodge monsters.

All of the tasks required three elements to complete. First, they all required the participant to draw guide objects on the board or on the characters to indicate where things could move. Second, they all required behaviors that had to be demonstrated with more than one example. In other words, the participant had to show some competence at demonstrating examples. Third, all tasks required the participant to create at least one offscreen object in order to represent the game's state. These steps also implied that the participant had to highlight various objects in the scene as well as objects he or she created in order to tell Gamut that they were important.

The first task was called Pawn Race and it was based on board games like Sorry where pieces race around the board trying to get to the end position first (see Appendix B.7.1 and B.8.4). The board and pieces for this game are shown in Figure 7.3. The participant had to demonstrate that each player's piece moved around the board moving the dice's number of spaces. It was a two-player game so the participant also had to demonstrate that the two pieces alternated turns. This required an offscreen object to represent the current player's turn. As another twist, the participant had to demonstrate that when one piece landed on another, the piece landed on would have to go back to the starting square. Finally, in order to win, a piece had to move around the board three times. This required still more offscreen state and the participant had to think of a way to represent that a piece won.

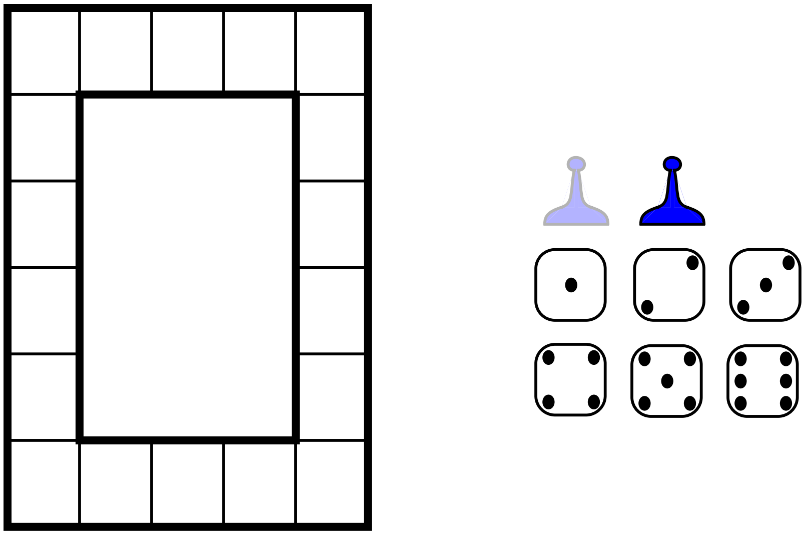


Figure 7.3: Board and pieces for the first task called Pawn Race. The two pawns race around the board to see which gets to the end first.

The second task was based on Pacman (and was called Pacman lest people get confused, see Appendix B.7.2 and B.8.5). In this task, the participant had to demonstrate how a single monster in Pacman moved. The Pacman character was assumed to be fully implemented and needed no other work. The basic idea was to see what the participant did to make the monster follow Pacman. The board and components for this task are shown in Figure 7.4. To make the task simpler, it was assumed that Gamut could determine that a path was the shortest in a graph as long as it was highlighted in full along with an object at the destination point. In fact, this ability is far more complicated than was first imagined and the final version of Gamut cannot actually do this. The participant was not told what Gamut could infer. The participant had to draw something in the maze for the monster to follow and highlight things to show what was important. Also, once the monster was moving, the participant had to demonstrate that when the monster touched Pacman, Pacman's lives would decrease and the monster and Pacman would move to opposite corners. Also, when the number of lives reached zero, the participant had to show that the game was over.

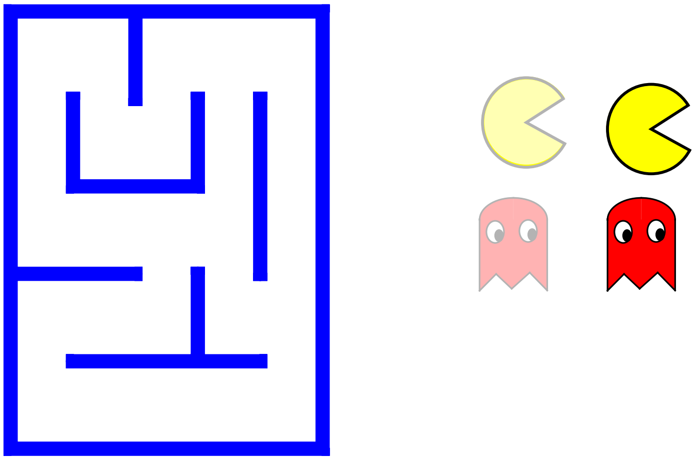


Figure 7.4: Board and pieces for the second, Pacman, task. The participant had to demonstrate the actions of the monster and make it chase Pacman.

The last task was designed to be more open-ended than the other two. The game was based on Space Invaders (see Appendix B.7.3 and B.8.6) and consisted of the player's spaceship shooting a randomly moving alien (see Figure 7.5). Here the participant had to use cards and a deck to demonstrate how the alien ship moved while still keeping it from moving off the screen. It was assumed that the player's spaceship could already move left and right along the bottom of the screen, but it still needed to be shown how to shoot bullets. Participants had to create a “fire” button and demonstrate how the bullet was created, moved, and disappeared as it left the top of the screen. The participant also had to demonstrate that when the bullet hit the alien, the alien would reappear in a random location. If the player shot three aliens, he or she would win the game. When this task was designed, it was not at all clear how the participants would handle it. It was hoped that participants who managed to get to the third task would devise an interesting way to demonstrate it.

7.1.3 Results

Since the paper prototype was an informal study, results could be tracked by taking notes directly from the video tape. During the study, most of the revisions and interface improvements were developed based on impressions the experimenter had with the previous participant. If the previous session went well, few changes would be made and if it went poorly, more changes would be made. After the study was complete, the video tapes were examined to look for problems common among the participants. The survey and questionnaire results as well as scans of the participants' drawings for each task is listed in Appendix C.

Overall, the participants performed well with the tasks. All of the participants finished the Pawn Race task and two of five finished all three tasks. The later participants had a slight advantage in that the experimenter had some experience with the problems that earlier participants faced and

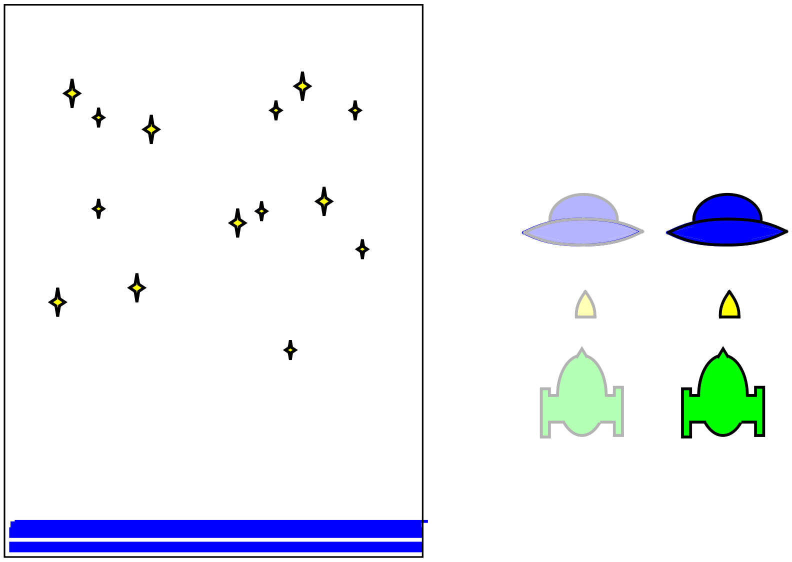


Figure 7.5: Board and pieces for the final task called Space Shooter. The participant had to demonstrate how the alien spaceship moved as well as how the player's bullets would move and shoot the alien.

could confront those problems in the tutorial phase of the session. Also, Gamut's design was getting simpler and easier to use.

7.1.3.1 Eliminating the Timeline Dialog

Simulating a computer on paper is not trivial. When a person has to work out what the computer ought to be doing mentally, one quickly learns shortcuts and tries to simplify the task as much as possible. Almost immediately when the first pilot user was tested, large portions of the computer interface were quickly forgotten and ignored. For instance, originally a timeline dialog (see Section 8.1.1.3) was to be simulated using strips of cardboard to represent the events (see Appendix B.8.11 and B.8.12). Each time the participant would do something using the interface, the experimenter was supposed to put down a strip corresponding to that action. However, none of the first participants ever even looked at the timeline. Attention was always focused on the main window. Quite often the experimenter would also forget about the timeline and would only update it sporadically. After the first two participants, it was clear that supporting the timeline in the paper prototype was useless. Furthermore, later participants seemed to perform just as well without it. This is a major reason why the timeline was never implemented in the actual system.

7.1.3.2 Eliminating the "Watch Me" Mode Switch

Another simplification eliminated extra modes in the nudges interaction. In the original design, the Do Something and Stop That nudges occurred after the developer entered Response mode. The developer would enter response mode by switching a toggle switch that would tell the system, in essence, "watch what I am doing" (see Appendix B.8.7). The "watch me" switch was meant to resolve the ambiguous situation where the developer wants to demonstrate new behavior and does not want the system to make a guess which is what Do Something would do. Also, when the developer switched off the "watch me" switch, that would signal when the example was complete. In the paper prototype, mode switches were represented by a single piece of cardboard with a word written on both sides. To switch the mode, the participant would flip the piece of cardboard over. For the “watch me” switch, the sides were labelled “Watching” and “Ignoring.”

One of the lessons learned during the paper prototype experiment was how strong a person’s mental barrier to modes can be. Users expect the computer to be in whatever mode they want it to be without their having to change it manually. This is true even when the user treats a computer artifact using obviously different interaction styles. It is not the case that users have trouble understanding what the different modes will do, it is simply that the mode changes in the person’s mind automatically whereas the computer must be told. As a result, the “watch me” switch was barely touched. It was either left on or off leading to serious problems. When left off, the computer would be unaware that the participant was demonstrating a behavior. When left on, the computer would watch as the participant started editing after the example and had to assume that those actions were also part of the behavior.

The “watch me” switch was eliminated by giving more responsibility to the Do Something and Stop That buttons and rearranging how the question dialogs would appear. As soon as the participant pushed Do Something or Stop That, a dialog would appear in the middle of their field of view. To make the dialog go away, the participant had to finish editing the example and push the “Done” button on the dialog. The addition of the dialog helped greatly. Since the dialog was partly blocking the participant’s view of the game, the participant had an incentive to make the dialog go away. This also helped to keep the number of actions demonstrated in a single example small. Still, the new solution was not perfect. The dialog worked best if the participant’s chain of thought was not interrupted. A flow developed where the participant activated Do Something, made a change, and pressed Done. If the participant made a sudden realization in the middle of a change, he or she would sometimes lose track and forget to cancel the dialog. This also happened in the final version of Gamut (see Section 7.2.4.6).

The streamlined version of nudges proved to be quite successful. The participants who used the new nudges technique understood the interaction quickly and had little difficulty. Other than forgetting to press Done when the example was complete, the participants had little trouble with the design. Some of the other misunderstandings included participants wanting to demonstrate multiple examples at once without supplying an intervening event. In other words, sometimes the participant would modify an object in the application and expect the system to infer that he or she was still demonstrating the same event. Another minor problem was that the participant would forget to activate the event that causes the behavior to occur in the first place. This was especially apparent for events that the system generated automatically, for instance with a timer. These sorts of minor problems did carry over to the actual system as discussed in Section 7.2.4.5.

7.1.3.3 Eliminating Run/Build Mode

Another major mode difficulty concerned the distinction between Run and Build mode. In most application builders, a mode exists to separate the time when the developer edits the application (Build mode) from the time when the application is being executed or tested (Run mode). Consider a button widget. In Run mode, the button can be pushed like a button, but in Build mode, the button can be selected, resized, and have its label changed. In the original paper prototype, alongside the “watch me” switch was the “Run/Build” switch (see Appendix B.8.7). To push a button, the switch had to be on the Run side and to move the button, the switch had to be on Build. Just like the “watch me” switch, the “Run/Build” switch was never touched. If the participant wanted to push a button, he or she pushed it. Likewise, no mode switch would prevent people from picking up an object and moving it. The paper prototype made clear that developers would have trouble with a separate Run and Build mode. This led to designing several interaction techniques for selecting and operating widgets without changing modes as discussed in Section 4.1.3.1.

7.1.3.4 Guide Objects

From the outset, it was suspected that creating appropriate guide objects would be the most challenging part of using Gamut. The paper prototype was used to see how developers could be taught to use guide objects effectively. The most common problem was that the participants tended to assume that no guide objects were necessary. Furthermore, participants would resist creating guide objects until after the system completely failed to understand anything demonstrated.

Not wanting to create guide objects is probably caused by an inherent desire to do as little extra work as possible. Demonstrating a behavior seems like the simplest way to tell the system to do anything. It is not clear from the start that it is much harder to demonstrate a behavior when the system has no basis from which to form inferences. After all, sometimes the system would seem to magically understand a behavior after one demonstration. Only after the system gets bogged down with ambiguous demonstrations did participants realize that something was missing from the interface and bother to create a guide object to represent the missing data. On the other hand, most participants did eventually create guide events after their initial period of fruitless demonstration.

When the system finds an ambiguity, it asks the developer to highlight an object that is important (see Section 4.2.3.2). If an object already existed in the interface that could be used to resolve the ambiguity, the participants were fairly good at highlighting it. If no such object existed, though, the participants were likely to highlight some other object that was important to the behavior instead of creating objects to represent the missing data. This happened often enough that the term “flailing” or “highlighting flail” was used to describe it. How badly a flail affected the system would depend on how often the inappropriately highlighted object changed and how many ways it could change. For instance, if the participant mistakenly highlighted the die in the Pawn Race task, then the system could assume that the number on the die was important and only perform behaviors when that number came up.

On the other hand, after flailing with the system for a period of time, most participants eventually did create guide objects to represent the missing data. Part of this success was due to the addition of the tutorial before the participant worked on any tasks. In the tutorial, the participant began with a blank sheet of paper. The experimenter then asked the participant to first draw a set of boxes on the paper with accompanying guide object arrow lines to show how the boxes were connected. Because guide objects were emphasized in the tutorial, participants more often remembered to draw them in the tasks.

Though it was hoped that some participants would use novel widget ideas to help them demonstrate behaviors, no such ideas actually occurred. It is likely that participants did not create new widget ideas because they did not want to create guide objects in the first place. A couple participants wanted to create what could be called “deus ex machina” widgets that essentially were specially tailored widgets that would accomplish exactly what the task called for but no more. Most of those ideas would have required advanced Artificial Intelligence to implement (such as the ability to understand English) and would not have been applicable beyond that one task.

7.1.3.5 Demonstrating Nothing as Something

Though this situation is not as common as the study's other findings, it is worth noting that some participants had a strong belief that demonstrating nothing at all should be registered by the system. For instance, in the Pacman task, one participant wanted to show that the monster was not allowed to move through walls. The participant was using an arrow line to show which direction the monster would move. Without demonstrating that the monster could move in the first place, the participant proceeding to point the arrow into walls and told the system to watch. This was the first participant so the "watch me" switch was still in use. Of course, the monster should do nothing when the arrow faces a wall, so the participant would immediately turn off the "watch me" switch and expect the system to notice that nothing happened.

After moving the monster to different places on the board and pointing the arrow at different walls and demonstrating nothing each time, the participant turned to the experimenter and asked, "Does the monster now know that it cannot walk through walls?" Of course, the system would not even know that the monster could move because that case had never been demonstrated. Also, the system cannot assume that when nothing happens that there will never be some future action that occurs at that time. For instance, the behavior may be incomplete and perform some parallel actions along with the actions currently being demonstrated. Though Gamut has not been implemented to handle situations such as this, perhaps a future researcher will see the proper way to handle it.

7.1.3.6 Informalisms

Though the experimenter is supposed to be vigilant and only treat the paper interface as though it were a real computer, sometimes the participant would not want to play along and would interact with the interface in a more relaxed fashion. For instance, instead of placing a penny on an object to highlight it, the participant would only gesture and say, "I highlight this object." How often the participant was allowed to do things like this depended mostly on the experimenter's mood. If the experimenter was too forgiving of these mistakes, the participant could make even worse hand waving kinds of assumptions. On the other hand, being too strict could anger the participant or cause him or her to be tense.

Another kind of informal behavior occurred when participants created guide objects. Although they had a whole blank sheet of paper to draw on, some of the participants preferred to draw on the edge of the background image. Of course, this would make the drawn image visible to the player, but the participants did not seem to care. If the experimenter asked if the player could see the objects, the participant would say that the image could be moved or recolored later. At the time the object was drawn, it was placed in the most convenient position in the most convenient way without concern for visual appeal.

The most troublesome informal behavior concerned the wording of sentences when the system asked the participant a question. Sometimes the experimenter would have to make up dialogs on the spot. The quality and uniformity of these dialogs were sometimes questionable. For instance, the experimenter might reword a dialog to see if a different sentence would help the participant more than others. This kind of dynamic interchange might have helped participants more than a real system would have. Furthermore, the more often the experimenter reworded a dialog, the more forgiving the experimenter became of mistakes the participant would make.

7.1.4 Summary of Changes Based on Paper Prototype Study

The paper prototype study had a significant impact on Gamut's design. The strongest influence was the refinement of the nudges technique and how the behavior dialogs were generated. There were also other influences such as how Gamut handles Run/Build modes and the elimination of some techniques from the thesis proposal such as mode switches and the timeline.

These are the changes that were based on the paper prototype study:

- The "watch me" switch was completely removed.
- Gamut's behavior dialogs were categorized into two groups: dialogs related to demonstrating behavior and dialogs related to asking questions.
- A "Done" button was added to the demonstration dialogs for the developer to end demonstration. The dialogs were also to be placed in an obvious location so the developer would be less likely to ignore them.
- The Run/Build mode switch was found to be undesirable so it was removed from Gamut. Instead of a single switch, Gamut uses a selection technique and multiple switches that control various interface features independently.
- The timeline widget was placed in a secondary status. Though the timeline had not been eliminated from Gamut's design, it was not implemented because developers did not seem to care to use it in the paper prototype study as well as for other reasons mentioned in Section 8.1.1.3.

7.2 Final Usability Study

The results of the paper prototype study suggested that a system like Gamut would likely work as intended. It took several years to implement Gamut. During that time, testing continued informally by testing aspects of the interface as they were implemented and having friends and colleagues stand in for nonprogrammer volunteers. Since much of the implementation concerned the inferencing algorithms, the interface stayed relatively unchanged through the whole period. After finishing Gamut, the interface was tested one more time to verify that the system could actually be used by nonprogrammers.

One of the claims of programming-by-demonstration is that it is easy to learn for nonprogrammers. However, previous demonstrational systems have rarely been tested with nonprogrammers in a formal setting. The largest exception to this rule was Maulsby's work with Turvy, Moctec, and Cima [55]. Most of the systems that Maulsby tested, however, were Wizard of Oz style mock-ups. Cima, the one system that did perform inferencing was not as thoroughly tested and only had a rudimentary user interface. Another notable exception was Modugno's Pursuit [66] which was a PBD system for manipulating files. Pursuit was actually constructed with two interfaces. Both interfaces were tested and compared to see which one was more understandable to users. Though it only possessed one interface, Gamut was also completed sufficiently to conduct a usability study.

7.2.1 Purpose

The final user study was designed to expose actual nonprogrammers to Gamut and see how well they performed. It was structured almost identically to the paper-based prototype study described is the previous section. The main goal was to see if the participants could learn to use Gamut well enough to demonstrate nontrivial behaviors. Participants would have to learn the system very quickly (in about an hour), and use what they learned to demonstrate game-like applications.

The nudges interaction technique was the most heavily tested feature. It was expected that most people would not have significant trouble with nudges after they learned how they worked. Other features in Gamut were also tested, but they did not factor in as strongly. For instance, the participants used cards and decks in each of the tasks. The first two tasks had cards and decks prearranged when the participant loaded the file, and the third task required the participant to build a custom deck to generate random events.

Designing and creating guide objects was troublesome for most of the paper prototype participants and it was interesting to see if this carried over to the actual system. The tasks were designed so that the participant had to create guide objects for almost every step. Though it was expected that participants would be reluctant to create guide objects at first, it was hoped that they could learn to build guide objects, eventually. The task instructions contained explicit instructions like, “Create an object to represent the player’s turn,” but did not tell the participant what to create. This eliminated much of the initial flailing that occurred in the paper study while still showing that nonprogrammers could create guide objects.

7.2.2 Experimental Design

The design of the final usability study was very similar to the paper prototype study. The write-up for the formal study as well as the paper materials that were used in the session are shown in Appendix D. The goal of the final study was to evaluate whether participants could use Gamut to construct behaviors. Therefore, Gamut would not have to be compared to other systems. The study would achieve positive results if participants were able to demonstrate nontrivial behaviors successfully. Like the paper study, the final study did not collect statistical results. Instead, the various behaviors that the tasks required were listed on a sheet of paper and each was checked off if the participant showed some expertise with it. The experimenter also tried to count the number of examples the participant made for each behavior, but as will be mentioned in the experience section, this was not always possible.

Since the study did not produce statistical results, there was no need to test many participants in the study. To produce a statistical result usually requires many subjects producing a large body of data to wash out statistical error and uncertainty. However, Gamut’s study was more like a proof of concept test. It is assumed that the people who use would use a system like Gamut would tend to be self-selecting. Thus, it did not matter whether everyone could use Gamut or what percentage of people could use it. The important matter was to show that at least some people could use the system effectively.

Making effective use of time was a critical issue in the final user study. Three hours were allotted for each session because that was about the longest time in which participants seemed to be willing to stay. Using longer times risked interfering with the participant’s normal schedules and the participant would become more likely to be tired and bored. However, using the real system is more difficult than using the paper mock-up. Though the tasks were no more complex than the paper prototype tasks, the participant was no longer able to articulate what he or she meant and had to actually use the computer to accomplish the goals. After a few pilot tests, the tasks were broken into two sessions. In the first session, the participant would learn the tutorial and attempt the two simplest tasks. If the participant did well, he or she would be invited back to a second session where a second tutorial would explain some more of Gamut's interaction techniques and the participant would try a third, more involved, task.

Once again, money was offered for participating in the study. The participants received $15 for taking part in the first session and received an extra $5 if he or she were invited back for a second session. More money was offered in this study than in the paper prototype study because the time commitment was longer and because using a computer is more troublesome. However, by offering more money, the study likely enticed people who did not really care about the experiment into participating.

7.2.2.1 Participants

Since the method used to find participants in the paper prototype study was so successful, the same technique was used for the final study. Members of Carnegie Mellon University were solicited by advertising on an electronic bulletin board service. The main concern was that the study was run during the summer break in July. This meant that fewer students would be attending school.

The participants were to be nonprogrammers as in the paper study and the same criteria was used to judge whether the participant was a programmer or not. It was okay if the participant had taken a class or two to learn how to program. However, people who wanted to program computers as a profession or who had earned money through programming and people majoring in Computer Science and related fields were not allowed.

Like the paper study, University students were recruited because they were locally available and were amenable to taking part of the study. The study did not use other venues to find participants for same reasons as in the paper prototype study (see Section 7.1.2.1). The advertisement did not forbid faculty and staff from participating and as it turned out, some staff members volunteered. This was likely caused by the summer timing of the study. The increased number of staff participants pushed the average age of the volunteers up to 28 with a median of 21. The study also received one adolescent volunteer, aged 14, who was recruited through his mother. The mother thought the study would be a good way for the child to earn “spending cash for the summer.” It was clear from the session, however that the child did not actually want to participate in the study on his own. As a result, the child's session was not considered valid.

The study also received one other volunteer who was unsuitable for the study. This person was a CMU staff member who had never had any experience using a drawing editor. The person did not know what selection handles were and could not learn how to consistently create objects using the tool palette. Since the study had recruited participants through the electronic bboard and email, it was expected that all participants would have a reasonable level of computer skill including exposure to drawing editors. It turns out that this one person only learned enough computer skills to use word processors and email and nothing else. Though the person should have been rejected from the study right away, the experimenter was overly optimistic and thought that using a drawing editor was easy and that the person could be trained. No other participant was inexperienced this severely; however, all subsequent participants were asked before the experiment began whether he or she was familiar with drawing editors. Striking the inexperienced participant and the child participant from the results leaves four other participants who were recruited through the expected channels and had enough experience to try the system. Of these four participants, two were male and two were female.

7.2.2.2 Procedure

The procedure for the final user study was copied almost verbatim from the paper prototype study (see Section 7.1.2.2). The major differences (besides using a real computer) was that the tutorial was written out and made more formal (see Appendix D.6), and that the setting for the experiment took place in one the HCI Institute's usability labs instead of a conference room. Once again results were recorded on video. Since the experimenter had free hands this time, he was able to take notes while the participant was using the system. However, the basic session routine was unchanged from the earlier paper sessions: first a survey, followed by a tutorial, then the largest block of time working on the tasks, and finally finishing with a questionnaire.

The usability lab provided space and camera equipment as well as a quiet, controlled setting for the participants to work. Using the lab meant that sessions could be held in a fixed location and all the needed equipment could be left set up between participants. The lab setup was minimal. The computer was placed in the middle of a table. The computer screen contained only Gamut and a small debugging window that was only partly visible. To the left and right of the computer was extra table space. The right side of the table was left empty except for a stack of blank paper and pens with which the participant could write. The participant also used that space to fill out the various forms. On the left side sat a TV monitor with which the experimenter could see what was being recorded on the video camera. The experimenter sat facing the TV monitor and took notes while the participant used Gamut. The experimenter and participant sat about an arm's length apart so the experimenter could talk to the participant and answer questions. The camera was set behind the participant and off to the side to get a clear shot of the computer screen. The camera only recorded the computer screen and the participant's voice. (Although one participant leaned so far forward that the camera inadvertently recorded his face as well.) Figure 7.6 shows a diagram of the arrangement.

The session began by having the participant fill out the permission form and a short survey (see Appendix D.1 and D.2). The purpose of the survey was to gauge how likely the participant would want to use a system like Gamut. As before with the paper prototype, the results of the survey had no correlation with how well the participant performed with Gamut.

Through the experience of the paper prototype study, it was clear that a tutorial would be necessary. This time the tutorial was written out in paper instructions (see Appendix D.6). By writing out the tutorial on paper, the instructions could be more comprehensive and more uniform for each person. The tutorial was interactive in that it had the participant try out various techniques using the computer as they read along. Each section contained a set of paragraphs first describing a technique and then instructions to perform some action using that technique. The instructions were designed to be slightly vague so that if the participant skipped reading the descriptive paragraphs, he or she would have to go back and read it again in order to know what to do. The tutorial began with the very basic editing techniques such as how to create and move objects and quickly

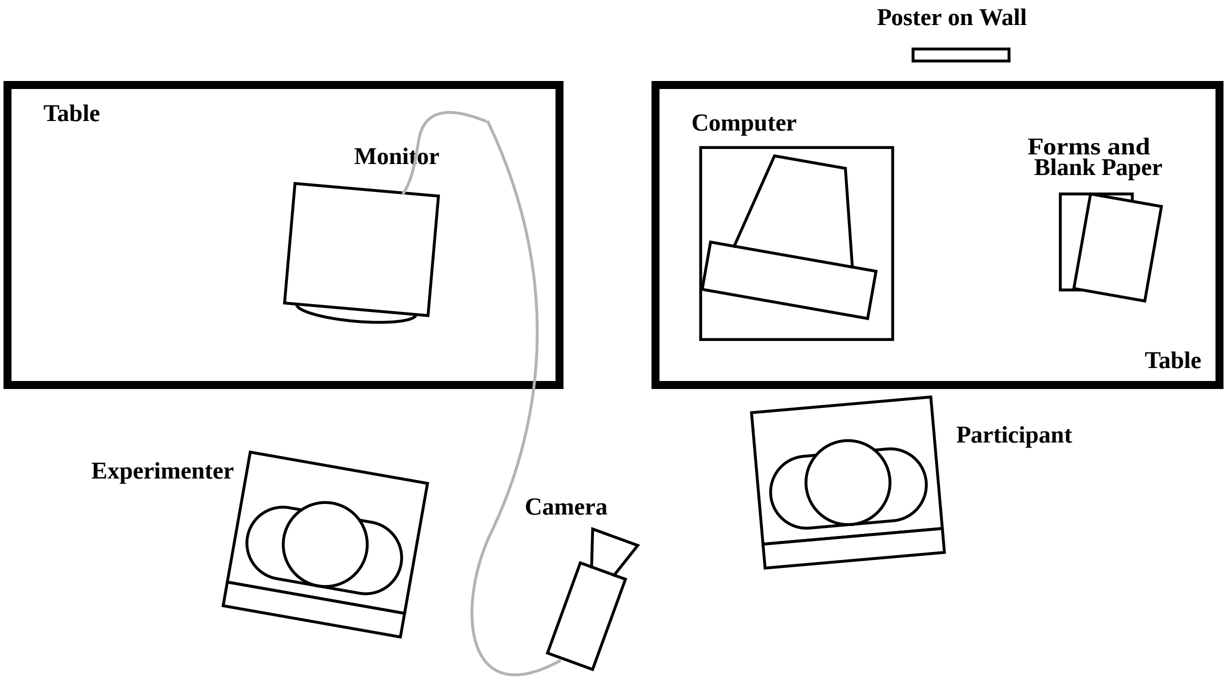


Figure 7.6: Table arrangement for the final usability study. The computer sat near the middle of the space. On the right was paper and pens for the participant to use and on the left was video equipment that the experimenter used to monitor the progress.

worked up to demonstrating behaviors. The tutorial had to be short so it could only show one or two examples of any given technique. The final version of the tutorial could be completed in about one hour.

Though all the participants successfully completed the tutorial, the first participant (as well as some pilot testers) read through it too quickly and did not remember any of its contents. To remedy this problem, later participants were required to give oral answers to a short set of review questions (see Appendix D.7). The review covered the basic nudges techniques to make sure that the participant at least knew how to demonstrate examples. Participants who had trouble answering the questions were asked to read sections of the tutorial again until they gave a reasonable answer. To help remind the participants after they began the task, there was also a small poster placed on the wall above the participant (see Figure 7.6) that listed the steps involved with making an example (see Appendix D.9).

After completing the tutorial and answering the review questions, the participant began to work on the tasks listed in the next section. Since the participants were still very new to the system even after finishing the tutorial, demonstrating the first task was usually time-consuming. The tutorial told the participant step by step everything he or she needed to do, but the first task does not. The experimenter was not allowed to intercede with the participant's progress or use the interface himself, but when the participant asked questions, the experimenter was allowed to answer. The experimenter tried not to answer high-level programming questions too directly. If the participant asked a direct question like, "would it work if I did this?" or "should I highlight this object?" the experimenter usually answered by saying, "that seems reasonable." If the participant became completely stuck, the experimenter would try to chat with the participant and try to get the person to state the problem and think of alternatives. However, the experimenter was not allowed to tell the participant what to do directly.

The one participant who was invited back for the second session received a second, short tutorial that only explained the new interaction techniques that were needed for the third task (see Appendix D.8). The short tutorial covered only player mouse events (see Section 4.3.5) and timers (see Section 4.3.3). The participant was reminded of the poster on the wall describing nudges, but was not asked to answer review questions. After the tutorial, the participant was asked to implement the third task.

When the session time elapsed, or the participant finished the tasks, the participant was given a questionnaire to fill out (see Appendix D.3). (The participant who completed the second session received the same questionnaire a second time.) This questionnaire was very similar to the one used in the paper prototype study. It asked which tasks were the easiest and hardest to demonstrate and what features in Gamut were the most difficult to use. The questionnaires were answered with well meaning and polite text but did not provide much information. In fact, the one participant who filled out the same questionnaire twice gave opposite answers on some questions.

7.2.2.3 Task Descriptions

The first task was called Safari (see Appendix D.10.1). A picture of Safari's screen is shown in Figure 7.7. Safari is a simple educational game that would teach a child about animals. Safari consists of two decks of cards. One contains a list of animals, and the other contains a list of questions about the animals like "Can it fly?" and "Does it have stripes?" The goal for the player is to push the Yes or No button to answer the question correctly. The smiley face at the bottom will turn to a frowny face if the player answers wrong. Also, the animal and question decks will both be shuffled to generate a new quiz whether the player answered right or wrong.

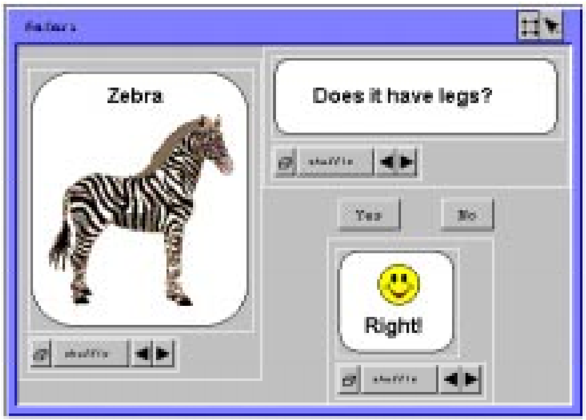


Figure 7.7: Task one in the final usability study was Safari. It is a simple educational game where the player answer yes and no to a series of questions about animals.

To demonstrate Safari, the developer must begin by drawing guide objects inside the cards. Although in principle, Gamut could learn a large decision table that correlated each animal card with each question card and create a complicated decision tree to solve the problem, Gamut would require more than forty examples to learn a table that large (actually forty is an estimate, I stopped demonstrating around thirty). On the other hand, if the developer draws guide objects into the decks that tells Gamut what the right answers are for each animal or question then the task is much easier and only takes about seven examples. For instance, the developer might create a duplicate of each animal’s name and put a copy of that name into the questions that are true for the given animal (see Figure 7.8). This way Gamut would only have to learn to compare a text string in both decks to learn the correct relationship. Several pilot testers used this solution; however, the actual participants tended to use more fanciful representations. For instance, one person used colored circles and rectangles to pair the questions with animals. In general, though the kinds of objects people used were different, the objects had the same purpose as the text in the figure.

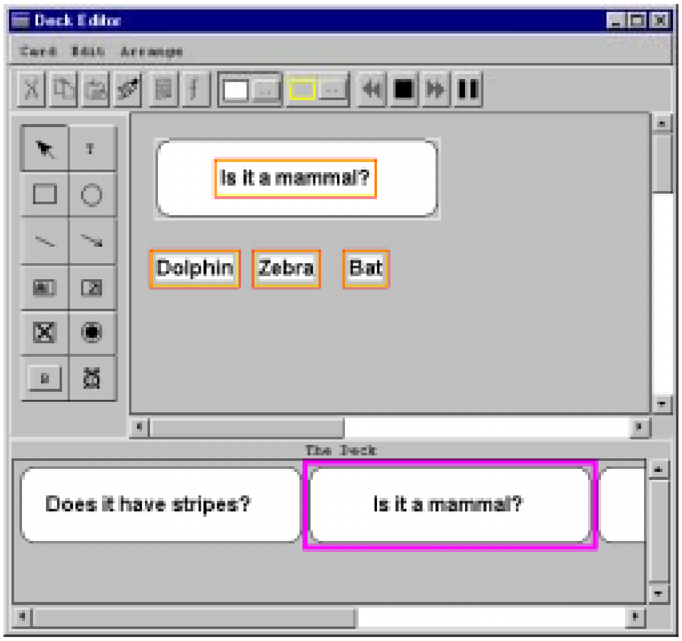


Figure 7.8: Here is how one might augment the question deck to contain the correct answers for each item in the animal deck. The extra objects are pre-highlighted so that the system can see them.

The second task was Pawn Race (see Appendix D.10.2) that was based on the Pawn Race game we tested in the paper prototype (see Section 7.1.2.3). In the new Pawn Race game shown in Figure 7.9, the participant demonstrated that two pieces moved around a rectangular board. However, to simplify the new version, the game board was given an obvious finishing square. Also, instead of rolling a pair of dice, the game came with a pre-constructed single die in the center. The die only had three sides with numbers one, two, and three. The game still required the participant to demonstrate that when one piece landed on another that the landed on piece would go back to start. When participants demonstrated this behavior in the paper Pawn Race game, they would often keep rolling the die until it randomly reached the right number. By making a three-sided die, the amount of variability was reduced so that positive outcomes would be more frequent. The participant still had to create a guide object to teach Gamut how to alternate between players.

The ending condition for the new Pawn Race was also different from the original. Instead of circling the board three times, a piece had to reach the final square. A piece would have to roll exactly the right number to reach the end. If a piece was near the end but rolled a die result higher than the number of spaces to the end, it would stay still. This was the normal behavior that Gamut

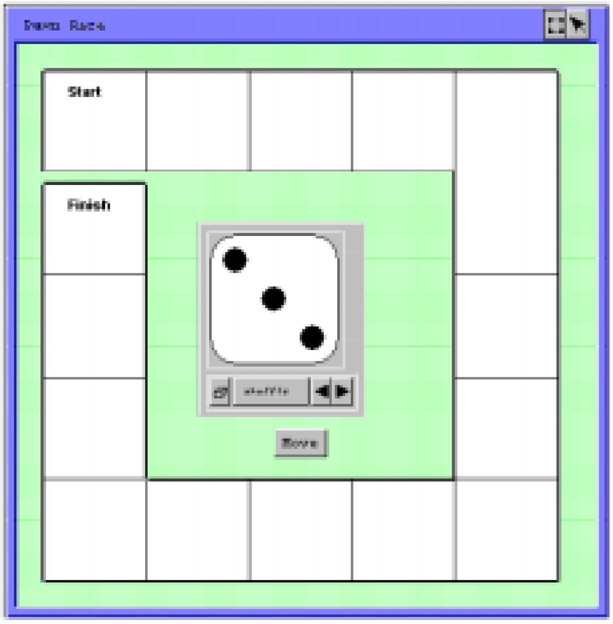


Figure 7.9: The second task in the final usability study was Pawn Race. It was very similar to the Pawn Race task in the paper prototype study.

learns when it learns to move an object a variable number of jumps; thus the participant would not have to do anything special to cause it to occur.

The third task was called G-bert which is played similarly to the arcade video game Q*bert produced by Gottlieb [35]. In G-bert, the player plays a character who lives in a sort of pyramid (see Figure 7.10). In the original Q*bert game, the character would jump on each cube of the pyramid to change the cube's top color. When all the top colors were the same, the player would win and move to the next level. In G-bert, a set of small yellow rectangles were used as markers and placed on top of each cube. G-bert has to jump to each cube to collect all the markers to win. G-bert's enemy is a bouncing ball that falls down the pyramid in a random direction.

The participant's task in G-bert is not really more difficult than what is required by Pawn Race, but there is certainly more behavior that needs to be demonstrated. G-bert requires the participant to demonstrate the behavior of the player's character using the mouse icons. It also requires examples that involve deleting objects from the screen when G-bert lands on a marker. To make the monster move randomly on its own, the participant has to use a deck to supply randomness and a timer to make the ball move. When the ball struck G-bert, the participant had to demonstrate that the character moved to the top of the pyramid and lost a life.

The hardest part of G-bert is demonstrating the winning condition that occurs when the player collects all the yellow markers. The participant chose to create a counter for the markers and used it in a similar way as the lives counter. When Gamut questioned why the "You Win!" sign was moved onto the board, the participant could highlight the counter and show that it was zero. Another, more subtle solution involves the participant highlighting a marker and the frame window together while there are still markers on the screen. This would tell Gamut that the reason the player does *not win* is when there is a marker on the screen.

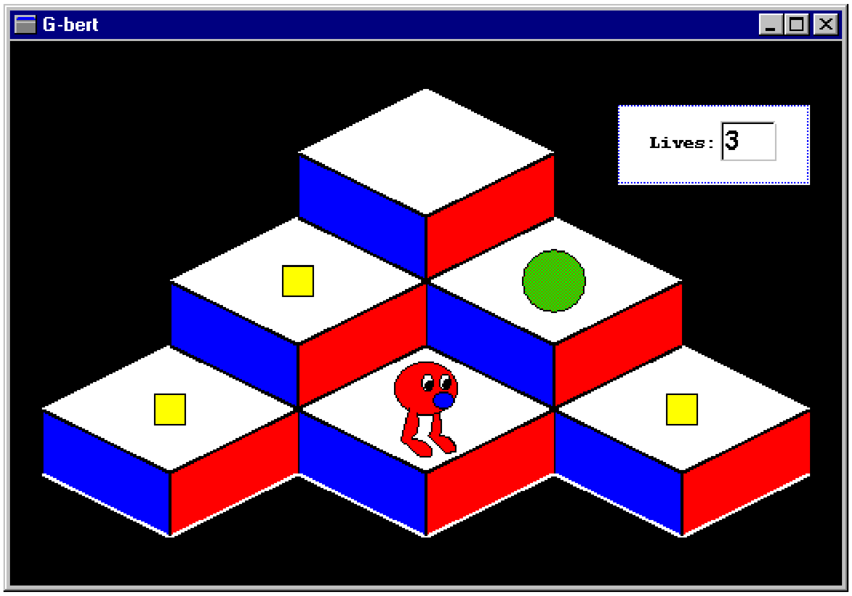


Figure 7.10: The last task in the final usability study was G-bert. The player's character jumps around collecting the little square markers while a ball falls randomly down from the top.

7.2.3 Pilot Testing Experience

The purpose of testing pilot subjects is to refine elements of the study in order to make improvements. In Gamut's case, the area that needed the most improvement was the system's own stability. Gamut's code had never before been stressed to this level and many bugs surfaced quickly. The majority of the pilot testing time was spent stabilizing Gamut's inferencing algorithms and implementing internal consistency checks at the higher levels of Gamut's code. The consistency checks helped prevent Gamut from crashing and made it print helpful error messages instead.

Besides fixing bugs, the pilot study also showed that the time needed to complete all three tasks was too long. Volunteers for the pilot study usually only attempted one task at a time and came back another day to attempt another task. The pilot volunteers tended to spend about an hour on each of the two simpler tasks as well as the tutorial and about two hours on the third task. As a result, the session was split in two giving the third task a three-hour session to itself. Since some of the techniques taught in the tutorial were only needed for the third task, the tutorial was likewise split in two. In all, there were fifteen pilot testing sessions involving eight different people.

7.2.3.1 Highlighting Ghosts Automatically

While Gamut was being made more stable, new heuristics were added that countered some of the inferencing problems that occurred in the pilot study. The most common problem concerned developers ignoring or otherwise not highlighting ghost objects.

It is not entirely clear why the pilot volunteers never seemed to want to highlight ghost objects. For instance, in the Safari task, the demonstrated response to pushing the Yes button is that both the animal and question decks get shuffled and the face deck shows the correct face. The card that the face deck shows depends on the values of the animal and question decks. However the animal and question deck get shuffled so that they no longer show the values that actually matter. It is the *ghosts* of the two decks that show their original configuration and still show the cards that have to be matched to choose the right face. However, the pilot volunteers rarely highlighted the ghosts of the decks and instead chose to highlight the actual decks.

When Gamut's inferencing algorithm creates code for the predicate of a condition, it constructs the code using the properties of the objects the developer highlights (see Section 6.6.3). The more useless objects the developer highlights, the more useless information the system will have to test and reject before it asks the developer for more information. Decks of cards contain a great deal of information, so when a developer incorrectly highlights a deck, it takes the system a long time to realize that the highlighted objects are not useful. Furthermore, when the system asks for more information, the developers do not seem to be any more likely to highlight the ghost decks.

To resolve this problem, a new heuristic was added to Gamut that would automatically highlight the ghost of any regular object the developer highlighted. Thus, the developer could highlight the shuffled decks in the Safari example and the system would still look at the ghosts of the decks to infer the correct relationship. This new heuristic required Gamut to handle other issues as well since the code created by automatically highlighted objects often needed to be kept separate from other code so that it would not interfere, as discussed in Section 6.5.4. Automatically highlighting ghosts also helped in the actual user study and allowed Gamut to infer the correct behavior in some situations where the participant was not highlighting the correct objects.

7.2.4 Participant Experience

Of the four user study participants, three were able to complete both the Safari and Pawn Race tasks. The one participant who tried the G-bert task was also able to complete it. This suggests that Gamut's techniques are reasonable and effective for enabling nonprogrammers to build applications and it suggests that programming-by-demonstration might be a useful programming metaphor in general if it is properly applied. Although the participants generally performed well, they did have some problems. This section will show the techniques the participants used successfully to accomplish the tasks. It then follows with the problems discovered in order to speculate about their causes so perhaps this information will be useful to improve future systems. Appendix E contains the participants' answers to the survey and questionnaire as well as screen shots of the applications that the participants created.

7.2.4.1 Table of Results

Each participant learned a different number of Gamut's techniques in order to accomplish the tasks. Since Gamut's techniques are often redundant the participants did not have to learn every technique in order to be successful. In fact, it was common for participants to only learn just enough techniques to be able to accomplish the tasks. As the participants became more comfortable, they might choose to learn another technique but this was uncommon. Table 7.1 shows which techniques each of the participants demonstrated expertise with during the course of demonstration. The definition for each category is listed below.

Do Something: The participants used Do Something to augment behaviors. All participants used Do Something to create a behavior for the first example, but only some used Do Something when they wanted to add a new feature to an existing behavior.

techniques	Participant 1	Participant 2	Participant 3	Participant 4
Do Something	X		X	X
Stop That		X		X
Highlighting Widgets		X	X	X
Highlighting Ghosts			X	X
Using Replace		X		X
Creating Guide Objects		X	X	
Time Controls		X		
Cards and Decks		X	X	
Player Mouse Icons*		X		
Timers*		X		

Table 7.1: Different participants gained expertise with different sets of Gamut's techniques. All participants except the first were able to successfully complete the tasks.
*Only participant two was given a task that required the Mouse Icons and Timers to complete.

Stop That: Participant two used Stop That to demonstrate most examples. She would select objects that she wanted to change regardless whether they were modified by the behavior or not. This allowed her to use Stop That as though it were Do Something, and Gamut successfully supported this style. Participant 2 did use Do Something occasionally, but it is not marked in the table because she did not use it very often.

Highlighting Widgets: All successful participants needed to highlight the right objects in order to complete the tasks. The one participant (number one) who could not learn this technique did not complete the tasks.

Highlighting Ghosts: Two subjects showed competence in highlighting ghost objects which was a surprising result. In the pilot test of this study, it was found that people did not like to highlight the ghost objects (see Section 7.2.3.1). However, two of the actual participants were comfortable with highlighting the ghosts of the decks in Safari.

Using Replace: Participant 2 also used Replace to defer highlighting. This result was also surprising because it was not taught in the tutorial. When the developer is asked to highlight objects, the intended objects are not always in a configuration where the salient property would be detectable by the system. For instance in Pawn Race, the participant had to teach the system to do something when two pieces overlapped. Sometimes the system asks its question when the pieces do not overlap. At these times, the participant correctly used the Replace button instead of Learn which essentially tells the system to ask again at a later time. See Section 4.2.3.2.1 for a discussion of this use for Replace.

Creating Guide Objects: Though all the participants had to create guide objects to make the game tasks work, some participants were more comfortable than others. The participants marked in the table were more able to create guide objects without help. Participant 4 was able to finish the tasks but does not get a mark in this category because she asked so many questions that it was unclear whether she had designed the guide objects herself or whether the experimenter gave away the needed answers.

Time Controls: One participant liked to use the history mechanism (see Section 4.4.2). She used the buttons to replay behaviors that she just demonstrated and to re-create marker objects that were deleted in the G-bert task.

Cards and Decks: Participants were checked in this category if they used cards or decks to create their own widgets. All participants had to use the deck controls to manipulate decks that were already created for the tasks, so participants were not checked if they only used the supplied decks. Participant 2 also had to use a deck to create random events which she did successfully and participant 3 used a deck to represent the player turn in Pawn Race.

Player Mouse Icons: Participant 2 was able to use the Mouse Icons technique to successfully demonstrate mouse input.

Timers: Participant two was also able to use a timer widget to make an object move automatically.

7.2.4.2 Gamut's Performance

Since Gamut is a prototype system, it would sometimes crash or exhibit bugs during the participants' sessions. Since some bugs would make it impossible for the participant to complete a task, the criteria for judging task completion had to be defined more loosely than if Gamut worked perfectly. A task was considered to be completed if the participant demonstrated examples that had worked in the past and should have been accepted by Gamut. In fact, the majority of examples were interpreted correctly by Gamut and on average, about one behavior of an entire task could not be successfully demonstrated because of a bug.

Of the four participants, only one was significantly hampered by bugs. This person had trouble because he chose not to use Stop That to correct the system's errant behaviors. Instead, he chose to move objects back to their original position manually. Since all other users up to that time used Stop That, Gamut had an undiscovered bug that did not allow the developer to manually undo behavior. After the experimenter realized why Gamut was failing, he explained the problem to the participant and asked him to use Stop That. From that point on, the participant was able to perform better.

The number of examples that the developer uses to demonstrate a behavior is a good measure for how easily the developer was able to demonstrate that behavior. Counting examples also suggests how long the developer will tolerate the system before giving up. However counting examples in a behavior is not very well defined. For instance in Pawn Race, there is only one behavior in the whole game, it is the behavior that occurs when the player pushes the Move button. However the ease with which each facet of that one behavior can be demonstrated is not equal. For instance, training a single piece to move is not the same as training the “You Win” sign to appear. So, instead of counting examples for the whole behavior, the experimenter counted the number of examples for each facet of a behavior where a facet is defined to correspond to one of the bulleted items in the task instructions (see Appendix D.10).

The number of examples that a participant used to demonstrate each behavior facet varied greatly between a minimum of three and a maximum of twelve. Three examples was the minimum required in any of the facets. Facets that were not completed often were presented with twenty or more examples before the participant gave up or time ran out. The behaviors that required the most examples were in the Safari task where the minimum number of examples was seven, but if the participant highlighted the wrong objects, the number would grow to fifteen or more.

Participants did not always demonstrate the behavior facet correctly on the first attempt. Sometimes the participant would go through a phase of trial and error testing to see what the system would accept. The trial and error period would continue until the demonstrated behavior became so convoluted that the system could no longer handle it and would crash. On the second try, the participant would have learned enough from the trial period such that the behavior’s facet could be demonstrated in five or less examples. This suggests that the participants used failed demonstrations to learn all the various aspects of a complicated behavior. After the system threw away the badly mangled test behavior by crashing, the participants had gained enough practice to teach the system the correct behavior more quickly. The system did not always have to crash for the participant to gain this advantage. Sometimes participants would realize that a behavior was mangled on their own and delete the behavior and start over resulting in a better demonstration the second time.

7.2.4.3 The Extra Session

The purpose of the extra session was to test some of Gamut’s more advanced techniques. Essentially, the extra session was a bonus task for participants who did well in the first session. However, the main goal of the study was to test the nudges techniques and see if nonprogrammers could use Gamut. This was accomplished sufficiently by the first session. The second session tested Gamut’s player input icons technique as well as timers. Since the second session was not as crucial as the first and since the first participant who participated in the session performed rather well, the second session was only attempted one time.

Basically, the second session showed that the participant did not have any trouble using the player input icons. Dropping the click icon into the window was not a problem. The participant also used the switch that made the actual mouse act directly in the window (which also turns off selection handles, see Section 4.3.5). The participant was not confused by the change in mode and also realized on her own that when she could not select an object that she had left the mouse switch active. The participant was similarly not confused that the mouse switch was independent of the interaction used alternately to select or push buttons.

Using the timer was only a little difficult for the participant. The participant originally did not want to create a timer and instead used a button to demonstrate how the circle moved down the pyramid. This same technique was used in the extra tutorial. Instead of demonstrating a new behavior, the extra tutorial trains a timer to use a behavior created in the previous tutorial. The participant was likely emulating this.

7.2.4.4 Guide Objects and Flailing

The most significant difficulty in using Gamut turned out to be the creation of guide objects. It seems that participants have difficulty understanding where and when guide objects should be created in order to make inferring behaviors possible. There seems to be an implicit assumption that guide objects are not necessary and that the developer should only have to demonstrate the outward appearance of the behavior for the system to understand. However, when the participants are explicitly told to create guide objects, they seem to be able to figure out what kinds of objects to create to act as guides.

The original task instructions told the participant to create guide objects. However in the original wording, the instructions to create guides sounded more like a suggestion than a command. As a result, the first participant chose to ignore the suggestion and proceeded to demonstrate without creating guide objects. This behavior was also seen with some of the pilot testers, but was not severe enough to rewrite the tasks. With the pilot testers, most managed to eventually go back and create guide objects after about ten minutes of fruitless demonstration. Unfortunately, the first participant also had difficulty remembering how to use nudges. This suggested that the participant was being saddled with too much to learn at once. To learn the nudges technique and to be expected to draw guide objects by choice is probably too much material to learn in a short one hour tutorial. As a result, the task instructions were rewritten to convert the suggestions about guide objects into full-fledged bulleted items. Thus, if the participant chose to ignore the item and did not create guide objects, the experimenter could ask the participant to finish that item before moving on to the next.

Once told to create guide objects, the participants were able to draw reasonable widgets and objects to represent the needed data. Sometimes the participants would draw guides that were too difficult for Gamut to recognize. For instance, one participant drew small arrows in the middle of the squares in the Pawn Race task to indicate which way the pieces should move (see Figure 7.11). The participant was told that Gamut was not intelligent enough to sight along arrows and eventually created a more suitable arrangement.

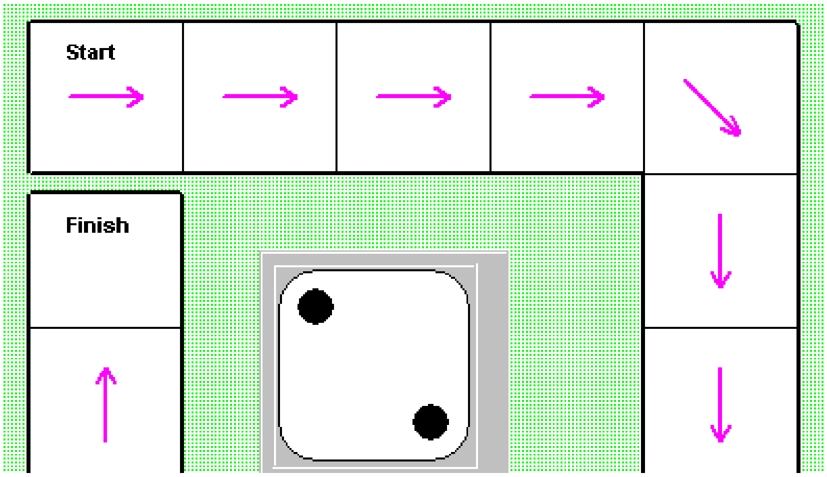


Figure 7.11: Some of the participants' guide object selections were too sophisticated for Gamut to recognize. In this example, the little arrows point in the direction the piece is supposed to move. For this to work, Gamut would have to sight along the arrow and be able to see the rectangle in that direction.

Another common trait was that the participants preferred to only use objects that were used in the tutorial. The participants seemed to assume that other widgets were off-limits or would behave poorly since they had never used them. So, although a simple checkbox would be sufficient to represent the current player's turn in the Pawn Race task, none of the participants ever used it. One solution was to create a deck of two cards. Each card was labelled with the colors of the pieces and set to show whose turn was next. The participant would then demonstrate switching the deck to make it alternate between players. Another participant wanted to use a number box to represent the player turn, but she did not know that the number box widget existed in Gamut. Since she did not know about number boxes, she did not know how to express what she wanted to the experimenter and basically froze and could not think of what to do. Eventually, the experimenter asked her to draw a picture on paper of what she would want to use to represent the player turn. She drew a number box which the experimenter immediately recognized and told her which palette item she wanted.

Even though the participants were generally able to create useful guide objects, they still were reluctant to highlight them when they demonstrated behavior with them. Thus, some participants continued to "flail" in the same way that participants in the paper prototype study did. In a highlighting flail, the participant gets confused by a question that the system asks and chooses to highlight random objects just to make the dialog go away. So, although the participants created an object to represent the player turn, some chose not to highlight it when the system asked why one piece had moved instead the other. It is not clear what causes this failure or what to do about it. Though there may seem to be an obvious object to highlight, people still miss it. One theory for this behavior is that the obvious object is too obvious and it would belittle the participant to have to point it out. In other words, since the participant had just created something to represent the data, it should be obvious to the system that it should be used in the behavior. And since it is so obvious, the system must be asking about some other mysterious relationship about which the participant can only guess. The fact that the system will accept almost anything as an answer is not helpful in this situation. It may be necessary for Gamut to judge the highlights of beginner developers to see whether the highlighted object is used elsewhere in the behavior. It could also perform more expensive checks to see if the highlighted objects in question have ever changed. Flailing might also be a transient behavior that would lessen as the developer learns to use the system better.

7.2.4.5 Problems With Nudges

In general, the participants who performed well in the tasks used the nudges technique to demonstrate behavior effectively. Of course, the one participant who did not perform well was the one who had the most trouble with the nudges technique. However, all participants did have at least some trouble. Some of the problems could be attributed to a lack of familiarity with the system, but most are probably due to a lack of feedback that would have told participants when they were not demonstrating what they thought they were.

The most common problem occurred when the participant demonstrated a behavior for the wrong event. In the nudges technique, the last event the developer supplies is assumed to be the triggering event for the behavior. Participants seemed to assume that Gamut could somehow infer the correct event when they pushed Do Something even if they never touched the object that would cause the event. For instance, in the Safari task, the participant might look at the state of the two decks, push Do Something and change the face deck to correspond to the condition if the player had pushed the Yes button. Without an event, Gamut would attach the behavior to the last editing command the participant performed. This would of course produce bizarre consequences. For instance, if the participant demonstrated a behavior just after moving an object, the behavior would run anytime the participant moved an object, but not when the participant pushed the button that was supposed to activate the behavior. Because the participant did not know that the behavior was attached to the movement event, the system would appear to be arbitrarily arranging objects at random moments.

A similar problem occurred when participants would perform an editing action just after performing the event. For instance, one participant would push the Move button in the Pawn Race task to train its behavior. The system would move the piece incorrectly because the behavior was not yet fully trained. The participant would then move the piece back to its starting point and push the Stop That button. The last event was moving the piece and not pushing the button and so the movement event would receive the new example (which moved the piece yet again). It is likely that the participant did not realize that moving objects could be considered events and thought that moving the piece back was the necessary first step in making Gamut stop misbehaving.

One final way that participants would demonstrate the wrong event was when they had too many buttons to choose between. For instance in Pawn Race, the dice deck has the usual set of four button controls. Sometimes the participant would push the Shuffle button on the deck as the event instead of pushing the Move button. Since the shuffle button is a different event source than the Move button, Gamut would treat the examples sent to each button separately. However, the participant would think that he or she was sending multiple examples to the same event. As a result, the participant would wonder why the system seemed to ignore examples that were sent to the wrong button.

It could be that Gamut is too flexible in what it allows as events. People seem to assume that only events that they want will be picked up by the system and it will ignore the same things that they ignore. For instance, maybe Gamut should assume that movement events are not events at all and only allow movement to be used as an editing command. Similarly, it might be better if the buttons on decks and timers were not considered events and not allowed to be given behaviors. Another option would have the developer note which event causing widgets he or she wants to demonstrate so that only those can be modified. For instance, developers might attach a “behavior object” to a widget or they might open up a widget’s property sheet and open up the behaviors that they want to modify. This way the developer could only demonstrate behavior for active widgets and all other editing commands would be ignored.

Another possible reason that participants failed to demonstrate to the right event is that Gamut produced insufficient feedback to tell the developer which event was being modified. Currently, Gamut draws an orange rectangle around the event’s widget. For instance, in Pawn Race, the Move button should always be marked. However, when a participant moved a piece before pushing a nudge button (thereby using the movement command as the event), the piece would be marked with the orange box. None of the participants noticed the orange box. It is also unclear if a textual dialog would be appropriate since users tend not to read dialog boxes very carefully. This is discussed more fully in the Future Work chapter (see Section 8.1).

A less pressing problem occurred when participants forgot to select the object they wanted to stop before they pressed the Stop That button. Since this was a known problem, Gamut's dialog actually anticipates this error and offers help to those who forget. It becomes a larger problem, though, when the participant had some other object selected when he or she pressed Stop That. In these cases, Gamut would not undo the actions on the desired object (because it was not selected). To the participant, Gamut would seem unresponsive and the participant would sometimes become confused. Gamut should probably try to detect cases where the developers select objects that are not modified by a behavior when they press Stop That. This way Gamut could provide a helpful message that might ease the confusion.

7.2.4.6 Problems Caused by Carelessness

One of the larger problems is that programming-by-demonstration seems to encourage a degree of carelessness. Since the system sometimes seems to guess the right behavior even though the developer did not understand what was happening, the developer might be lulled into a false sense of security that anything he or she does will be interpreted correctly.

A common careless behavior occurred when a participant was augmenting an existing behavior. The error occurs when the developer forgets to modify the demonstrated application's state correctly in Response mode. For instance in the Pawn Race task, when the participant demonstrated that one piece was supposed to stop moving and the other piece was to move, the participant would sometimes forget to move the other piece. Usually, the participant would not know that a mistake occurred and if the system were to present a dialog that mentioned the mistaken situation, the participant would get confused and be likely to flail. Participants were unlikely to be able to correct an error caused by carelessness and the most likely result would be a behavior that was badly mangled due to the flailing. Once a behavior went out of control, the participant had to start over again, though on the second try, the participant was much less likely to make the mistake again. Apparently, after failing to succeed on the first try, the participant would become more careful and be able to keep track of the application's state better.

Other careless behavior occurred because Gamut does not provide feedback to show when objects are properly aligned graphically. Since this is such a well-studied problem in other systems, it will not be discussed at length. However, sometimes Gamut would respond poorly because participants could not tell that pieces were not aligned with other objects and other similar situations.

One other careless behavior concerned coloring of on-screen guide objects. Some of the participants did not seem to care that the player could see the guide objects they drew in the application. For instance, when drawing objects in the cards in the Safari task, one participant drew the objects in the on-screen area so that she could see them herself. When asked what the player would see when the game is played, she related that she did not care what the player would see at the time and she could fix up the objects later. This also occurred in the Pawn Race task where participants would often not color the path of arrows in a guide object color. If they were asked about the player's view, they would either respond that they did not care or they would immediately begin to recolor the guide objects.

7.2.4.7 Confusing the Developer's Role with the Player

When some participants began the first task, Safari, they had difficulty distinguishing their own role as the game producer with the role of being a game player. This particular confusion seemed to be transient, though, as it did not seem to affect people attempting the next task. Some participants seemed to think that in order to demonstrate how the game worked, they needed to be able to play the game. For instance, when the participant was demonstrating the Yes button, if the answer was correct and should produce the smiley face then the participant was fine. However, if the answer was incorrect and the system needed to produce the frowny face, then the participant felt a compelling urge that he or she needed to push the No button.

One participant was especially vexed by this confusion. The participant insisted that in order to demonstrate the Yes button, he had to push the No button, but since the No button was not demonstrated yet, he could not do what he wanted. Of course, if the No button did work, it would have shown the smiley face and not the frowny face which was the opposite of what the game should be doing. Other participants were not quite as badly confused though some pushed the No button instead of the Yes button when the decks showed an incorrect question and animal pair as if to show the system that he or she knew the right answer.

7.2.4.8 Selecting Versus Pushing a Button

The Interaction Techniques chapter discusses the “use-mention” problem and how it affected the interaction used either to select or operate upon widgets (see Section 4.1.3.1). Specifically, the developer can select widgets by clicking on their edge or clicking in a non-operational area. The user study showed that this was an acceptable solution.

The most significant finding was that none of the participants were confused that a button could be both selected and pushed without changing a mode. In cases where there were two or more buttons, the participants were not surprised that one button could be selected while the other pushed. The most common difficulty the participants encountered was that sometimes they would accidently select a button when they wanted to push it and *vice versa*. The most difficult technique to master was making a button unselected if somehow it was accidentally selected. To unselect an object in Gamut, one must either click on a vacant area of the background or select a different object. This was not clear to one of the participants. Though all participants accidentally selected a button at one time or another, none of them were confused by what occurred when they made this mistake. They immediately saw that they selected the button and that they had to unselect it to do what they wanted. The rate at which this mistake occurred was not high enough to cause the participants to become noticeably frustrated.

7.3 Summary

The two user studies showed that Gamut’s interaction techniques work fairly well. In both studies, the majority of the participants were able to make significant progress and successfully completed the tasks. This suggests that programming-by-demonstration is a good technique for nonprogrammers to create software and that Gamut’s techniques are a reasonable mechanism for performing programming-by-demonstration. The problems uncovered in the studies show that Gamut could be improved. Of particular importance are the problems that allowed the developers to make careless mistakes without realizing it.

The studies showed that beginning developers are unlikely to create guide objects of their own accord. In the paper study, some participants would proceed to demonstrate and only create guide objects after the system failed to understand. This phenomenon also occurred during the pilot testing of the final study as well as during the first participant's session. This effect had to be countered with training and in the final study, so explicit instructions were added to the tasks telling the participants to create guide objects. Once the need to make guide objects became clear, though, participants were usually able to represent the needed data with appropriate objects.

Even when the demonstrated application had all the required guide objects available, participants would sometimes not want to highlight them in response to Gamut's questions. Instead, the participants sometimes became confused and highlighted some other object in order to skip Gamut's question. This effect was called *flailing* because participants would seem to highlight anything in sight when they did not understand what to do. Flailing occurred in both the paper study and in the final study. More effort will be needed to better detect or prevent flailing in order to make the highlighting technique more robust.

Overall, the nudges technique for demonstrating examples worked well. Some participants understood the technique almost immediately. Those that required more training usually understood the technique after a period of experimentation during the first task. However, one participant did not learn the technique at all, suggesting that nudges would probably not be universally accepted. Though the effects of the Do Something and Stop That were fairly well understood, some participants elected to use only one of the two nudges to demonstrate all examples. Basically, the participant would learn one of the two nudges and use it even when the situation would be better served by the other.

Better feedback might help solve problems brought on by developer carelessness. For instance, the current feedback for showing what object caused an event is not sufficient. Also, participants would sometimes forget to modify key objects in their examples and would sometimes move objects to the wrong place. The system does not know whether the demonstrated examples are correct or not so it has to assume that whatever the developer does is correct. Therefore, the system should have better ways to tell the developer the consequences of their examples so that they will be better able to prevent these kinds of errors.

A common theme in both the paper and final user study was that the participants performed much better after the instructional materials were improved. In the paper prototype, adding a tutorial greatly improved the participant's ability to draw guide objects and create behaviors. In the final study, adding the review questions forced the participants to consider what they learned in the tutorial and made them better able to use the system. This emphasizes that no computer interaction technique is truly natural. People have to be given time and training to learn how the system reacts so they can build their own mental model of how to use it.

Chapter 8: Future Work

This chapter describes a range of ideas and topics that go beyond the work in this thesis. Several of the ideas could be used to make Gamut easier to use and able to infer more complex behaviors, others are broader and suggest potential directions future researchers could explore programming-by-demonstration. Many of the ideas are straightforward and, although potentially tedious, would not be difficult to create. Others are more complex and require extensive research before they could be made to work. In general, each section in this chapter will begin with the simplest ideas and progress towards issues that require more intense research. Some of the topics cover known deficiences in Gamut including the ways it provides feedback, how the internal code might be displayed, and how the editor might be expanded to cover more media types. Other topics include more general problems concerning inferencing, producing code, and interacting with a developer.

8.1 Additional Interaction Techniques

Most of the work in creating Gamut was directed towards making the developer's input facilities easy to use and making the inferencing algorithms sufficiently intelligent. Not nearly as much effort was spent in presenting feedback to developers to assist them in resolving conflicts and problems. Clearly, if the developers are to receive better help to fix their problems, Gamut's feedback methods should be improved. Of the techniques listed below, most would be basic extensions to the system. However, generating better question dialogs would probably require more research.

8.1.1 Implementing Lost Features

Over the course of implementing Gamut, many ideas and features had to be discarded due to the lack of time. The majority of the "lost" techniques involve new interaction techniques that would make some aspect of demonstrating an application easier. Most were lost because they were not quite useful enough to warrant the time needed to implement them. Others were lost because their purpose was usurped by another interaction technique that was more useful. This section presents interaction techniques that were proposed or partially investigated but were not included in the current Gamut implementation.

8.1.1.1 Do Something Guessing

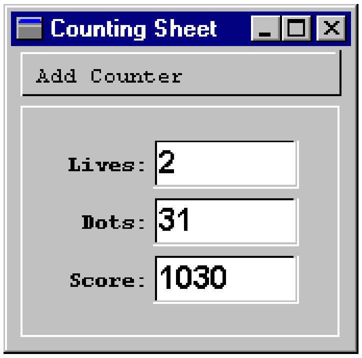
Gamut's Do Something nudge was originally meant to be a more active participant in the demonstration process. The developer would select objects and tell the computer to "do something" to those objects in a way similar to how the developer selects objects to tell them to "stop that." The system would search previously demonstrated behavior and look for a suitable set of actions that could be applied to the selected objects. This set of actions would form the system's "guess" and would be applied immediately to the selected objects. Currently, the system just waits for the developer to demonstrate what to do when he or she presses Do Something.

The criteria for guessing actions would be based on heuristics. The simplest heuristic would be to search for actions in an unused branch of the behavior that refer directly to the selected objects. Other, more complicated, heuristics would involve comparing selected objects to previously recorded actions using some metric. If the objects were deemed similar to objects used in an action, then that action would be applied as the guess. For example, if the developer trains a monster to follow a complicated path, the developer may create a second copy of the monster intending to make it move the same way. When the developer selects the new monster and presses Do Something, the system might compare the new monster object with the old monster object and see that they are similar. This would cause the system to apply the original monster's behavior to the new monster as a guess. If the developer accepts the system's guess, then there would be no need for the developer to edit the new monster manually.

Since guessing would be based on heuristics, the system could make mistakes. The developer would either manually edit the scene to correct the behavior, or would have an option to have the system guess again. When the system cannot find a suitable match for the object, or when it runs out of candidate actions, the system would give up and ask the developer to show what behavior was actually intended. If the developer does not want the system to make a guess, he or she would only have to push the Do Something button without selecting any objects. With no selected objects, the system would simply enter Response mode as it does currently.

8.1.1.2 The Counting Sheet

Many game applications rely on numbers for many purposes. For instance, in Monopoly, numbers are used to buy and sell property, keep track of the score, and move pieces around the board. The purpose of the proposed "counting sheet" dialog shown in Figure 8.1 was to simplify keeping track of numbers. It acted like an application level property sheet with each row representing a numeric value (usually an integer).



Lives: 2
Dots: 31
Score: 1030

Figure 8.1: The counting sheet was a dialog area designed to track and infer numbers. The idea was later abandoned because other, simpler widgets could perform its function.

The most intriguing use for the counting sheet would be to dynamically count a set of objects in the interface. For instance, consider the game of Pacman. In Pacman, the player clears dots out of a maze. When all the dots are gone, the player advances to the next board. The counting sheet would provide a convenient way to convert the number of dots into a numeric value. The developer would first select the remaining dots in the scene and use a menu command on the counting sheet to create a new row. The developer might name the row, "Dots." The counting sheet from that point forward would keep track of the number of dots on the screen automatically. Then, when the number of dots reached zero, the developer would be able to highlight the counting sheet entry (as a hint) to tell the system why the player advances to the next board.

The counting sheet could also be used to count objects more difficult to describe. To make the counting sheet learn complicated constraints, the developer would have to watch how the value changes and train the system using nudges when the value is wrong.

Counting sheets have some problems, though. First, when the developer changes the number in a counting sheet row, the system does not know which objects the developer intends the system to count. The system might initiate a dialog with the developer asking for the new set of objects to be selected somehow. If the desired expression is based on objects besides the ones being counted, then the developer would somehow have to highlight those objects as hints so the system would know to use them. However, the developer cannot currently highlight objects during normal editing, yet the counting sheet rows would probably be created in editing mode. Hence, if the new row needed hints, the developer would not have a way to give them.

As it turns out, there is little difference between the rows of the counting sheet and number box widgets. An offscreen number box can hold numbers in identically the same way as the counting sheet. The main difference is that number boxes do not update automatically each time a set of objects change. Instead, the widget reacts to events just like all other widgets. For instance, if the developer uses a number box to count the number of dots remaining in a Pacman game, the number box must be trained to decrease each time the Pacman eats a dot. If the developer changes the number of dots in edit mode, the number box would not be expected to update its number automatically. A number box needs to be trained to react to the events that affect the objects being counted. This is why a number box is not affected by hint highlighting problem. A number box does not try to maintain its count at all times so it never has to learn to perform actions during normal editing when there is no hint highlighting.

8.1.1.3 The Timeline

Handling time is another important aspect when building an application. Many tools are strongly based on timeline or "score sheet" models such as Director [53] and Pavlov [97]. The purpose of a timeline is to provide a graphical model for time so that the developer may visualize how events unfold as behaviors progress. Timelines are especially useful for designing long sequences of actions such as in an animation or a presentation.

Gamut's original design also included a timeline abstraction, but it had a different purpose. In Gamut, all actions demonstrated in an example are considered to be instantaneous (see Section 4.2.2). Actions occur as the response to only one event so there are no other cues to attach later actions in a sequence. However, sometimes the developer wants to use the sequence of actions to hold partial results.

Consider the example in Figure 8.2 in which a circle uses a vector to move to the right. In this case, the developer creates a line object, moves it to the right, and then deletes it. The circle is moved to the end of the line's last position. Since the position of the circle depends on an object that was both created and deleted, it creates a dilemma for Gamut. In a sense, the line does not even exist. When the behavior is executed, neither the developer or player would see a line object because it is immediately deleted. (Actually, Gamut would never even create the line because its Delete action would cancel out the Create action when Gamut simplifies the developer's example and neither would appear in the behavior.) However, since the line's position is used in the behavior, Gamut would need some way to let the developer manipulate it. This is where the timeline becomes useful. The timeline would record each edit made by the developer as an action in its list even when the edit was performed on the same object multiple times or if the object is eliminated before the end of the behavior. When a behavior is revised, Gamut might have to add actions to the timeline to show objects that a behavior uses internally but does not manipulate visually (such as objects that are both created and deleted). The developer could click on actions in the list which would cause ghost objects to appear that show the state the objects at the time of that edit. So, by clicking on the first move operation that was performed on the arrow line in Figure 8.2, the developer can view its intermediate state. This would allow Gamut to build a description for it.

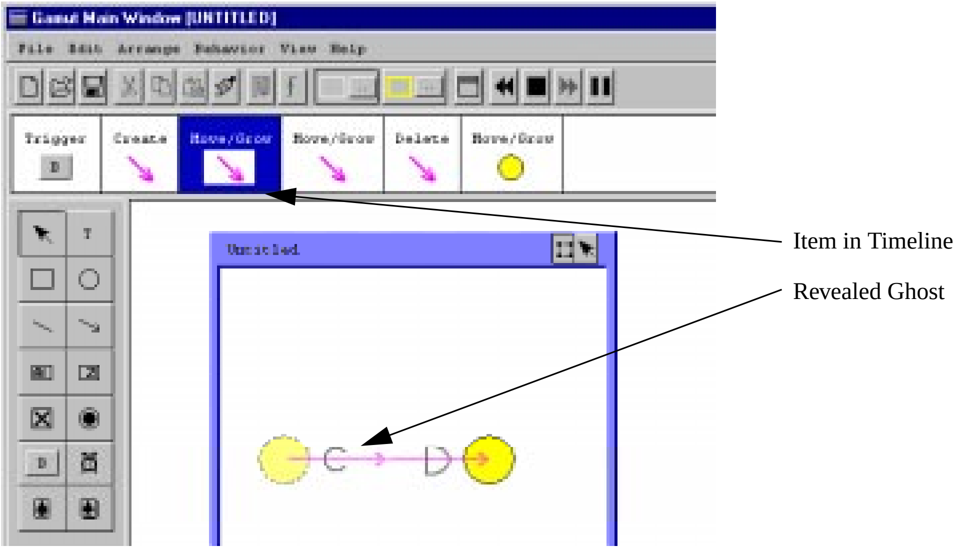


Figure 8.2: Here the developer uses the timeline (not implemented) to reach an intermediate state. The developer clicks on the item in the timeline when the vector was first moved to generate a ghost for the intermediate position.

The timeline would also be useful for referring to states that transpired before the current behavior began. For instance, a behavior might depend on a toggle switch being activated twice. Instead of counting how many times the switch is toggled in a separate variable, the developer might use the timeline to bring back ghost objects of the two times that the switch was active. Of course, actually inferring this kind of condition is very difficult. Highlighting a historical object could have many different meanings. It could mean the number of times the application reached that state, something about the value of the state, some property of the event that caused the state to occur, and many other things. In general, reaching back in history further than a single behavior seems quite difficult to infer.

Though implementing the timeline would still be a useful interaction technique for Gamut, the facilities that the timeline would provide are not actually mandatory. For instance, in the example above where a character follows a line twice, it would be equally possible to make the line twice as long. One advantage of the timeline is its feedback. The timeline would show what actions the system actually performed. If the list of actions does not concur with the developer's intentions, the developer has a resource for investigating why the system did what it did. Many kinds of feedback might be incorporated into the timeline to make debugging easier. For instance, the timeline could provide an entry point for showing the code the system produced for a given action.

8.1.1.4 Behavior Icons

Gamut currently has no mechanism to indicate whether a certain object will respond to a given behavior without actually executing the behavior. Originally, Gamut was to use "behavior icons" for this purpose. A behavior icon is a small marker that would be placed above objects that have a behavior defined on them. In Figure 8.3, there is a button that when pushed causes the counter alongside of it to increase by one. The figure shows how the button and the counter might be marked with icons. The button would be marked with an "event" icon ("E") to show that it is the instigator of a behavior, whereas, the counter would be marked with an "object" icon ("O") to show that it will be affected by a behavior. To reduce clutter, only event icons would be displayed initially. If an event icon is selected, the objects that the behavior affects would also be marked.

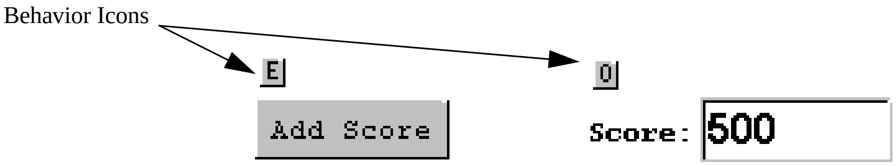


Figure 8.3: Here a button and number box are marked with "behavior icons" to show that they have had a behavior defined for them. The "E" marker indicates that the button's event is used for a behavior and the "O" marker shows that the number widget will be affected.

Besides providing feedback, the behavior icons would be manipulable. For instance, the event icon from one button may be copied and pasted to another button causing the second button to perform the same behavior. Another benefit of the behavior icons however would be manipulating the icons themselves. For instance, a behavior that affects one object could be replicated to affect two or more. For instance, the developer might use a stand-in object to demonstrate a behavior and then transfer the behavior to the actual object once it is ready. Manipulating the behavior icons can also reduce the number of examples the developer must demonstrate. For instance, a behavior that affects a set of objects will mark that set with behavior icons. The developer may see that the wrong objects are marked and manually change the marks to reflect the correct set.

One particularly advanced operation that behavior icons would allow is the demonstration of meta-behaviors, i.e. behaviors that modify behaviors. The icons would be available in Response mode so it is possible for the developer to demonstrate moving or copying a behavior as part of the response. A meta-behavior can often replace the need for modes. For instance, consider a game where only one monster is allowed to move at a time. Instead of using a guide object to mark the mobile monster, the developer could transfer the movement behavior from one monster to the next. In essence, the behavior icon would become a guide object.

Behavior icons would require more research to implement because manipulating the icons can be ambiguous and may require extra information from the developer. For instance, some behaviors can only be applied to some types of objects. A behavior that changes the color of a rectangle would not be applicable to a button because a button does not have a color property in Gamut. Similarly, if the behavior were applied to a line, there is still confusion because a rectangle has both a fill style and a line style whereas the line only has a line style. More difficult problems occur when modifying a set of objects. If the behavior is conditional, then changing its object icons may mean that the behavior should use a different branch or it may mean that the current branch should be modified.

Only some behavior editing is available in Gamut. For instance, Gamut allows behaviors to be deleted using a menu command (see Section 4.4.4). Likewise, when an object containing a behavior is copied along with the objects that the behavior affects, the new copies of the objects will perform the behavior independently from the original. However, other features are left for future work.

8.1.1.5 Mode Switches

Game applications usually have a large number of modes and conditions. For instance, the game of Monopoly has modes to indicate the current player's turn, whether a player is "in jail," the contents of two decks, and numerous other things. Since modes are so prevalent, the original design for Gamut included a special-purpose widget to represent modes called the "mode switch" which was adapted from Marquise [76]. Figure 8.4 shows how the mode switch might look. The key feature of the mode switch is that the system could use special heuristics to treat the mode switch as a mode in the application and not as a random piece of data. For instance, the system might infer that the mode switch causes a change in behavior without requiring the developer to highlight it. Furthermore, the presence of the mode widget might encourage developers to think about modes and not to forget to represent these kinds of data.

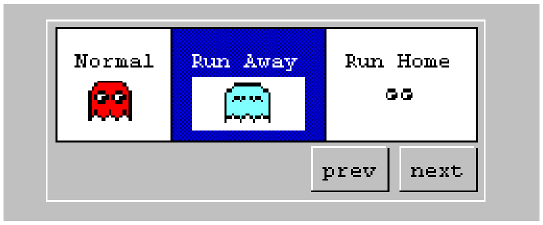


Figure 8.4: This is what the mode switch might look like if implemented. It could be used to represent an arbitrary number of modes. The mode is set by clicking on the one desired or clicking on the next or previous buttons.

The mode switch was abandoned because its purpose could be adequately served by other widgets such as check boxes and radio buttons. In the paper prototype study (see Section 7.1.3), developers were much more likely to use a number box than a mode indicator to represent the player's turn in a board game scenario. It is likely that the developer drew a strong relationship between the player number and the number in the counter and thus did not realize that the state would be better represented by a binary toggle switch. When the finished system was tested with a similar task (see Section 7.2.4.4), developers still tended to create and train a custom widget rather than use a standard checkbox.

8.1.2 Extending the Temporal Ghost Metaphor

The temporal ghost feedback is designed to show the developer the past states of modified objects. However, it may also be possible to use ghost objects to display other states as well. For instance, if an object is moved multiple times during a single example, the intermediate positions might be made visible with ghost objects. This would be possible if Gamut provides a timeline feature that would let the developer select which ghosts to display.

Another use would be to show future states that the developer wants to prevent. When the developer selects an object and presses Stop That, the actions performed on the selected object are undone. However, a ghost object might be left in the state where the object was before it was stopped. Suppose a character in a game is supposed to walk in a maze but is not allowed to walk through walls. If the character steps into a wall, developer selects the object and presses Stop That. The system could leave a ghost in the position where the object crossed the wall as shown in Figure 8.5. The developer would highlight the ghost and the wall together to explain that their overlapping is what caused the behavior to change. It turns out that behaviors like these can also be demonstrated using guide objects. For instance, a guide arrow pointing out from the character in the direction that the character moves would cross a wall before the character and thus could be highlighted in a similar way. The ghost object method, though, might be more understandable for some developers.

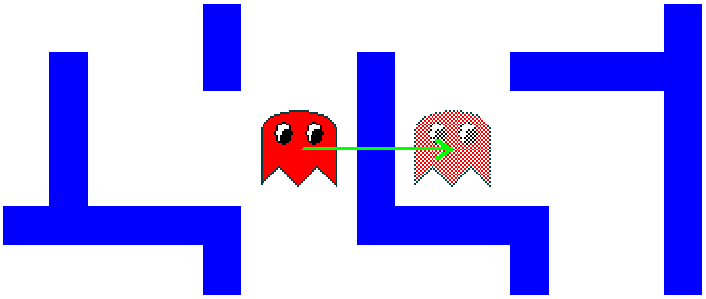


Figure 8.5: The system shows a ghost object of the character crossing the wall. The ghost here represents a future state the developer prevented using Stop That.

8.1.3 Modifying the Look of Decks

Gamut provides the deck widget so that developers can create lists of objects for their applications. Decks can provide randomness and can be used to iterate through their list in order (see Section 4.3.2). Currently, the visual look of the deck widget cannot be altered. It could be useful if decks had a variety of graphical configurations as in Figure 8.6 besides only showing the top item. For instance, it might be nice to fan out several objects in the deck so that multiple objects were visible at once. This way a deck could be used to represent multiple graphical objects like a player’s hand of cards. Another view might be as a scrolling list of objects such as in a menu or list of choices. Allowing the developer to modify the look of decks would save the developer from having to use different widgets in order to achieve those looks for the player.

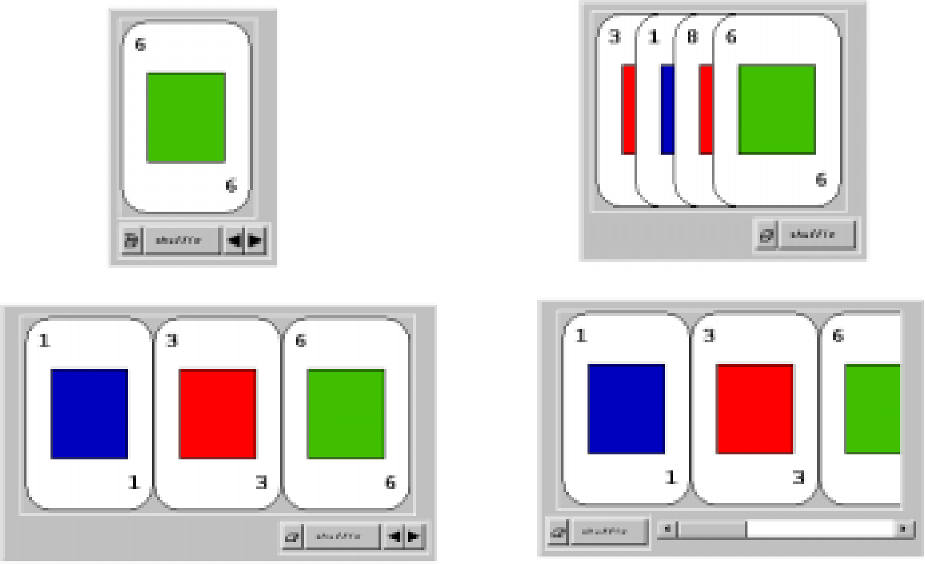


Figure 8.6: Alternative widget designs for decks. Scanning from left to right and top to bottom: the current deck design showing the top item, a fan of multiple items, a row of items, and a scrolling list.

8.1.4 More Graphical Feedback

Sometimes participants in Gamut's usability study would do the wrong thing in their example and not realize it (see Section 7.2.4.6). For instance, if they wanted to center a game piece over the end point of an arrow line, they would sometimes miss. Gamut needs to display better feedback to relate the various constraints and alignments that occur frequently in applications.

There has already been considerable research in displaying graphical constraints and making it easier to align objects in a direct manipulation editor. For instance, Bier's work in "snap-dragging" [6] shows how to align objects using a gravity-like metaphor. Karsenty's system called Rockit [46] used a variety of graphical and sound techniques to tell users where objects are located relative to one another. These techniques would be useful in Gamut, and they would be straightforward to add. The main reason Gamut does not implement these techniques already is that some of the techniques require serious programming effort. For instance, programming the sound cues in Landay's system is quite problematic given the lack of standard sound libraries and the difficulty of producing sounds in general.

A more pressing feedback issue is having something that tells developers what Gamut sees in their examples. For instance, the system might use a timeline dialog to list a set of English sentences describing each constraint the system has inferred from the example. The real issue is to make the developer realize when they are demonstrating the wrong thing. For instance, the developer needs to notice when he or she needs to move an object but did not. Of course, if the display is too cluttered, then the developer is just as likely to make mistakes because the errors are hidden beneath all the feedback. Clearly, this is an area for future research.

8.1.5 Clarifying Questions

When the developer presents an example to Gamut, the system may find conflicts in how the new example relates to the currently inferred code. Gamut presents those conflicts as a series of questions that the developer answers by highlighting objects in the display (see Section 4.2.3.2). These question dialogs were among the last things implemented in Gamut and were not designed as thoroughly as other parts of the system. As a result, there are some improvements that would make the question dialogs better.

8.1.5.1 Referring to Objects in Questions

One of the problems with asking the developer questions is signifying the objects in the application to which the question is referring. Currently, the system draws red and blue boxes around the discussed objects, but this could be improved. For instance, replicas of the objects might be drawn in the dialog itself. The developer might also be given the option of naming objects so that the dialogs could refer to them by name. The text of the dialogs might also be improved using color. The system could use different colors to signify different parts of the question, highlighting the referenced objects and the situation that needs to be resolved. The main reason these kinds of display improvements were not implemented in Gamut was due in part to the absence of needed facilities in Amulet [74], the interface tool in which Gamut was developed. Amulet lacks a widget that can mix various media types such as pictures and text together. It also lacks a text widget that can draw multiple colors and fonts of text together. Implementing such a widget is difficult and time consuming so Gamut was constructed without these features.

8.1.5.2 Clarifying a Question

The questions the system asks can sometimes be confusing. The problem is that manner in which a question is posed can often influence the answer one will receive. Since the system does not know what the developer intends, the questions can be too high-level and more vague than the developer can understand. For instance, if the developer is training a game piece to follow a path of arrow lines but never bothers to highlight the arrows, the system might be able to ask the developer to “please highlight the path the red marked object is following” in order to elicit the proper highlighting. But the behavior of the piece may have nothing to do with following a path so the system has to ask a vaguer question, “what does the red marked object’s position depend on?” Since “depends on” is such a vague question, the developer may be just as likely to highlight other objects beside the path, assuming the path itself is too obvious to need highlighting.

It might be possible to make the question dialogs more understandable by using heuristics. Certain patterns in the application and behavior code might be interpreted to mean different situations and the questions would be changed to follow suit. Categorizing the various situations is not trivial and the algorithm for discriminating between situations would not likely be simple. It may be possible to use an AI algorithm like a Bayes network [15] to select forms of questions. A Bayesian algorithm can determine the probabilities of a set of situations in order to choose which one to apply. Such an algorithm might be ideal for selecting text for the question dialogs.

The question dialogs might also be integrated with a dialog for displaying Gamut’s internal language (see below). Though it would be improper to force the developer to read the code, if the developer could see where in the code the question applied, it might be easier to highlight appropriate objects. For instance, if the question referred to a constant object in a description, the developer might know that the context that describes the object needs to be highlighted. If the question refers to the predicate of a condition, the developer might know to highlight objects that pertain to the application's mode. The connection from the question dialog to the language display would have to be indirect in order not to force the developer to read code. For instance, if the language dialog is already visible, it might automatically jump to the code where the question applies and highlight the conflict.

8.2 Displaying the Language

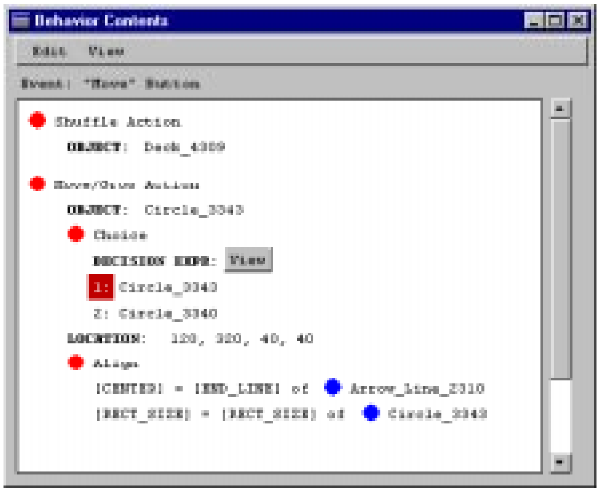
Gamut currently does not support a way for the developer to examine or edit inferred code. Gamut's internal language is difficult for a novice to understand for a host of reasons (see Section 9.5). As a result, it was never a priority to make a dialog for displaying the language. Furthermore, one of the tenets of this thesis is that it is possible to build applications entirely with demonstration. By not providing a code editor, this maxim could be explored in earnest.

However, sometimes seeing the code can be helpful. If the developer is allowed to see the code that the system generates from each example, it could provide several benefits. First, the developer might learn to read the code and be able to see bugs and errors that might not be obvious from simply watching how the system performs. Second, the developer might learn to provide examples in an order and depth that makes it easier for the system to infer. At the very least, the developer might be able to know when the system cannot infer something because it has no simple way to express it in its internal language. Finally, the developer might learn how to program so that some code might be entered by hand when it is easy to do so. The manner of presenting code to a novice programmer is an active area of research.

8.2.1 How To Display the Code

Gamut's behaviors have a hierarchical design. They are built out of actions that in turn contain descriptions that themselves contain other descriptions. Using an outline approach, the system could display a single behavior's code as the list of actions that occurred and place evaluated values in for the actions' parameters as in Figure 8.7. Parts of that behavior's code that were not executed need not be displayed right away to avoid showing the developer more than is needed. This way the code would look like a list of actions which is similar to how a macro looks in other systems. If the developer wants more information about one of the parameters, he or she would click on it and expand it into a textual description. Once again, the description's parameters would show evaluated values. With this approach, the developer need not be overwhelmed by the entirety of Gamut's code. Though the amount of code for a single behavior can be large, typically a behavior will only have three or four actions (see Section 6.4.1) making the size of the dialog seem small and manageable.

Of course, the developer may want to view other parts of the behavior's code that may not have recently been executed because they exist in the unused parts of conditional statements. For these items, the system can still display values for the parameters, but the developer will have to understand that those actions or descriptions did not occur this time. Unexecuted actions might be made a different color to distinguish them from executed actions. Also, the developer will likely want to search the code for references to various objects on the screen as well as connections where parts of the code are shared. Providing search mechanisms for values and objects would not be difficult to provide.



```
Event: "Move" Button

● Shuffle Action
    OBJECT: Deck_4309

● Move/Grow Action
    OBJECT: Circle_3343
    ● Choice
        DECISION EXPR: View
        1: Circle_3343
        2: Circle_3340
    LOCATION:  120, 320, 40, 40
    ● Align
        [CENTER] = [END_LINE] of ● Arrow_Line_2310
        [RECT_SIZE] = [RECT_SIZE] of ● Circle_3343
```

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 storing 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.

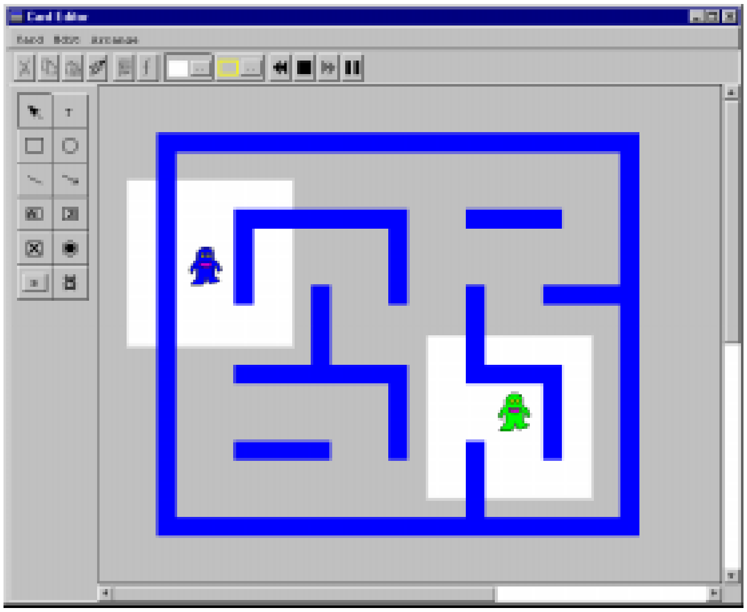


Figure 8.8: The developer uses multiple viewports in a single card to create a multiple-player game.

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, different 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 developer 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 studies 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

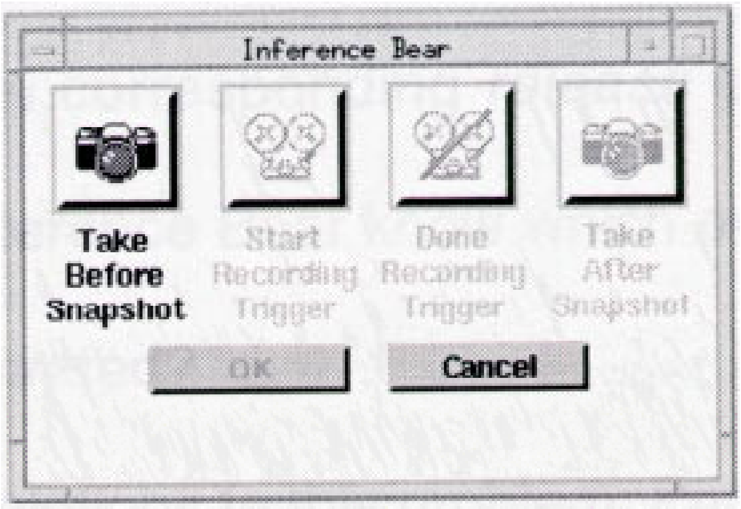
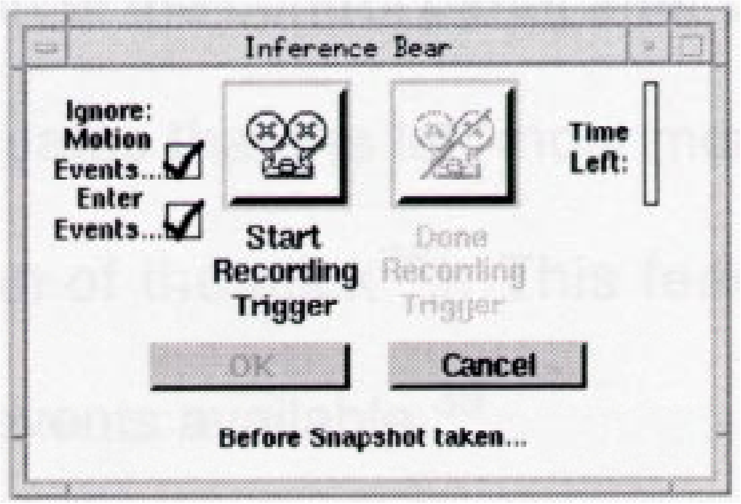


Figure 9.1: The augmented macro recorder dialog in Inference Bear [30] (reprinted with permission).

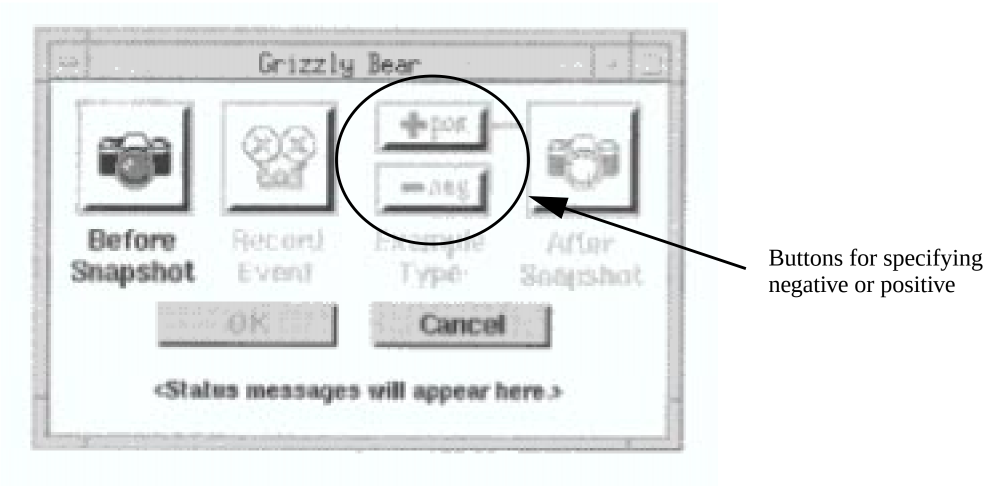


Before Snapshot taken...

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.



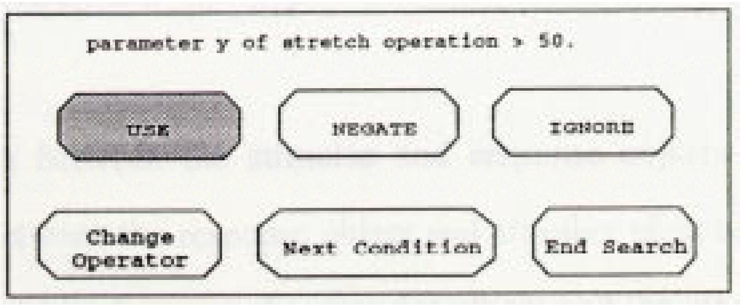
<Status messages will appear here.>

Buttons for specifying negative or positive

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.



parameter y of stretch operation > 50.

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 algorithms 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.

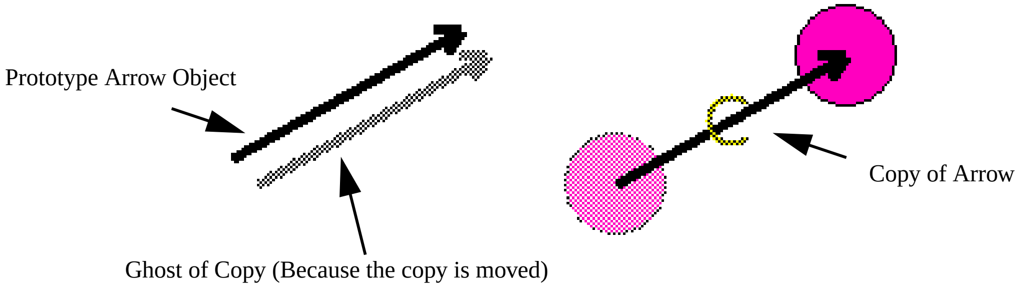


Figure 9.5: The developer trains a behavior that creates an arrow and moves a circle to its end point. The position of the circle thus depends on the arrow.

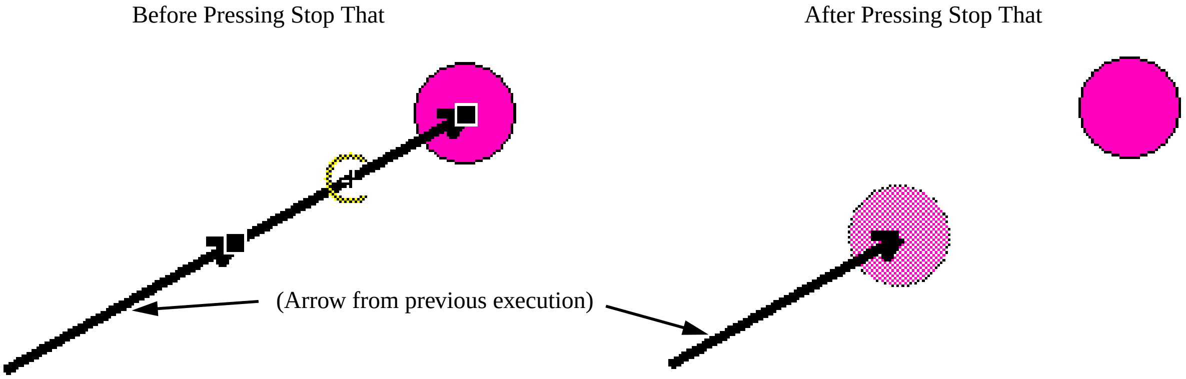


Figure 9.6: In a surprise demonstration, the developer tells the system to stop creating the arrow but continue to move the circle. The system then has to deal with the dependency some other way.

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.

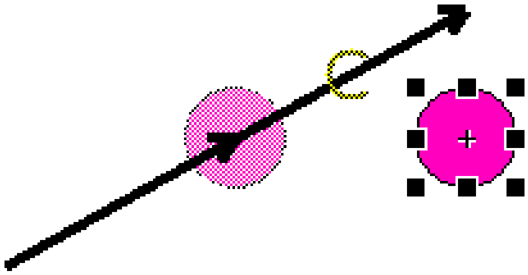


Figure 9.7: Instead of using Stop That, the developer changes where the circle moves. The system must consider the actions associated with the arrow because the circle’s action originally depended on the arrow.

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.

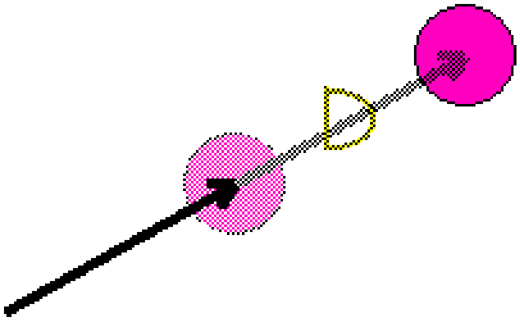


Figure 9.8: This time the developer selects the arrow and deletes it. The system learns to create an object, use its position, then delete it. However, since the arrow is never visible, it never needs to be created at all.

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

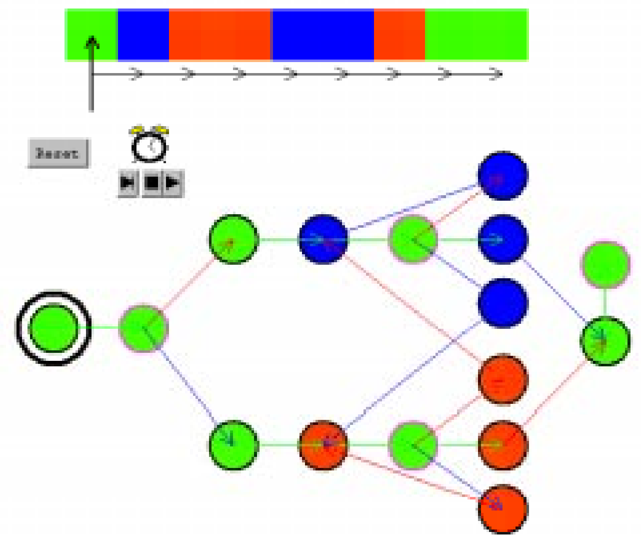


Figure 9.10: This Turing machine has been created completely using only demonstration. It contains a finite state machine at the bottom that controls the tape along the top of the display. The effect of the program is to reverse the colors of the tape from green to red and back again.

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 creates 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

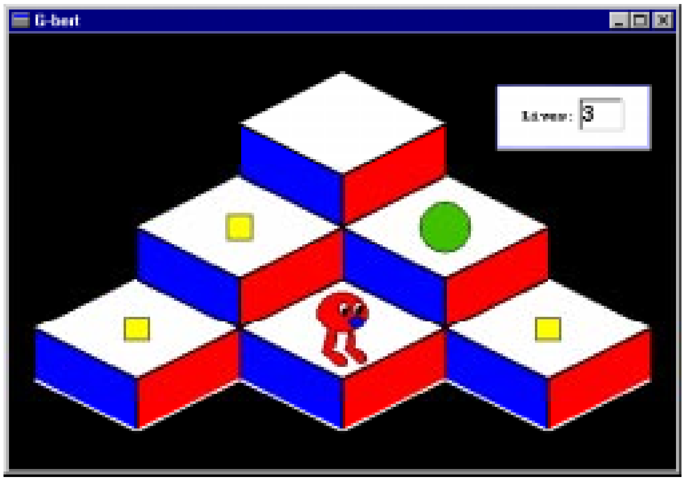
Gamut has been a successful project and has produced many interesting results. The result that I most hope to achieve, though, is to show that programming-by-demonstration is a feasible and useful approach for making software. This project has certainly shown that PBD allows nonprogrammers to create 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.



Lives: 3

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.

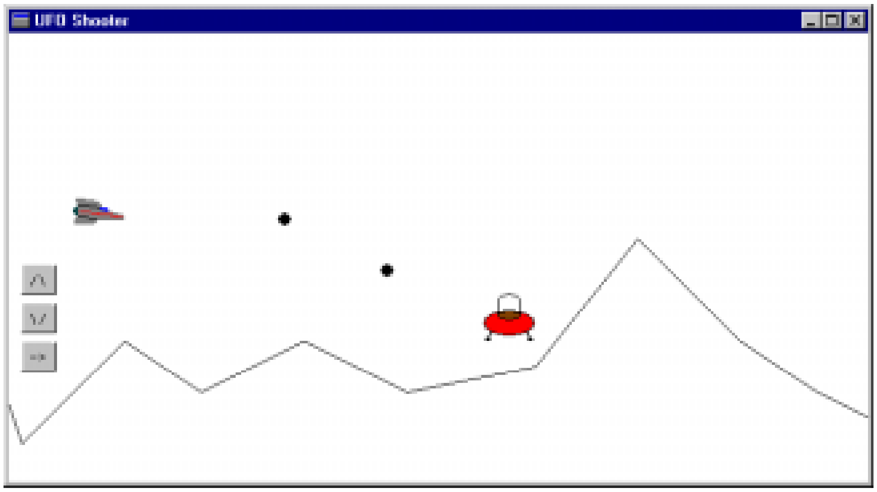


Figure A.2: The game, UFO, shooter, is a simple shooting game similar to Space Invaders. The player's ship on the left shoots at a randomly moving UFO object. The buttons on the left move the ship up and down and shoot the bullets.

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.

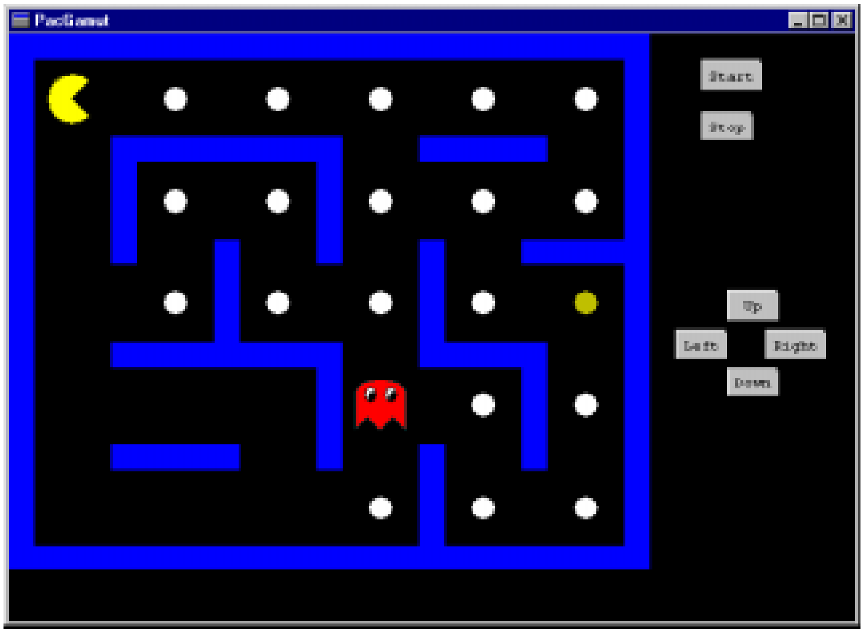


Figure A.3: In this Pacman game, the monster chases the Pacman object, while the Pacman tries to "eat" all the dots on the board. The player controls which direction Pacman faces. When Pacman eats a darker colored dot, the monster runs away from Pacman.

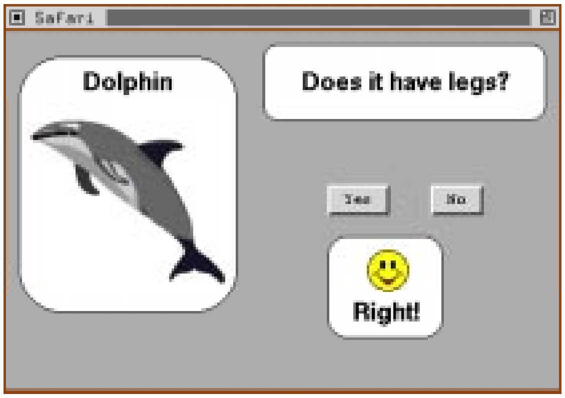


Figure A.4: Safari is a simple educational game. The computer asks questions about animals to which the player must answer yes or no.

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.

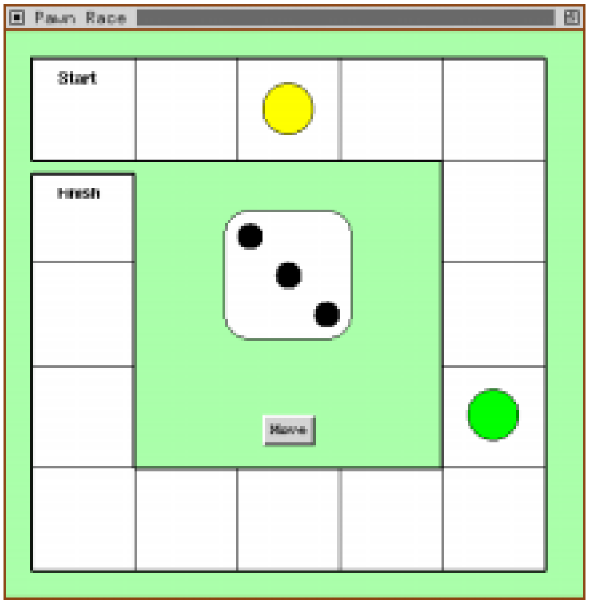


Figure A.5: Pawn Race is a Parcheesi-like board game where two players' pawns race each other to see who reaches the end first. When one player's piece lands on the other's, the landed on piece moves back to the start.

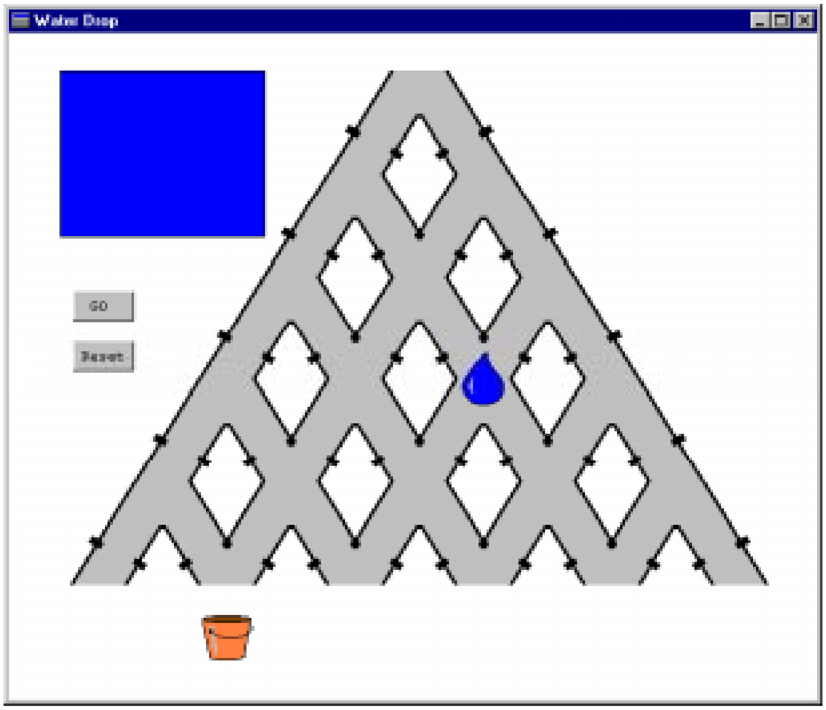


Figure A.6: In the Water Drop game, the player controls which direction the water droplet falls along a series of pipes. The goal is to get the droplet into the bucket at the bottom. The player chooses the droplet's direction by clicking on the blue rectangle at the top left.

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.

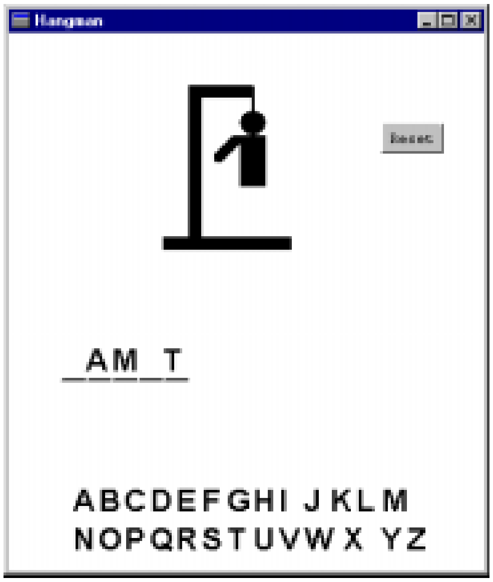


Figure A.7: In this Hangman game, the computer selects a word and the player must guess which letters are in the word.

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

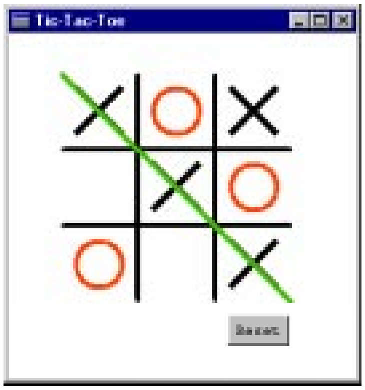


Figure A.8: This is Tic-Tac-Toe. Players alternate turns marking X or O in the grid of squares. The first player to get three in a row wins.

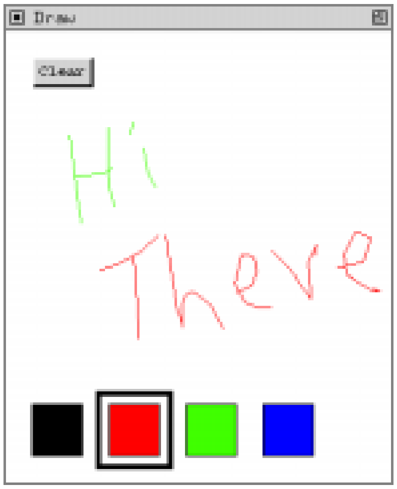


Figure A.9: In this drawing program, the player draws by dragging the mouse. The player can change the color of the line by clicking on the palette.

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.

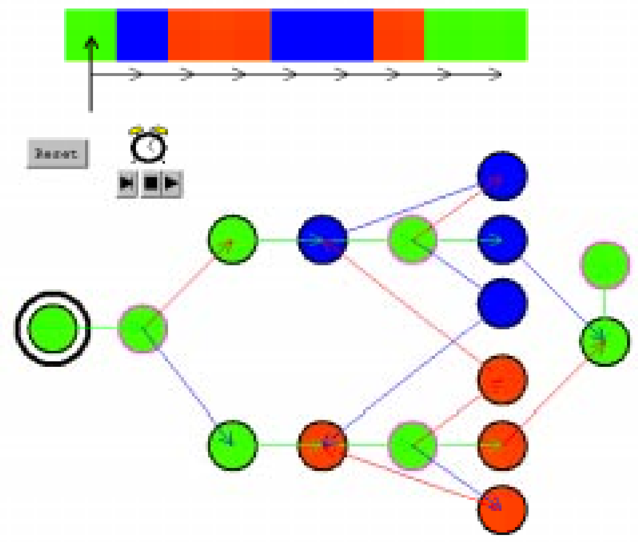


Figure A.10: This Turing machine simulation has a tape in the upper portion that records the program state and a finite state machine in the bottom portion to tell the Turing machine what to do.

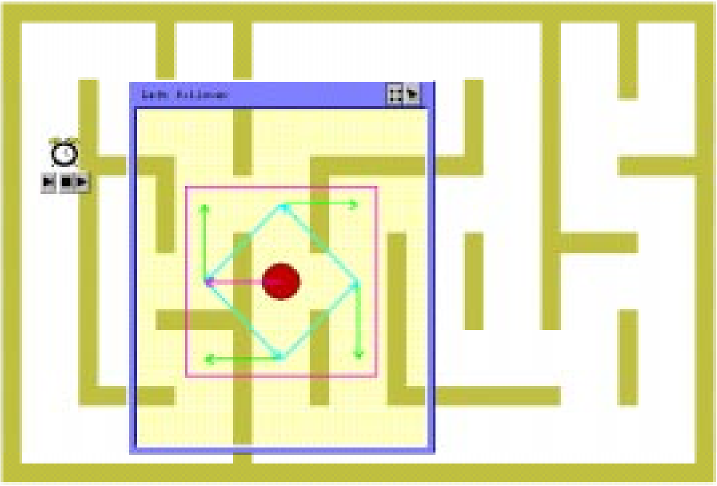


Figure A.11: In the Left Hand Maze Follower application, the circle follows the left wall of the maze by progressively testing the walls with arrow lines.

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.

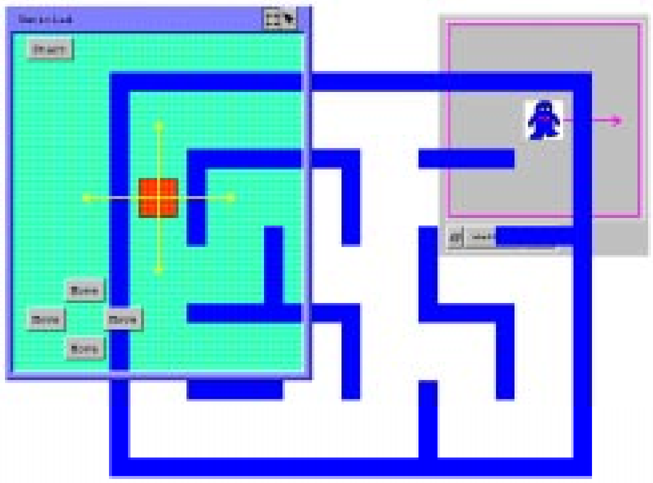


Figure A.12: In the Monster Hunt game, the player moves a circle object and the computer moves a monster object randomly.

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.

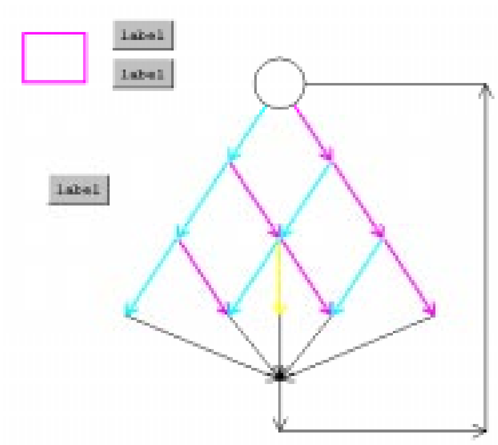


Figure A.13a: In Q1, the colors of the lines indicate which way the circle moves.

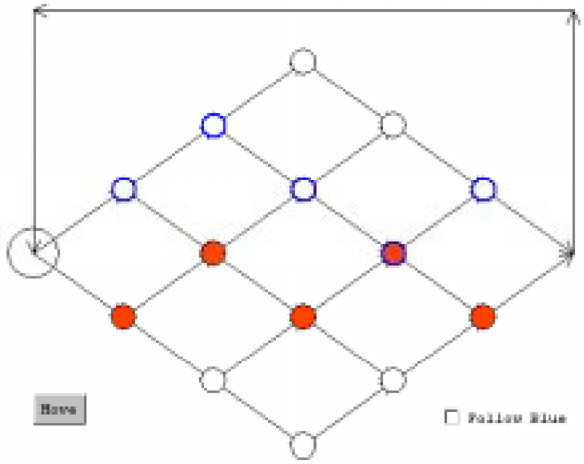


Figure A.13b: In Q2, the colors of circles at the ends of the lines indicate which way the circle moves.

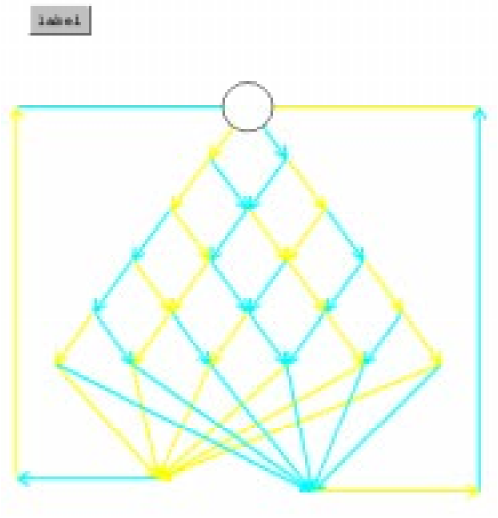


Figure A.13c: In Q3, the colors of the lines once again indicate which way the circle moves, but the circle must move two jumps at a time.

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.

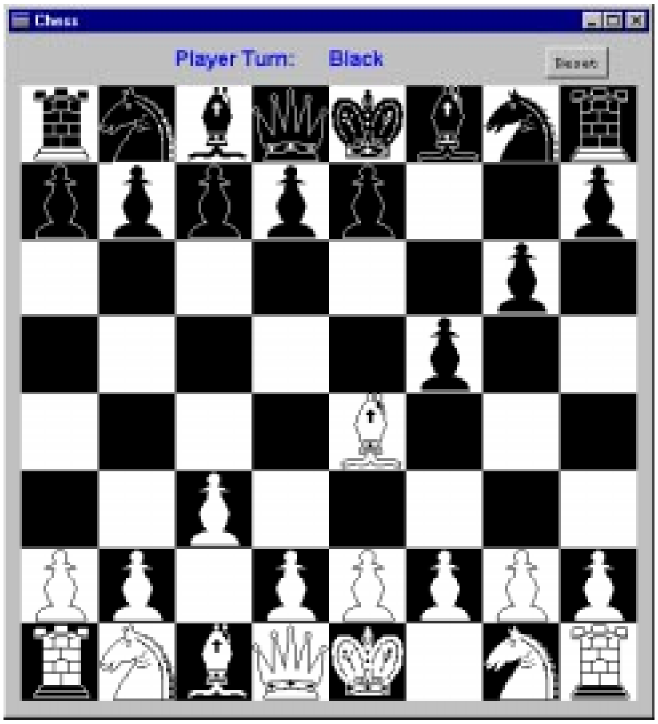


Figure A.14: This is the unfinished Chess game application. Currently, the only implemented feature is that the players can drag pieces on the board. Conceivably, much more of the game could be built.

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.

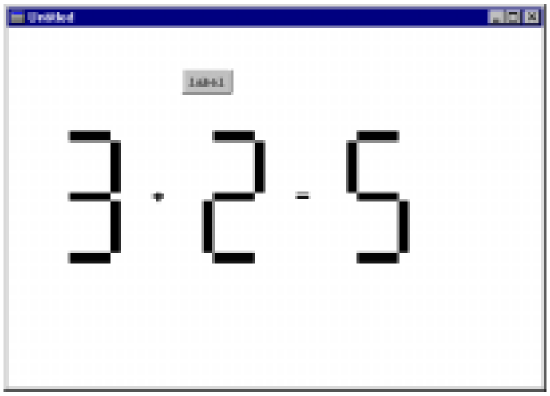


Figure A.15: The digital-looking numbers are stored in decks. The last deck was made to add together the values of the first two decks when the player pushed button at the top.

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

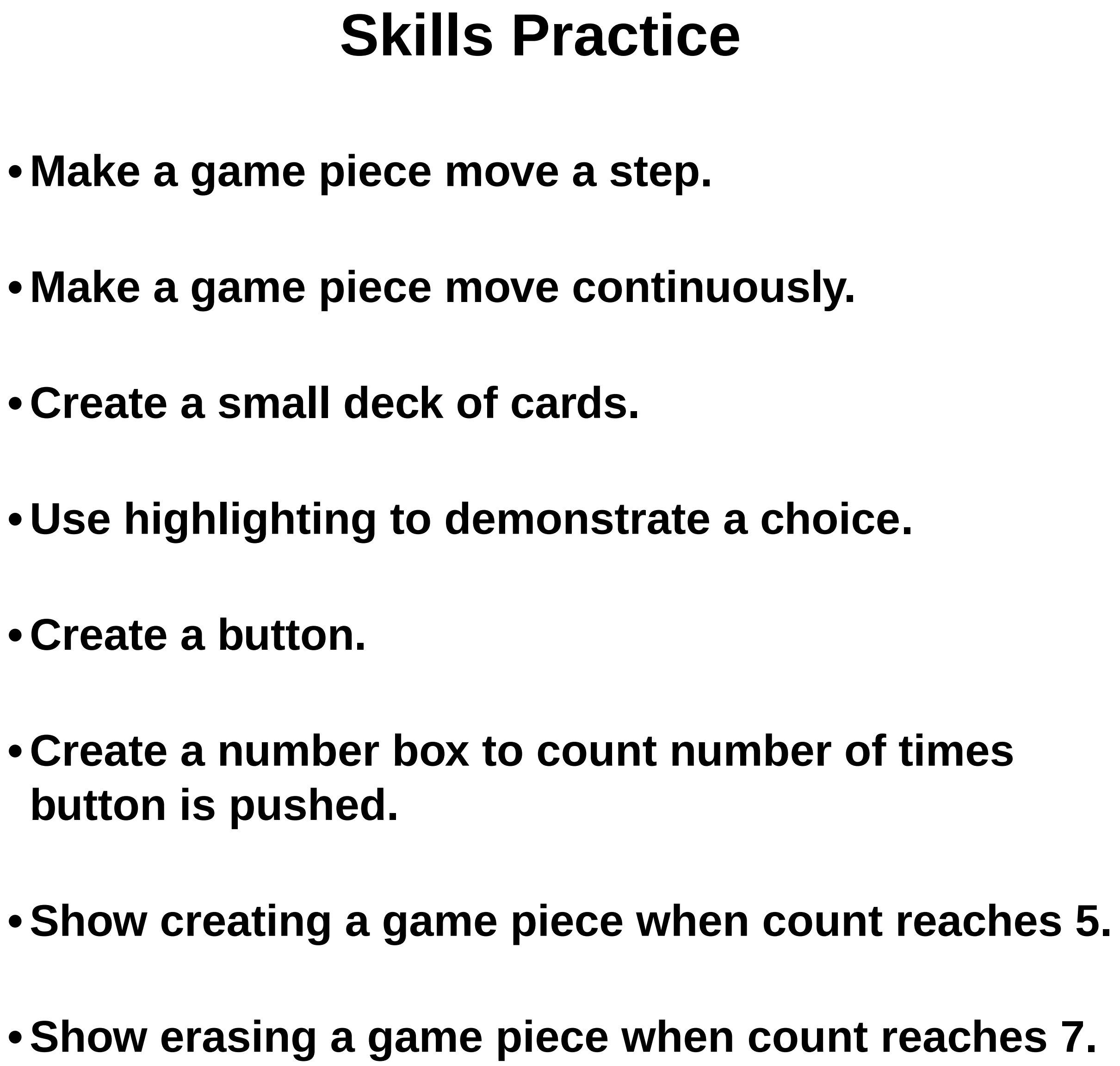
When each experiment session was complete, the experimenter would debrief the participant using this statement.

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.



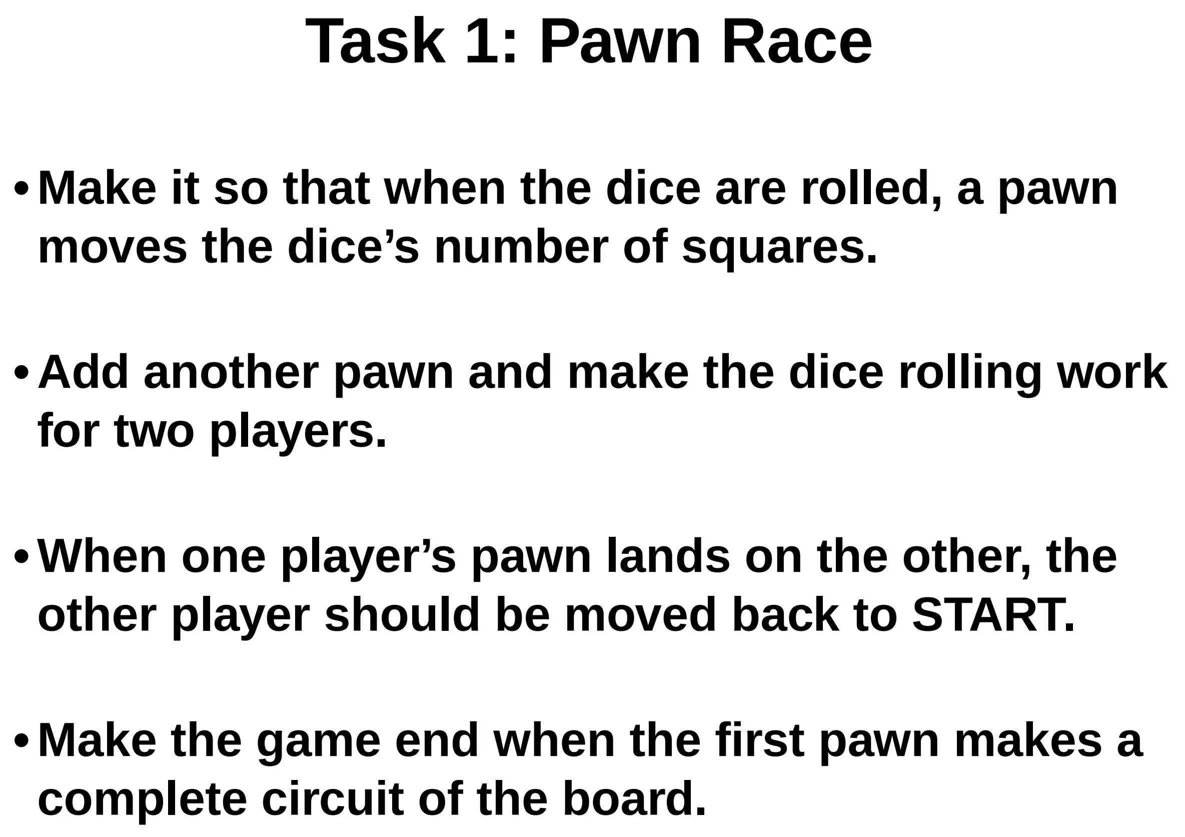
Skills Practice

- 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.

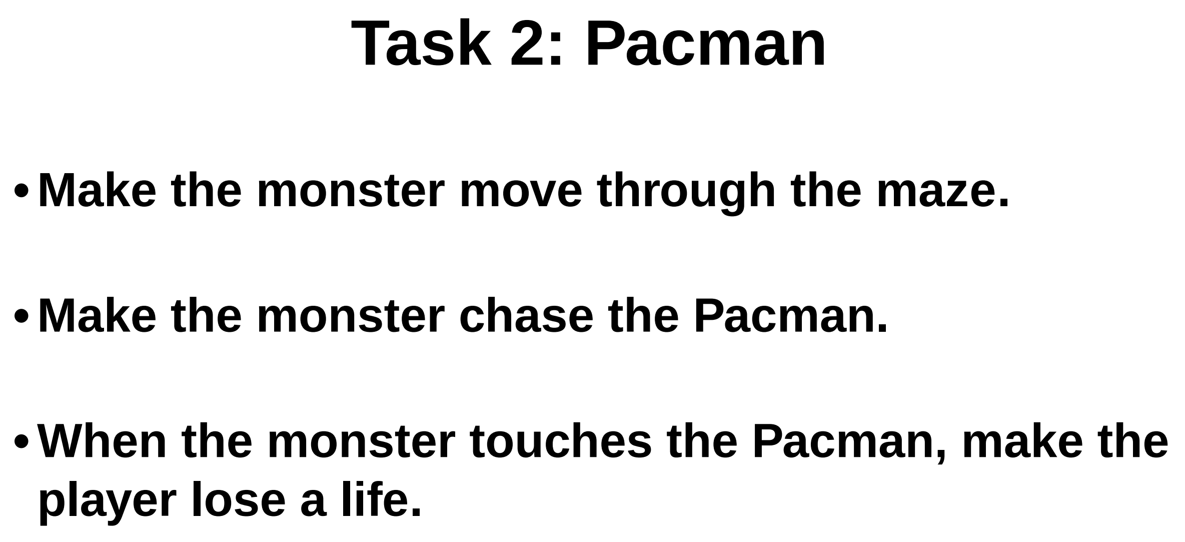
B.7.1 Pawn Race



Task 1: Pawn Race

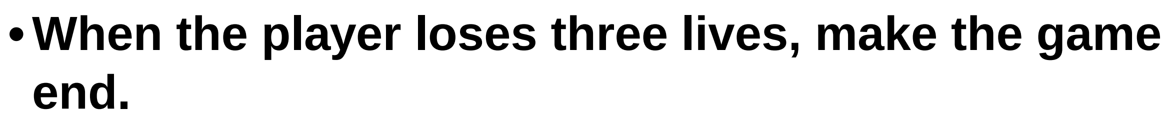
- Make it so that when the dice are rolled, a pawn moves the dice’s number of squares.
- Add another pawn and make the dice rolling work for two players.
- When one player’s pawn lands on the other, the other player should be moved back to START.
- Make the game end when the first pawn makes a complete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

B.7.2 Pacman



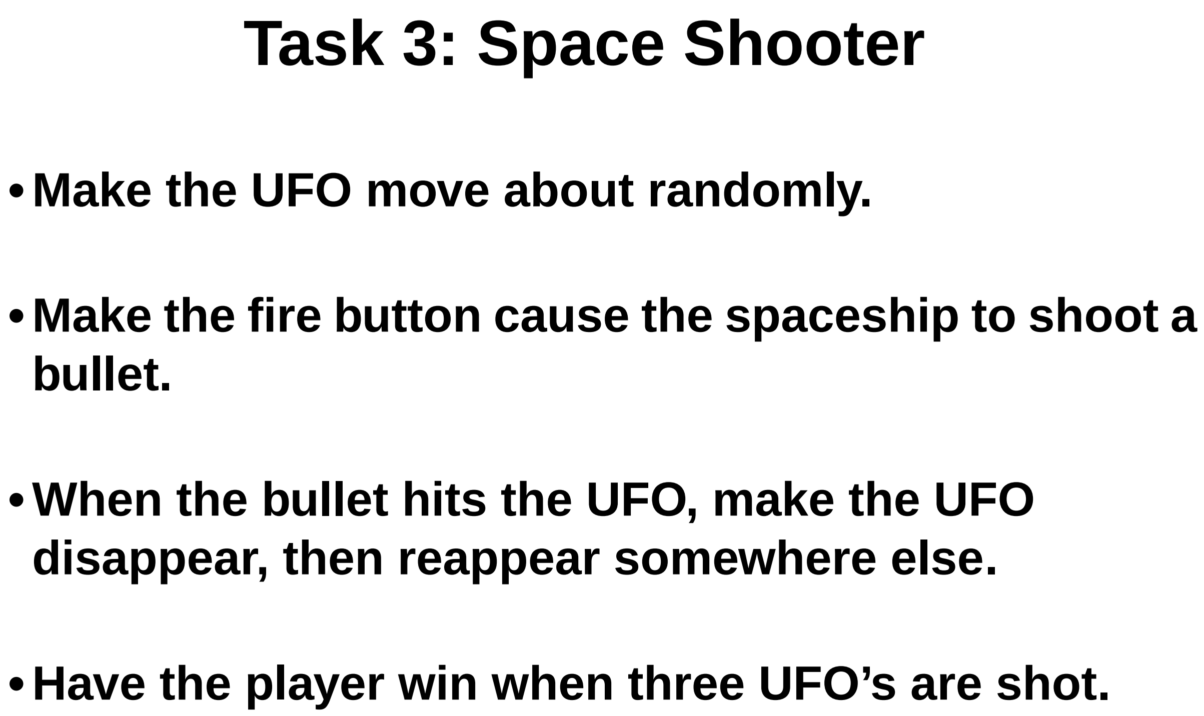
Task 2: Pacman

- 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



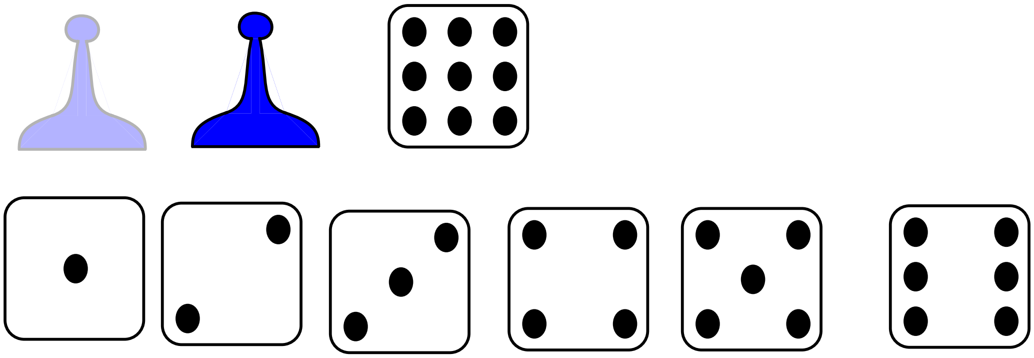
Task 3: Space Shooter

- **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.**

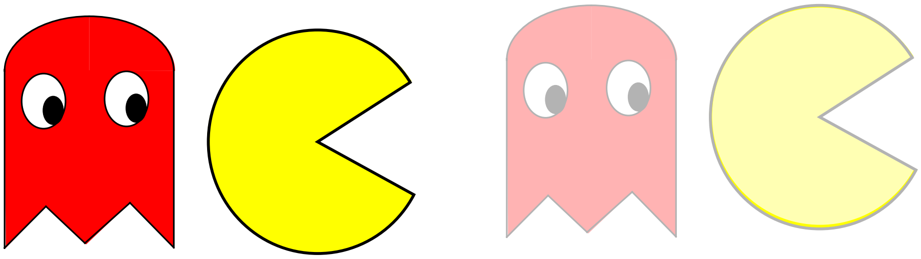
B.8 Paper Prototype Materials

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.

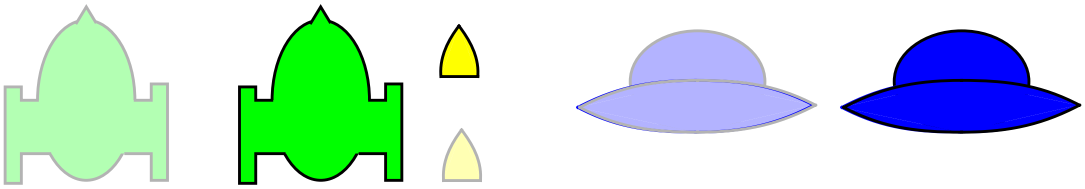
B.8.4 Task 1 Pieces



B.8.5 Task 2 Pieces



B.8.6 Task 3 Pieces



B.8.7 Buttons and Modes

Do Something

FIRE

Stop That

Roll Dice

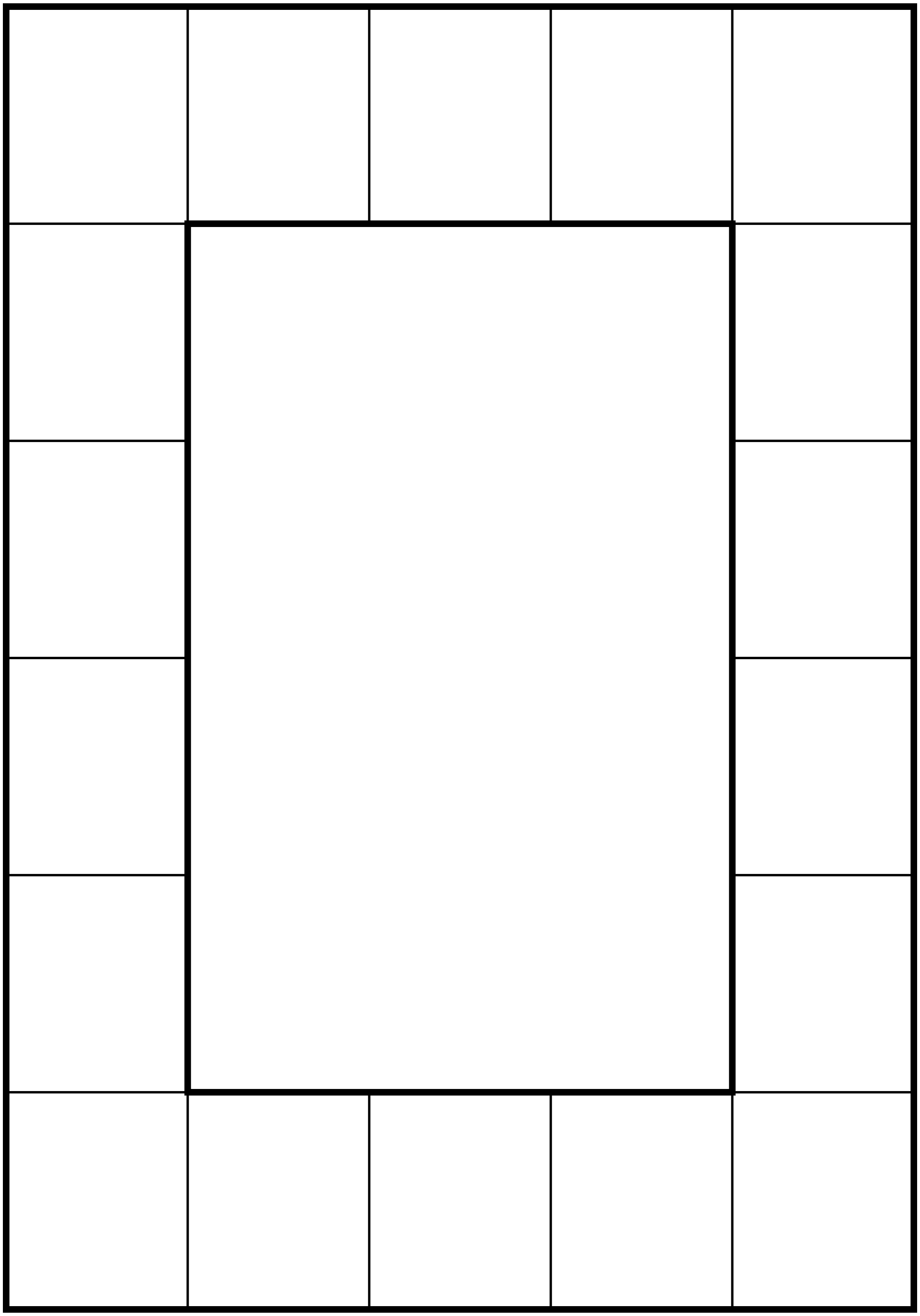
RUN

BUILD

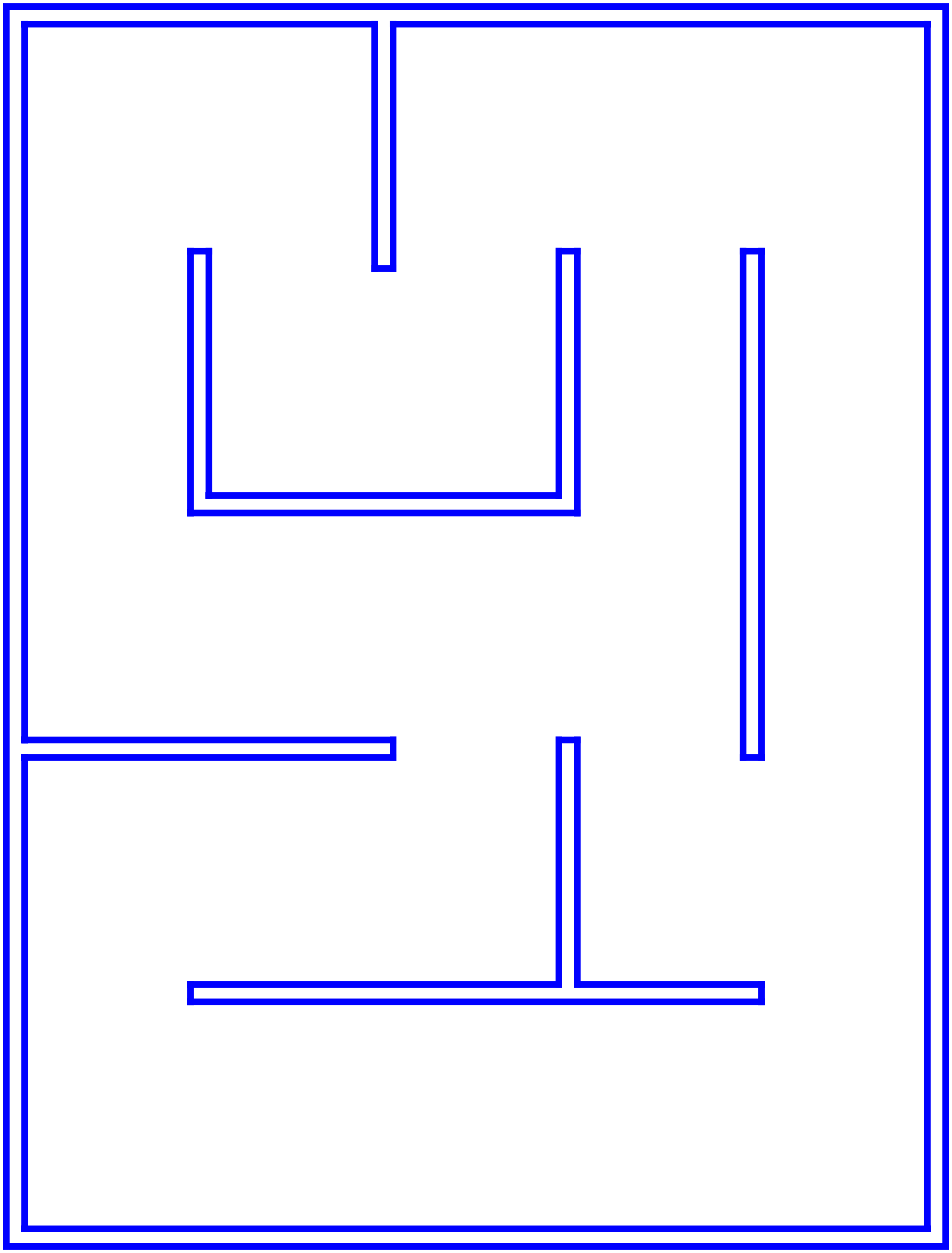
WATCH

CONTINUE

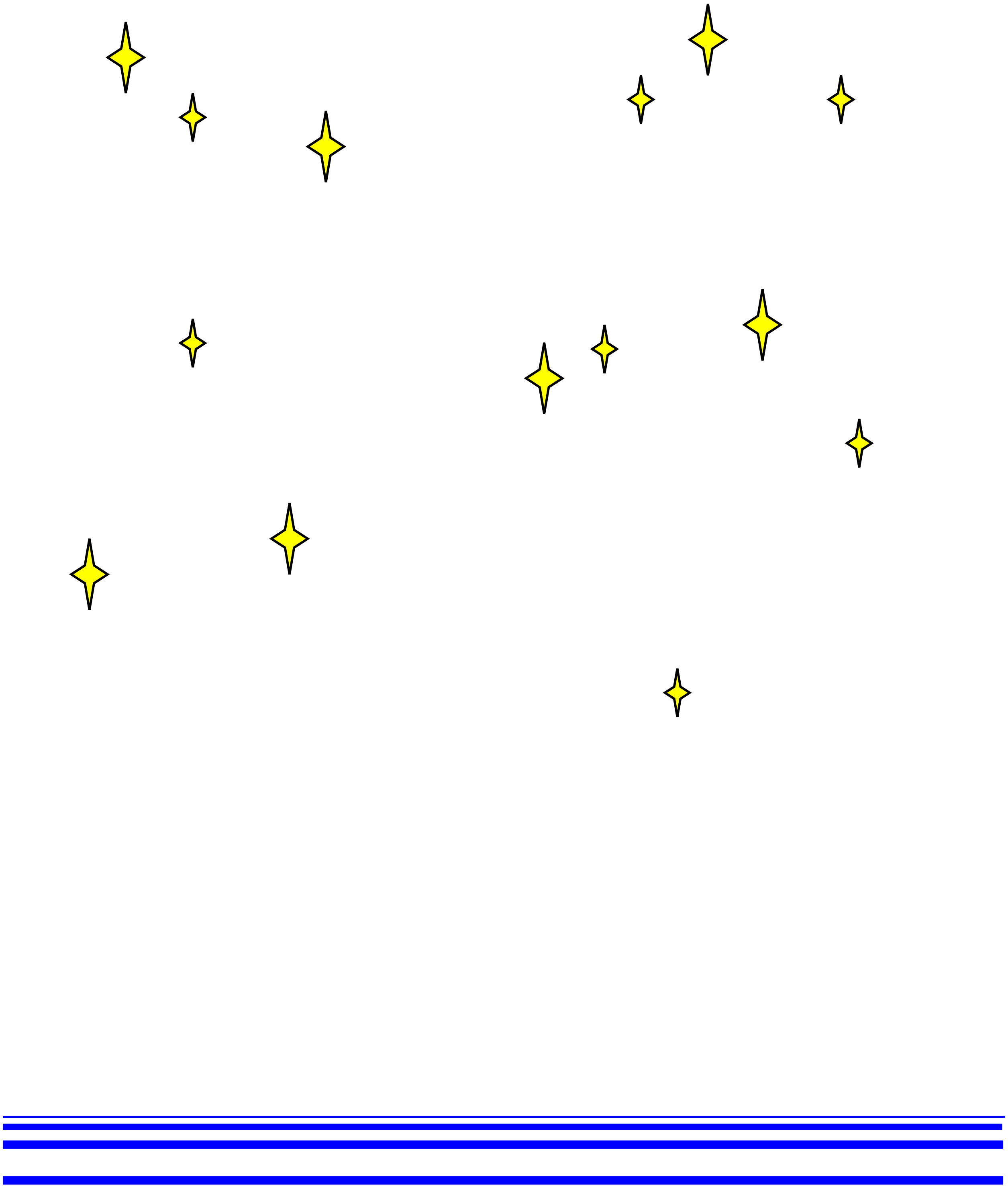
B.8.8 Task 1 Background



B.8.9 Task 2 Background

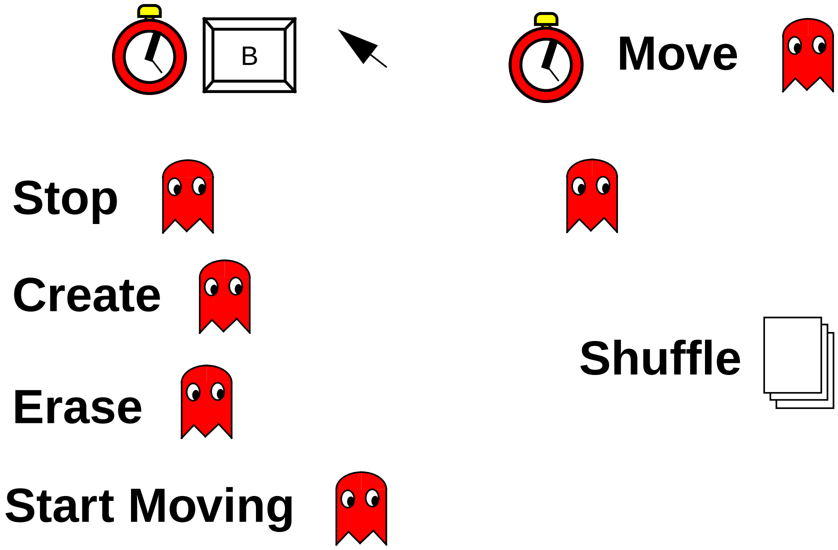


B.8.10 Task 3 Background



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.

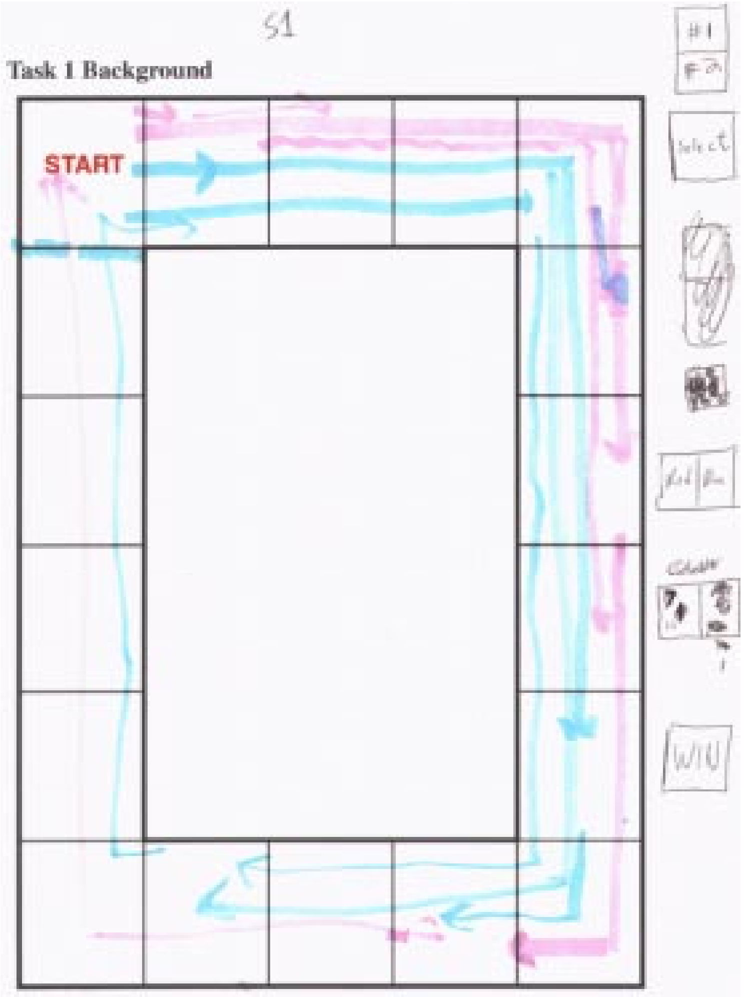
(3) Explain the logic the system uses - why it does what it does
Add more help/manuals to explain and for reference

(4) do an if thing

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

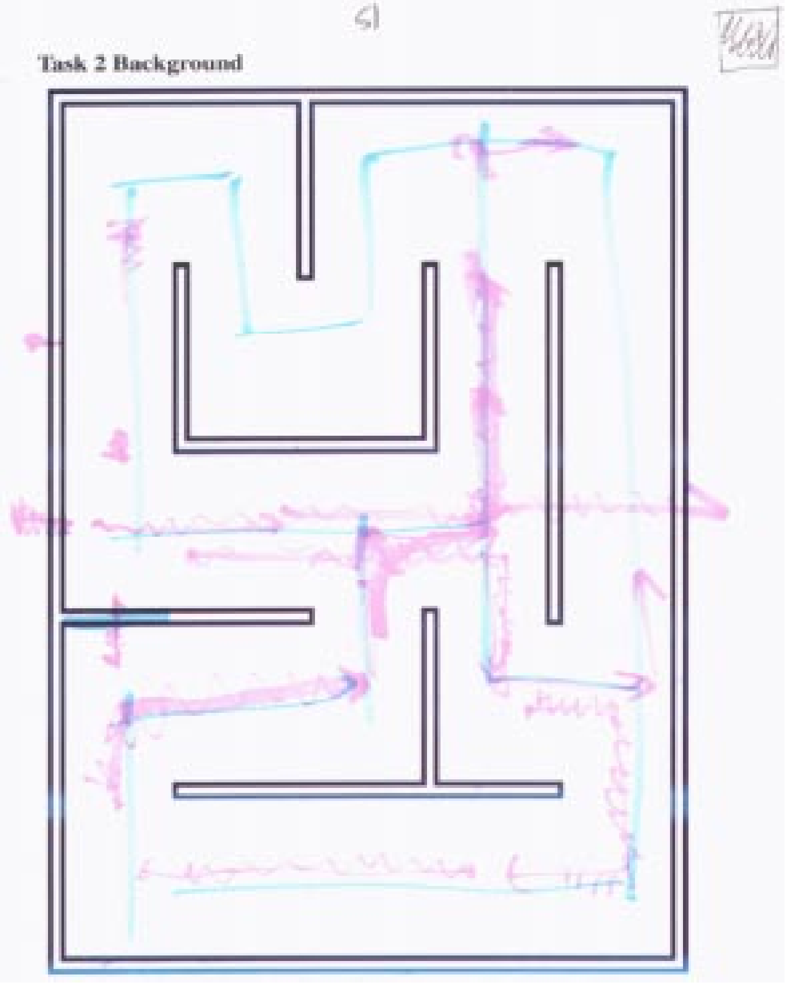
C.2 Participant One’s Drawings

C.2.1 Task One: Pawn Race

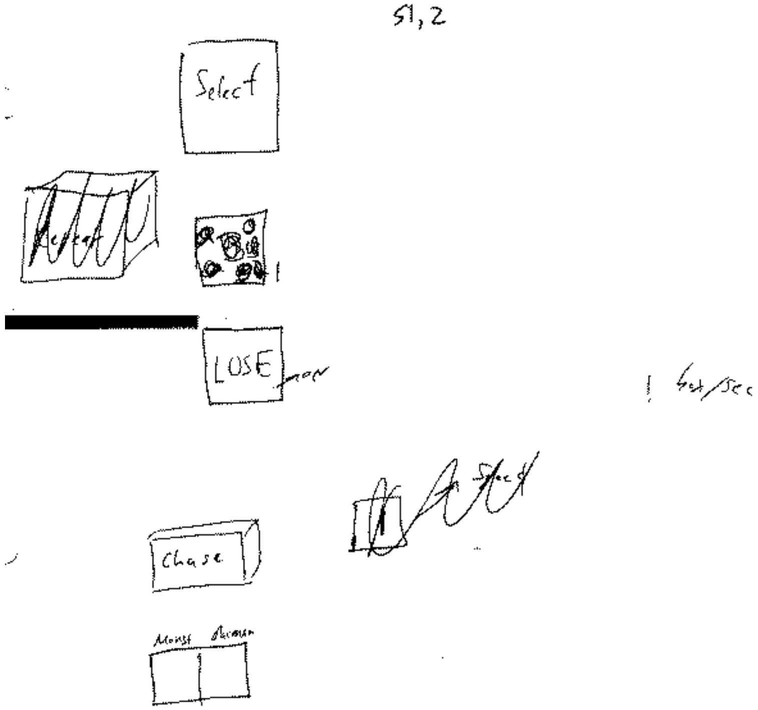


C.2.2 Task Two: Pacman

C.2.2.1 Main Board



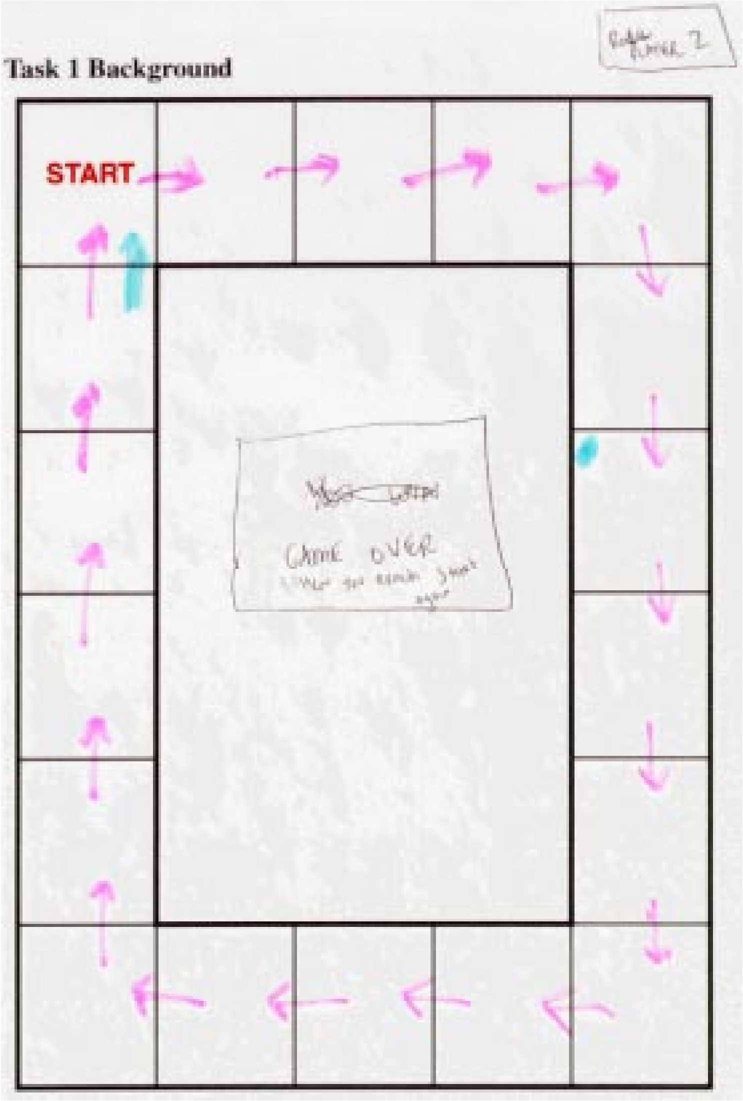
C.2.2.2 Offscreen Area



C.3 Participant Two’s Drawings

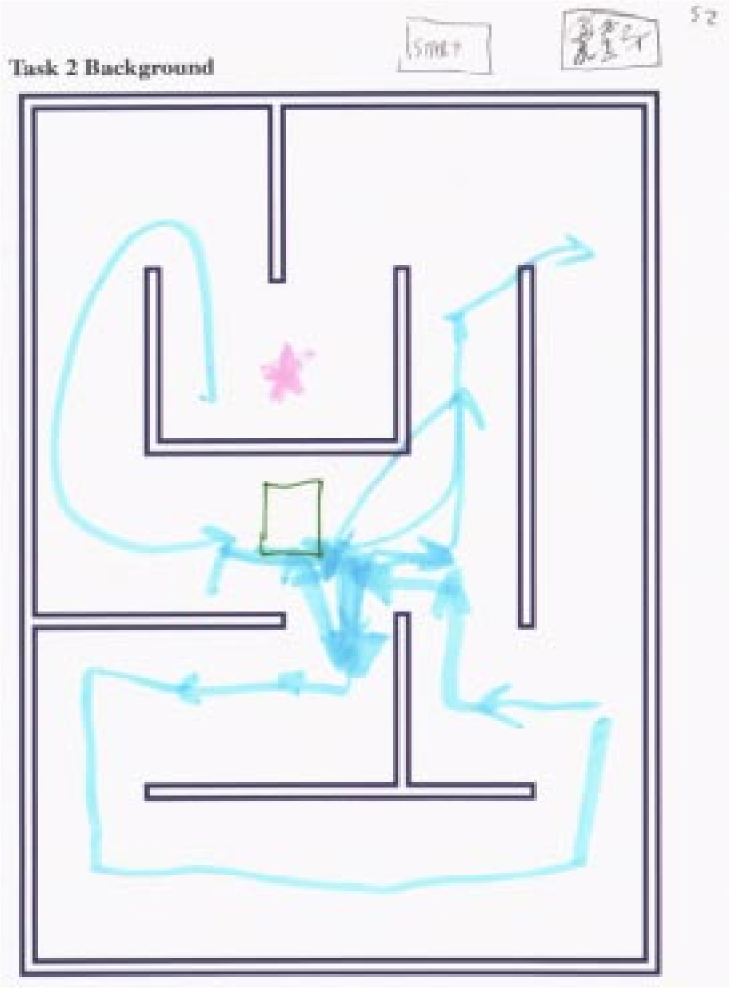
C.3.1 Task One: Pawn Race

C.3.1.1 Main Board

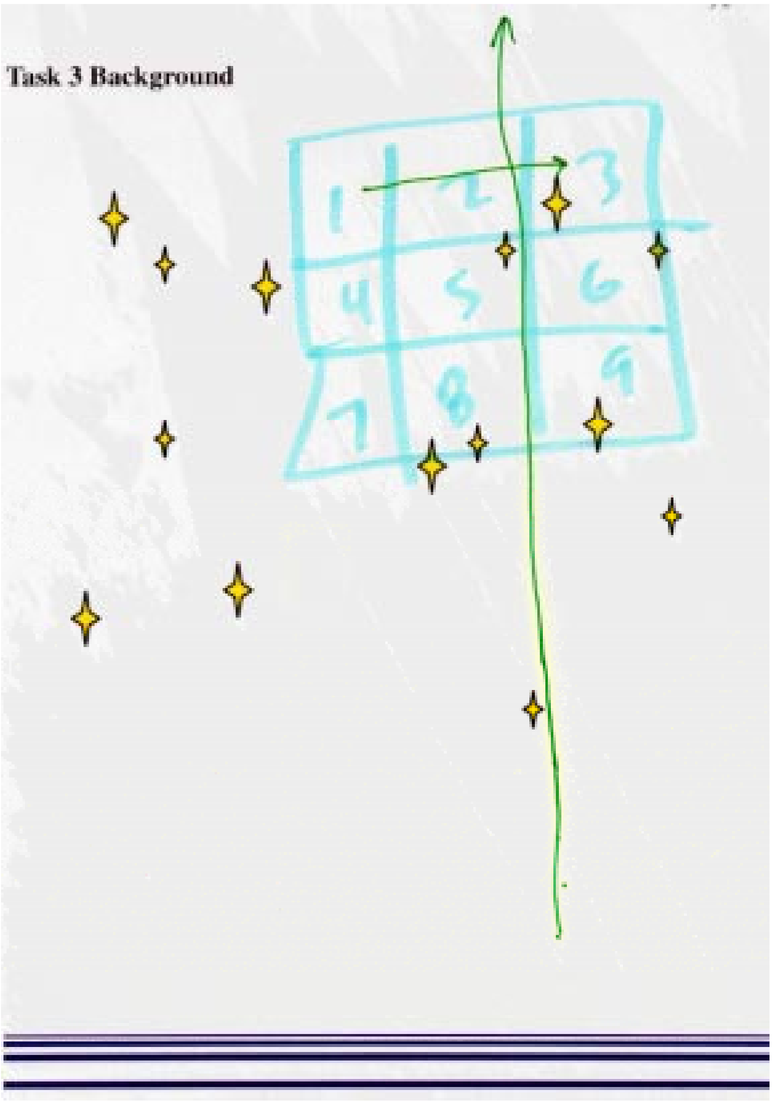


C.3.1.2 Offscreen Area

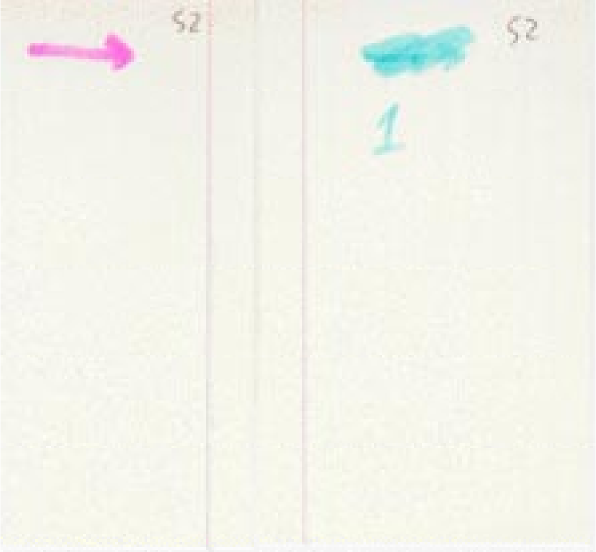
C.3.2 Task Two: Pacman



C.3.3 Task Three: Space Shooter



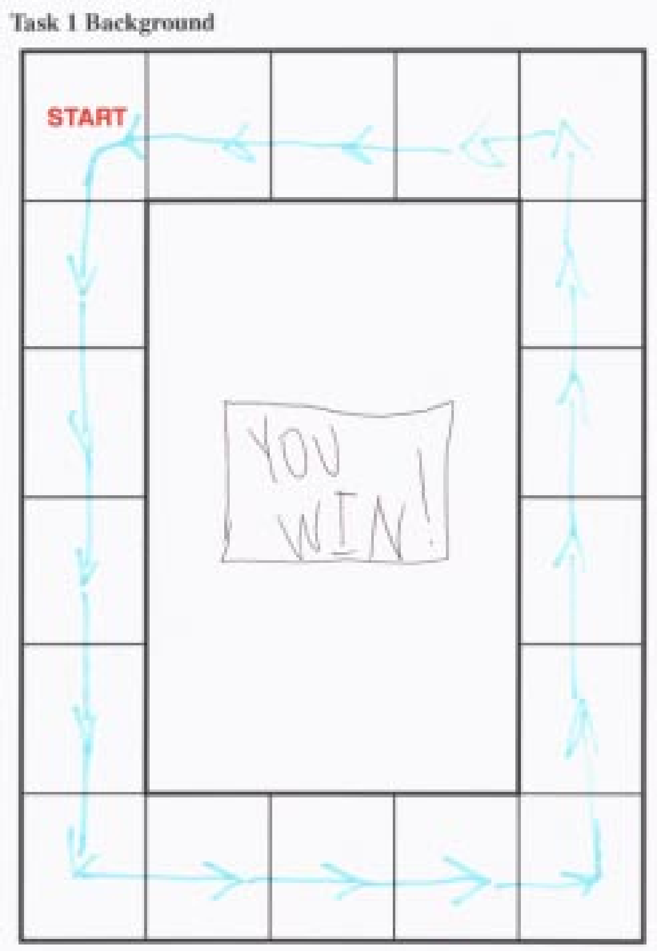
C.3.4 Participant Two’s Cards



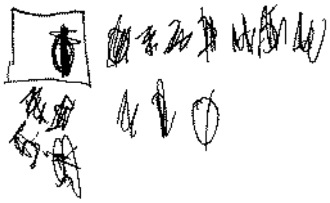
C.4 Participant Three's Drawings

C.4.1 Task One: Pawn Race

C.4.1.1 Main Board

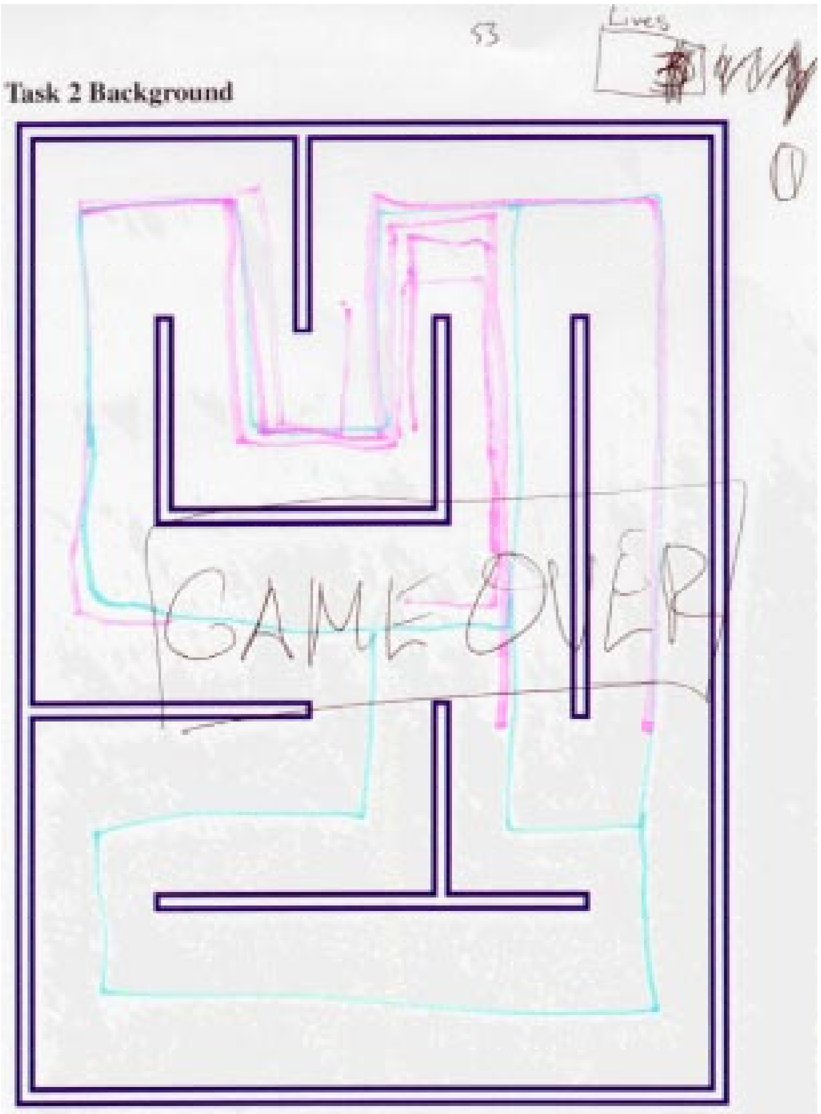


C.4.1.2 Offscreen Area

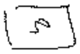


C.4.2 Task Two: Pacman

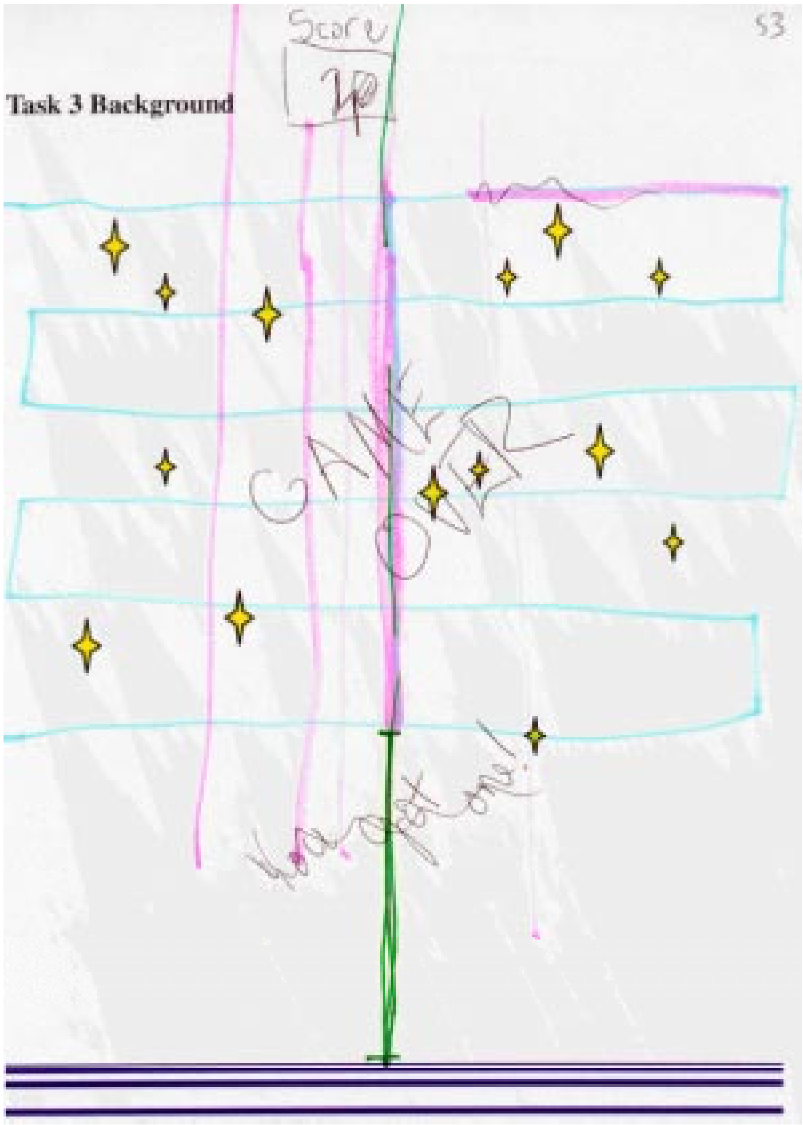
C.4.2.1 Main Board



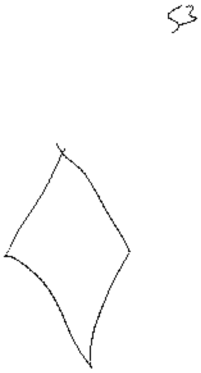
C.4.2.2 Offscreen Area

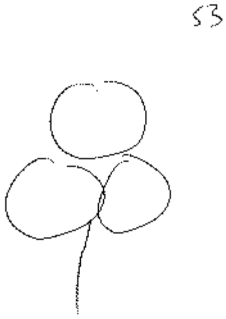


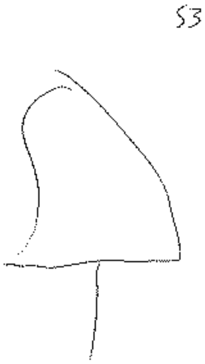
C.4.3 Task Three: Space Shooter

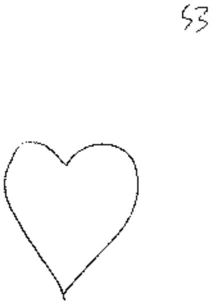


C.4.4 Participant Three’s Cards





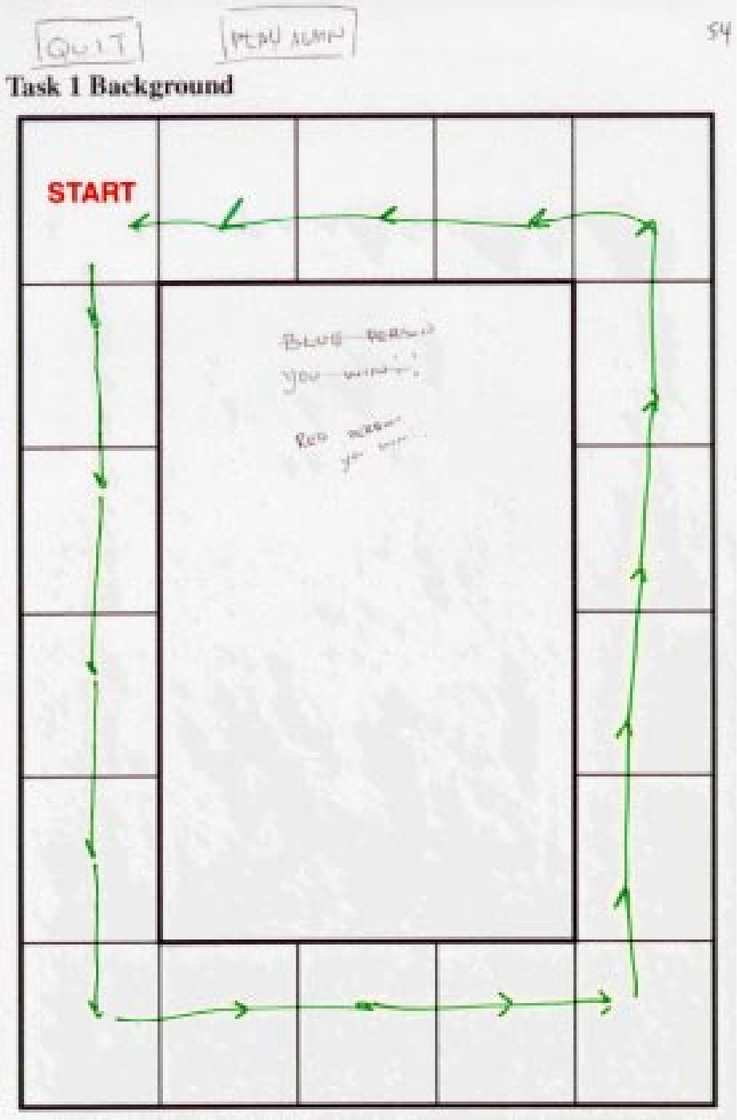




C.5 Participant Four’s Drawings

C.5.1 Task One: Pawn Race

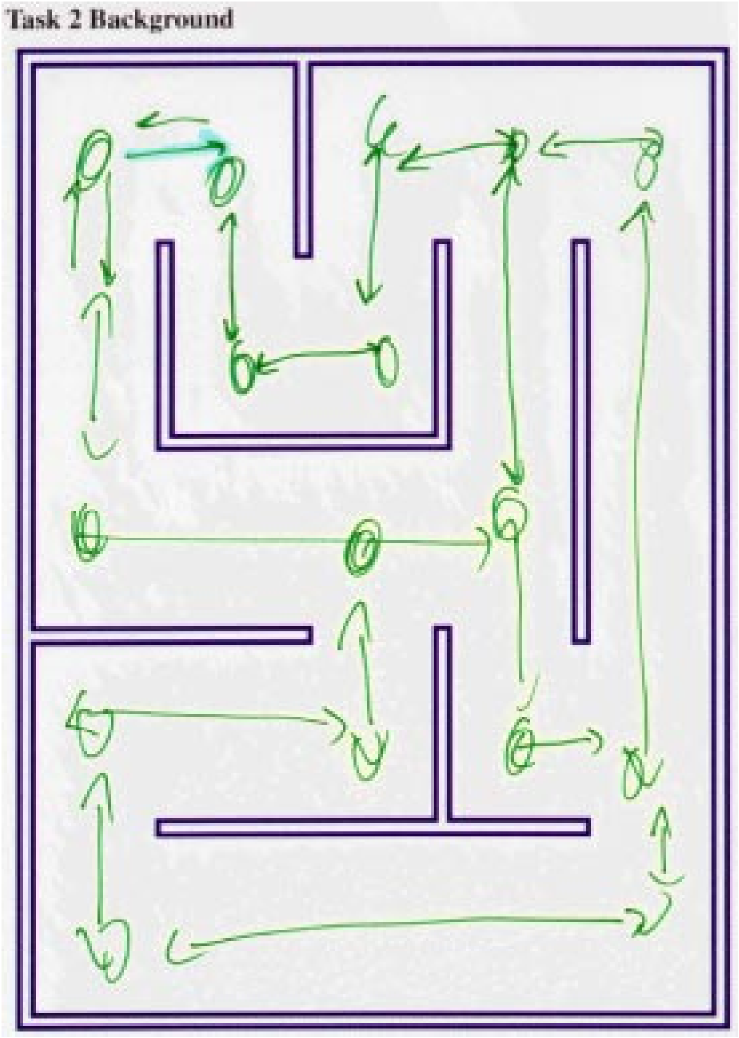
C.5.1.1 Main Board



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

up

down

left

Right

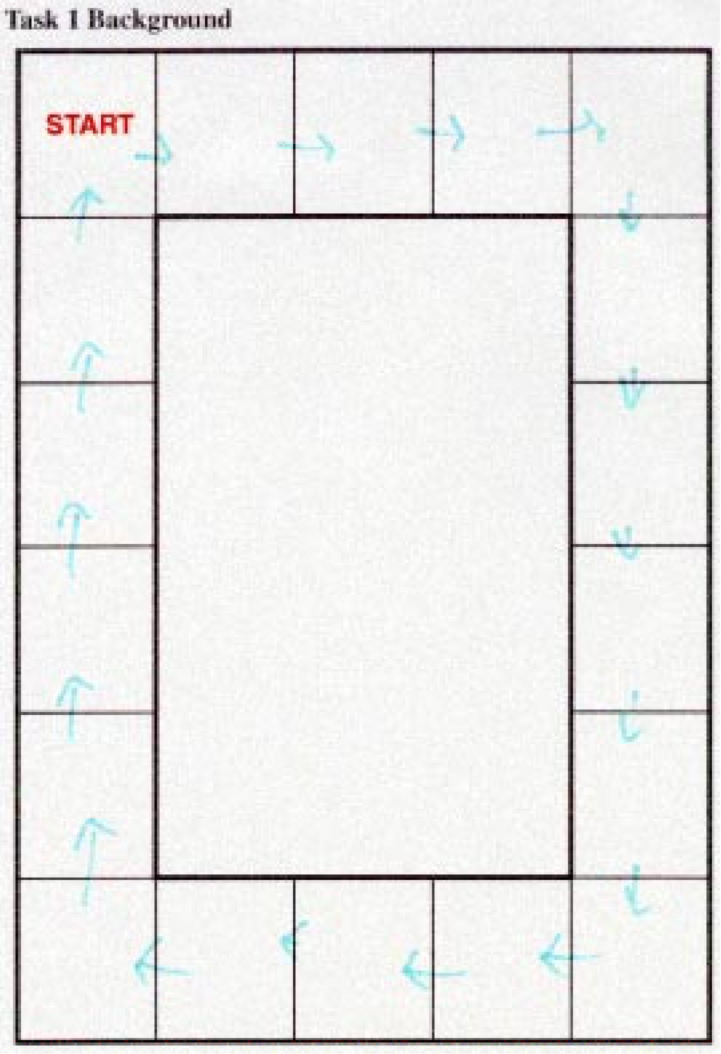
Pink

Blue

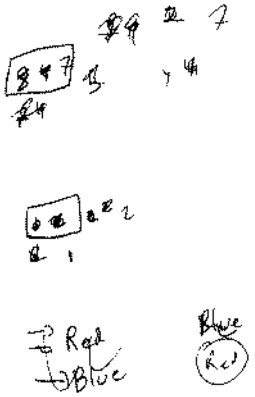
C.6 Participant Five's Drawings

C.6.1 Task One: Pawn Race

C.6.1.1 Main Board

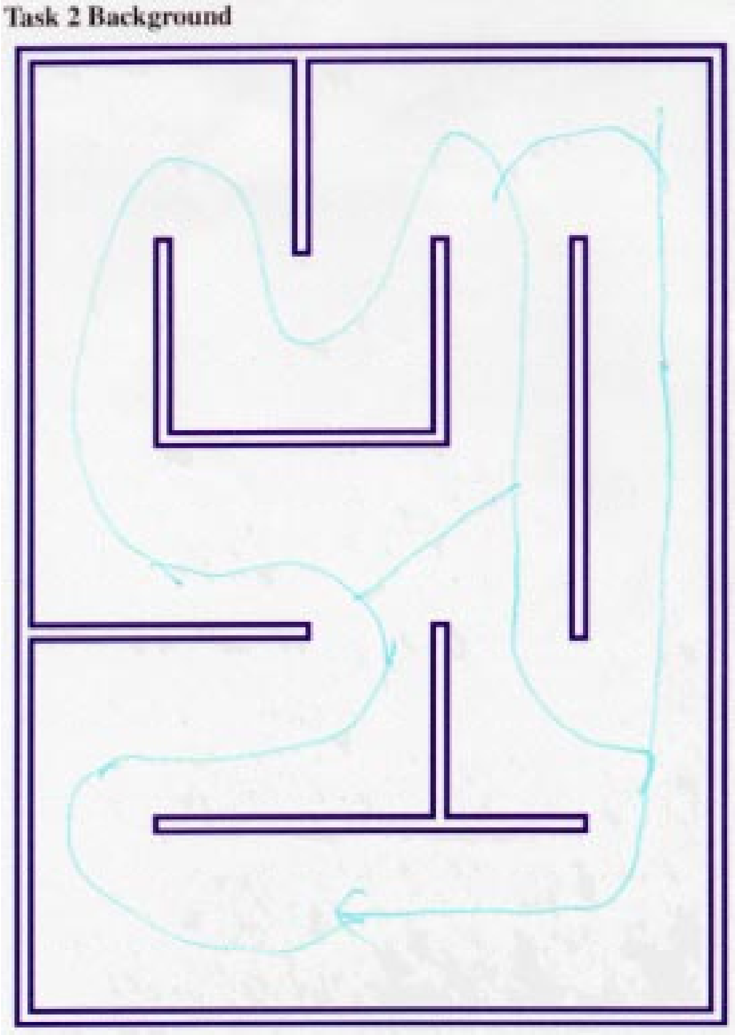


C.6.1.2 Offscreen Area



C.6.1.3 Task Two: Pacman

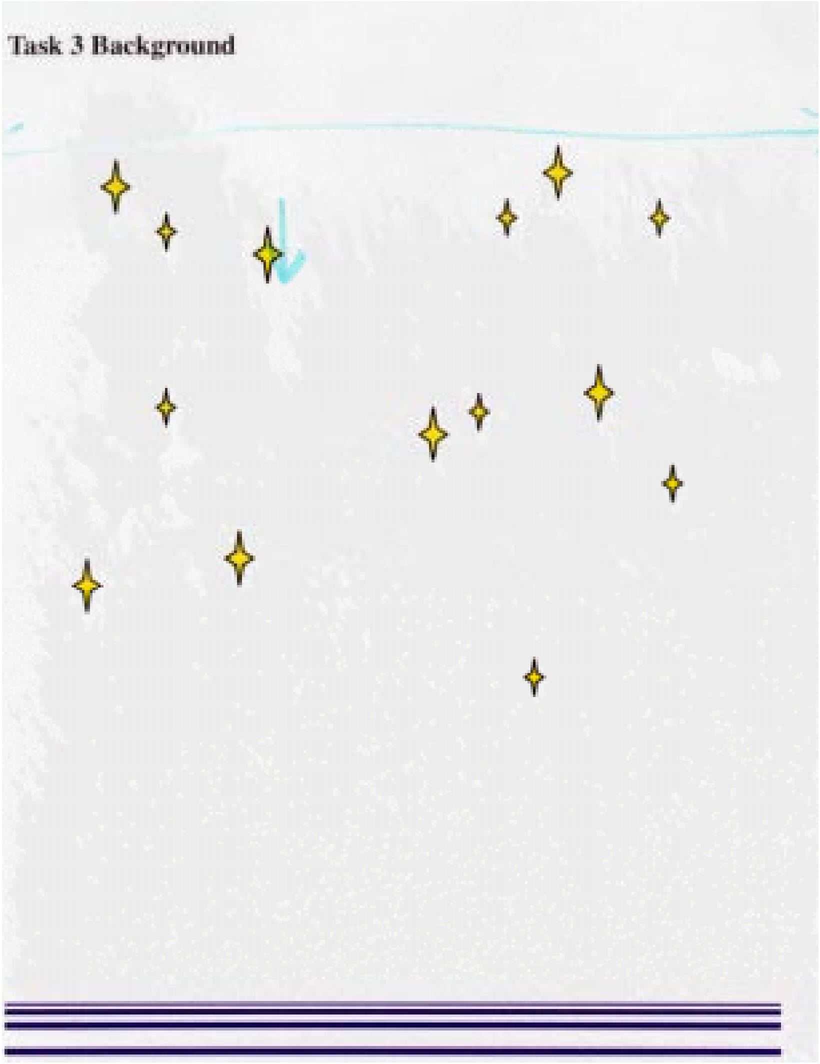
C.6.1.4 Main Board



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

Goto A

5

Up

5

Go to B

5

down

5

Go to C

5

left

5

Go to D

5

right

5

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 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.

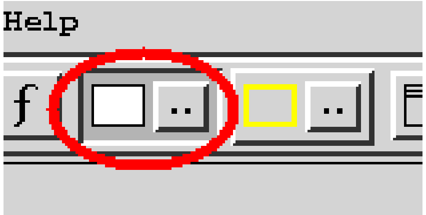


Figure 1: The color selector. This shows the color Gamut uses to draw new objects. Push the “..” button to change the color.

- **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.)**

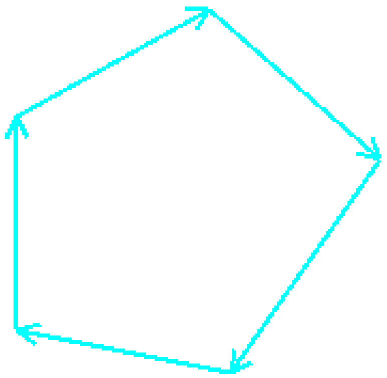


Figure 2: Draw a path of arrow lines like this.

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

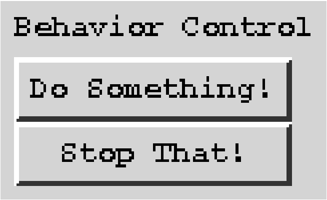


Figure 3: The Do Something and Stop That buttons. Use these to demonstrate new behavior.

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.**

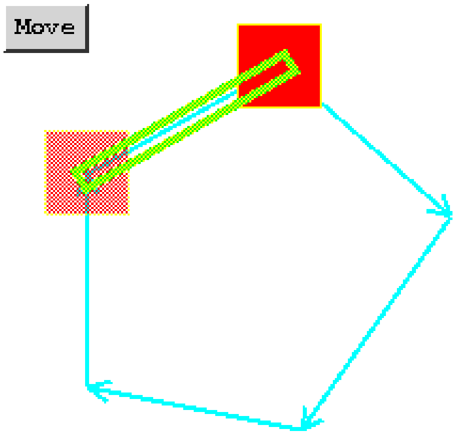


Figure 4: This is what the area near the rectangle should look like right before you press the “Done” button and finished demonstrating.

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.

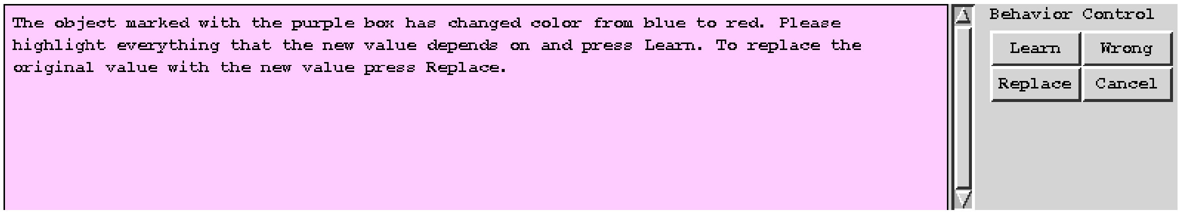
You can make more behaviors by continuing to show Gamut more examples. Let’s make another behavior that changes the color of the rectangle back and forth between red and blue.

- **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



The object marked with the purple box has changed color from blue to red. Please highlight everything that the new value depends on and press Learn. To replace the original value with the new value press Replace.

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.

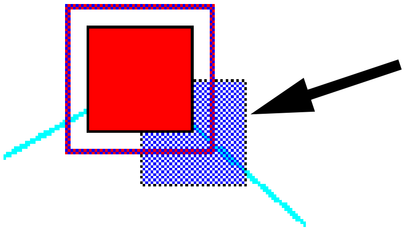


Figure 6: This shows the rectangle along with its “ghost object” which is the blue faded rectangle that is offset to the lower right from the original.

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 T) 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.

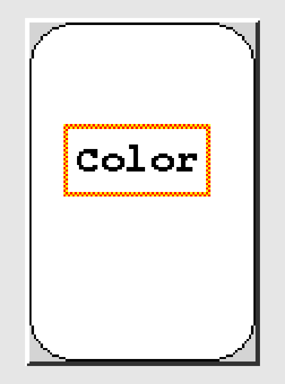


Figure 7: The “Color” label is placed in the visible area to make it visible on the card and it has a mark around it to show it is “pre-highlighted.”

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.

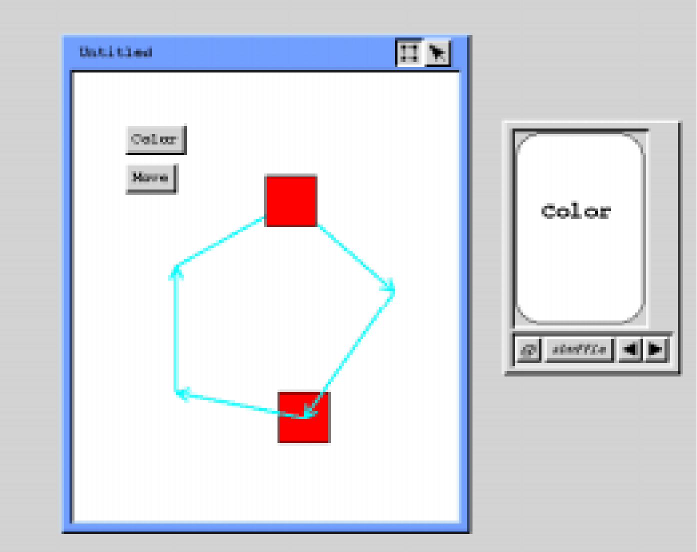


Figure 8: The main window with the assembled deck.

- **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 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.

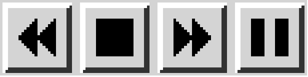


Figure 9: The time control buttons. The buttons are: back, stop, forward, and pause.

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.

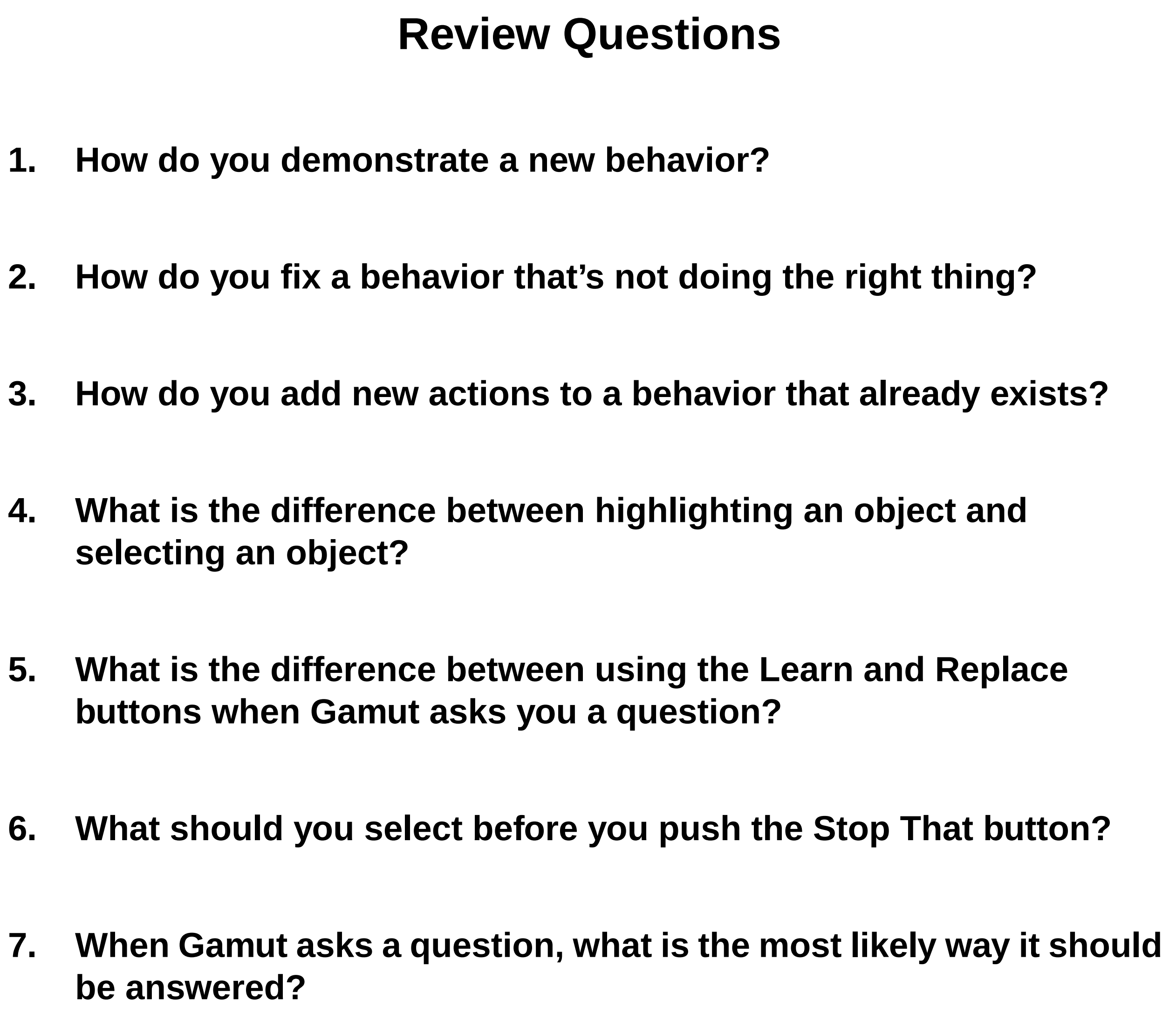
Starting Over: If Gamut crashes or you are really having trouble building one of the trial games, you can always start over. The first part of each experiment will ask you to load a file as your starting point. You can restart by reloading that file. You can also save your progress if you get the game 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.

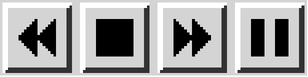


Figure 1: The time control buttons. The buttons are: back, stop, forward, and pause.

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).

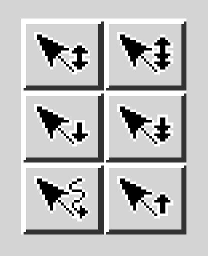


Figure 2: This palette contains all of the player input icons. The “click” icons have an arrow head pointing both up and down.

- **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.”**

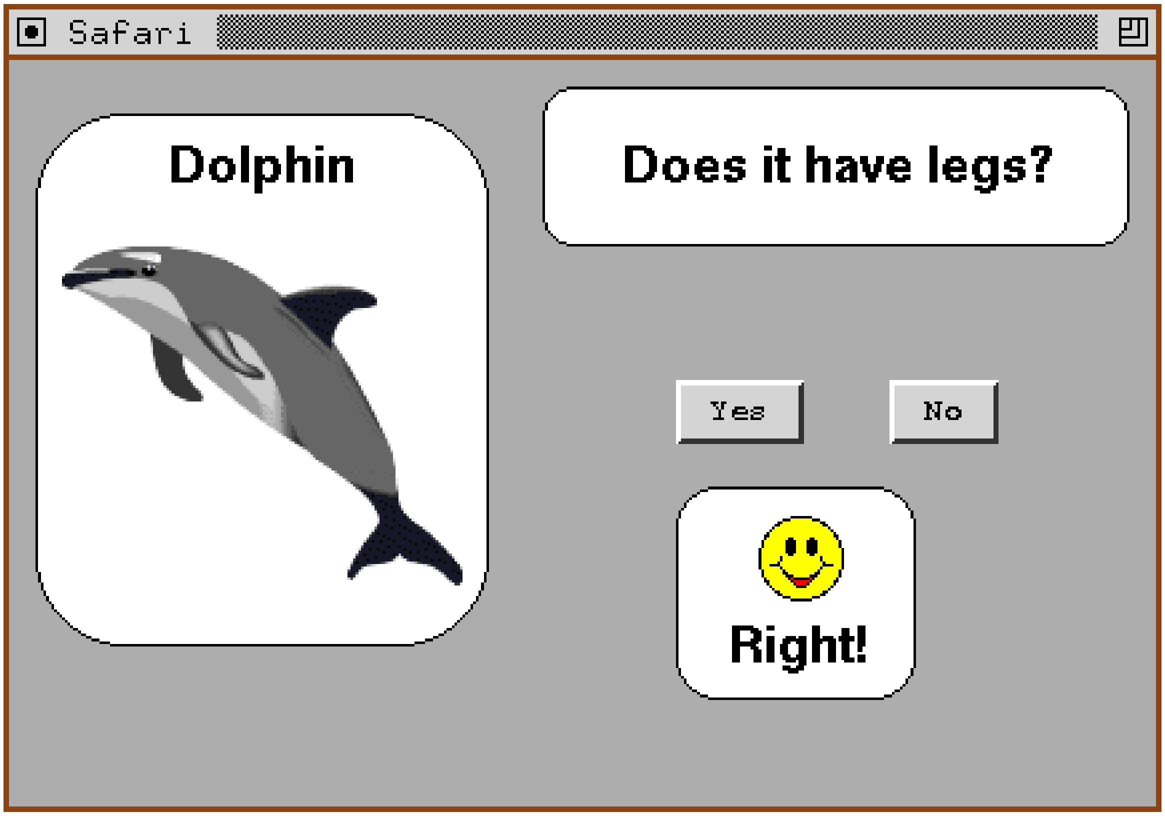
D.10 Final Usability Tasks

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.

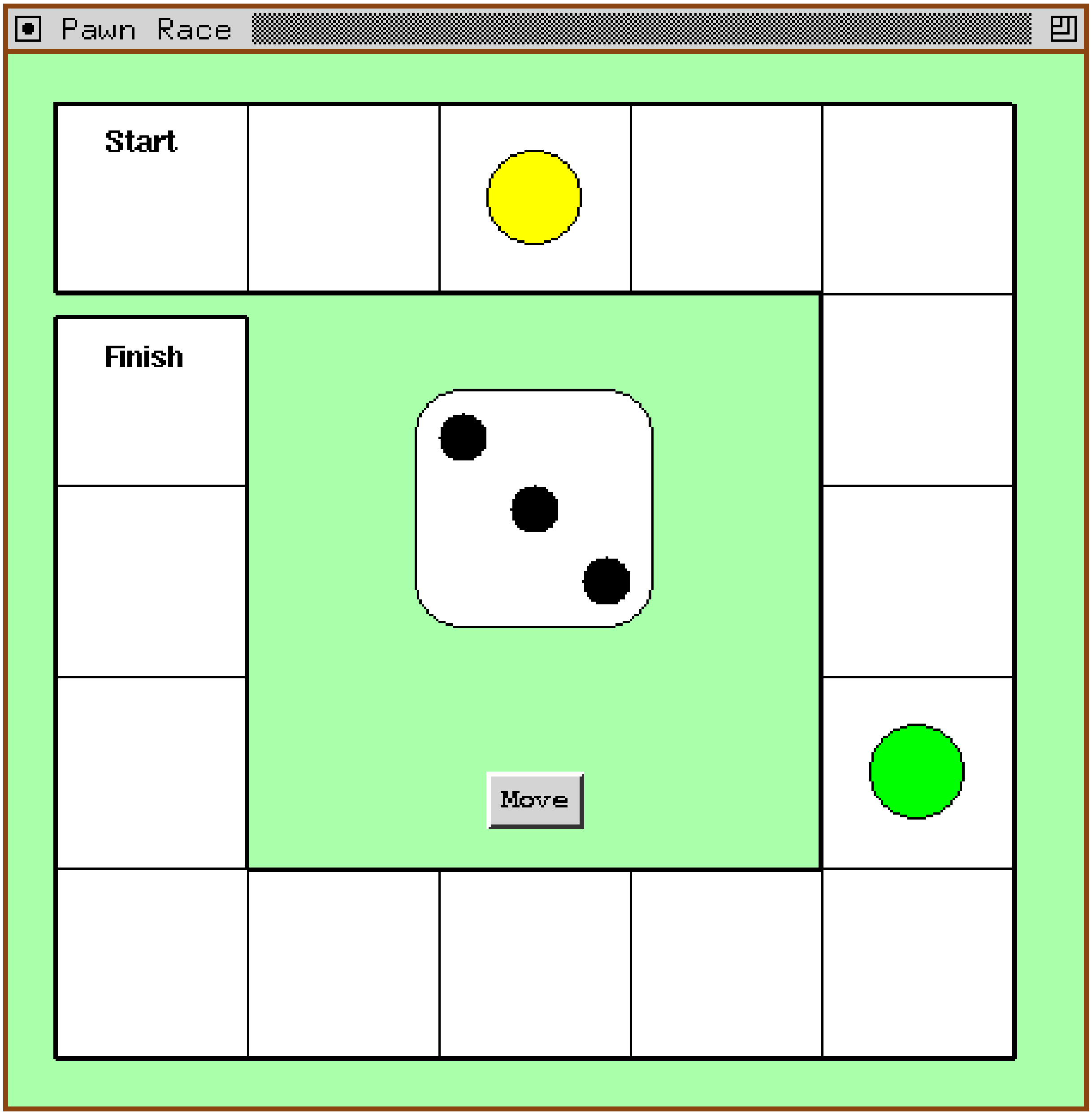
Gamut is just a prototype system, so it only has a limited range of properties that it can detect. It doesn't know anything about animals and it certainly cannot read. Also, Gamut can only match the whole text in a text object and cannot pick out words or phrases. For instance, Gamut cannot see the word "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.



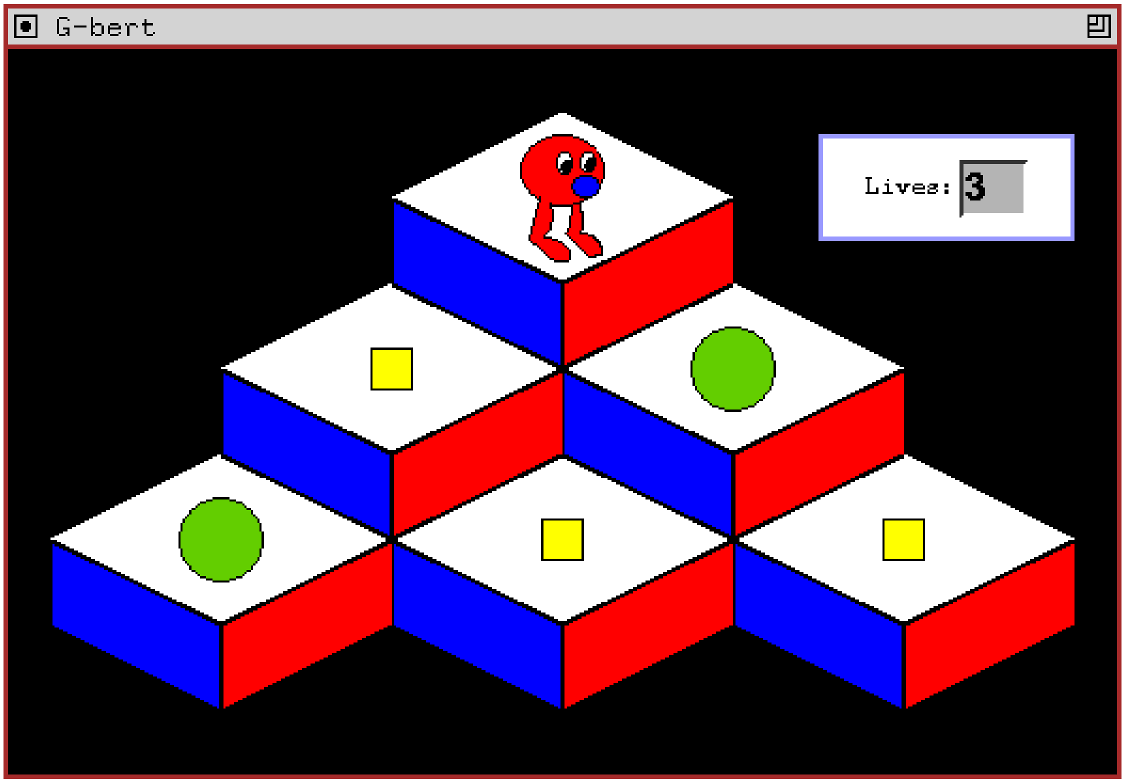
To start you off, we have already assembled the game board for you. You will have to make the pieces yourself, as well as demonstrate the game's entire behavior. Please note that although the board looks like a circle of squares, it is actually composed of lots of little pieces and Gamut cannot see the 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

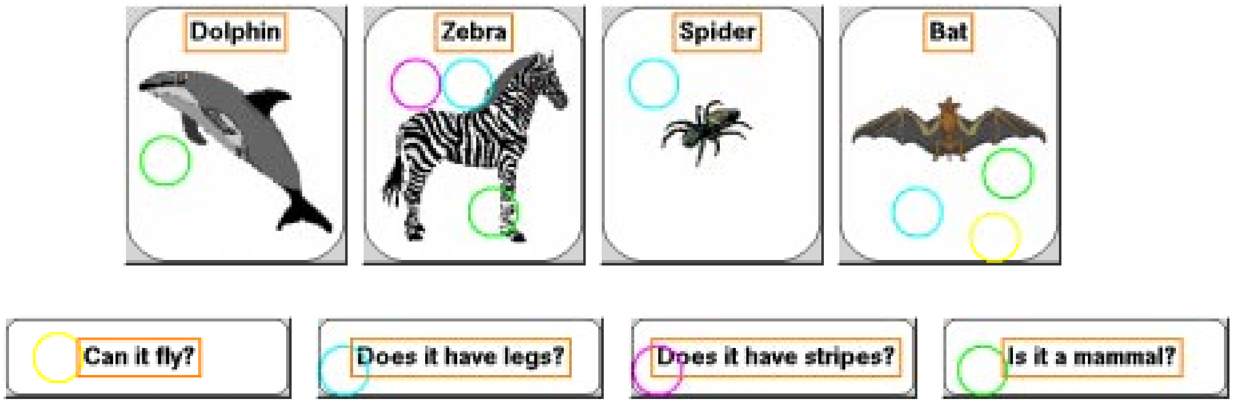
(2b) Make it more obvious what it thinks it's supposed to be doing

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

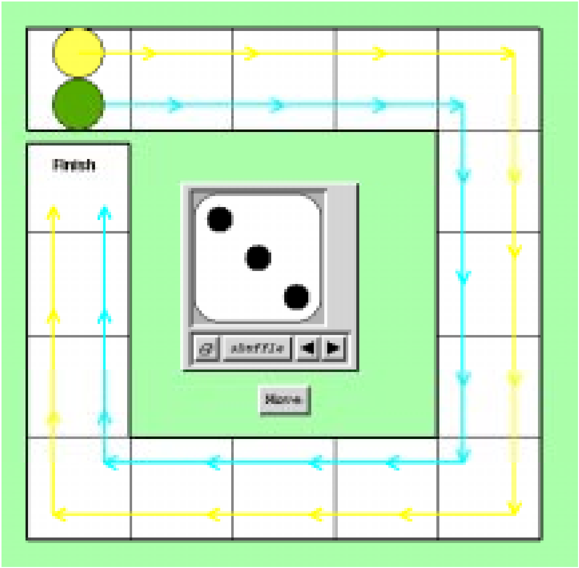
(4) Gamut Quick reference card for the palette.

E.2 Participant One's Saved Files

E.2.1 Task One: Safari

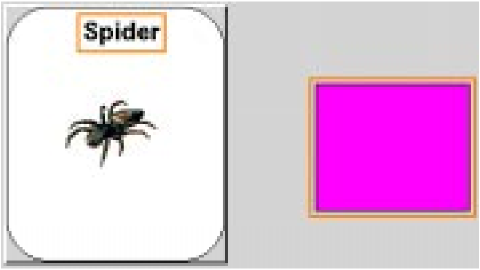


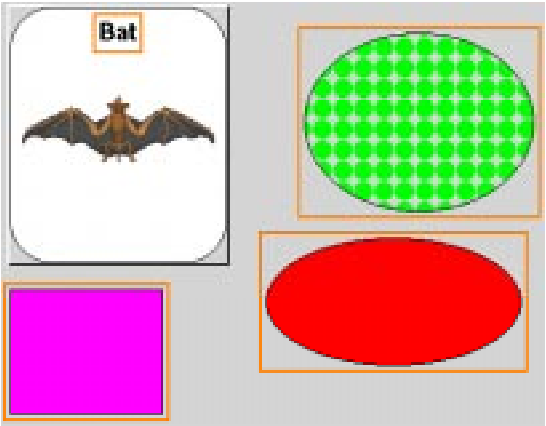
E.2.2 Task Two: Pawn Race

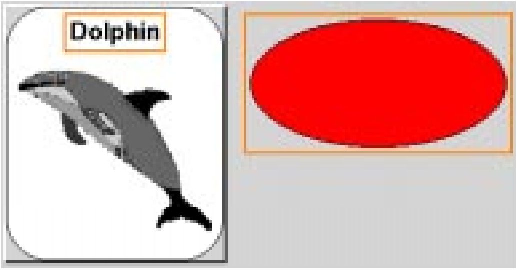


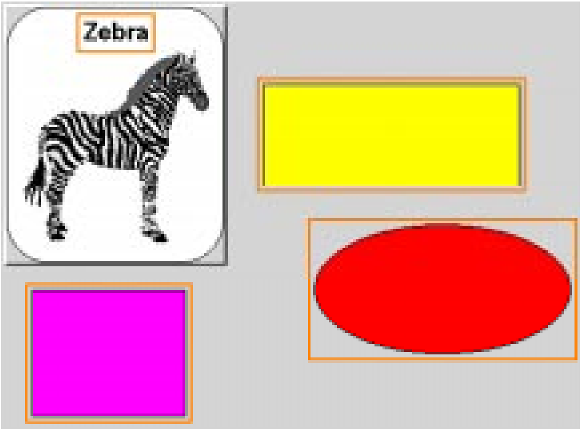
E.3 Participant Two's Saved Files

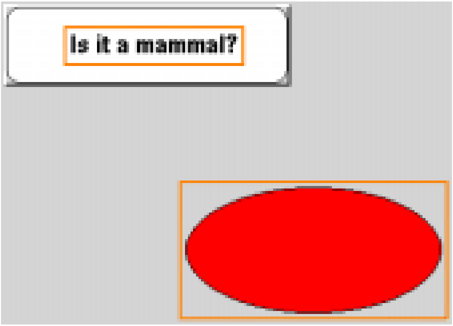
E.3.1 Task One: Safari

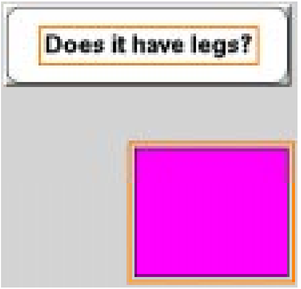


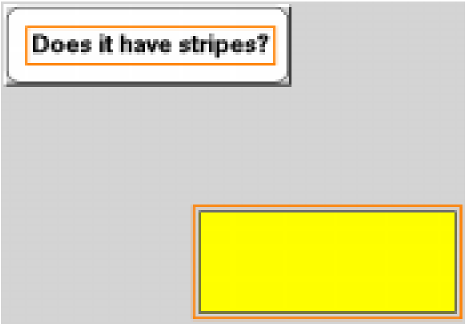


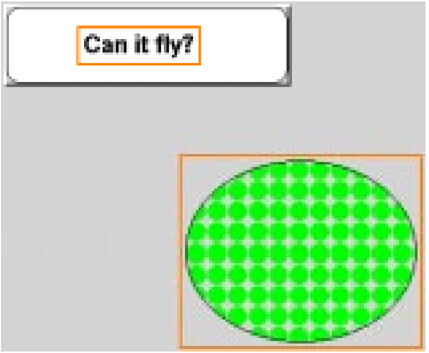




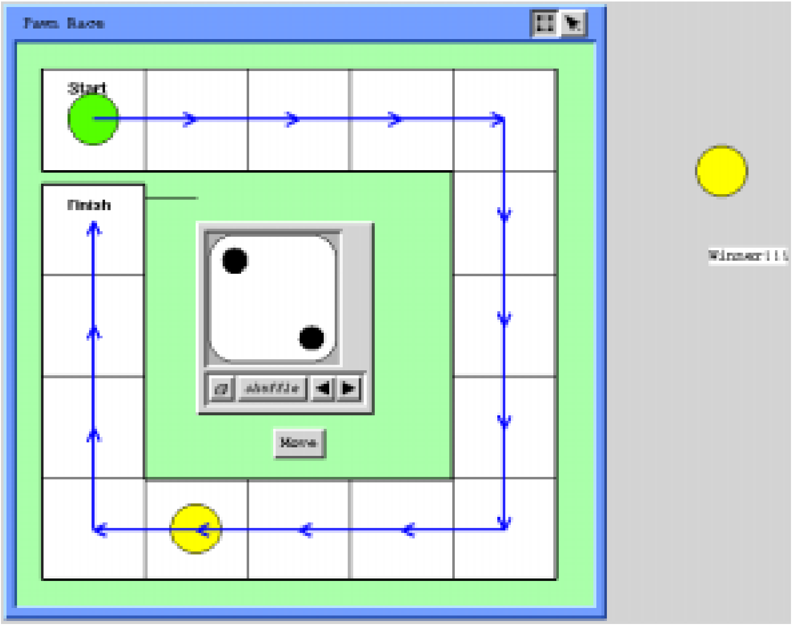




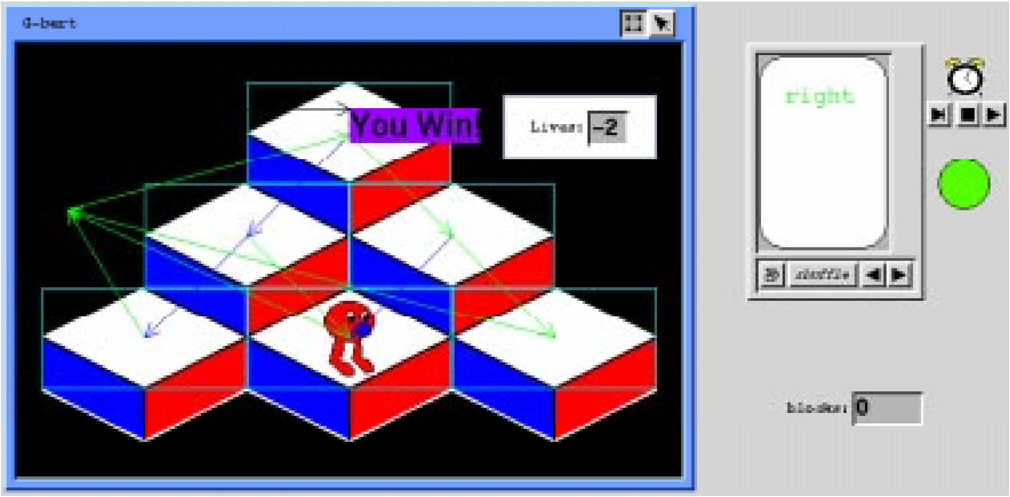




E.3.2 Task Two: Pawn Race



E.3.3 Task Three: G-bert

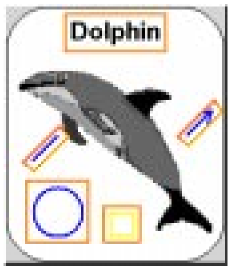


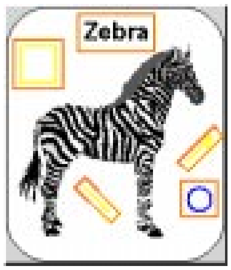
Lives: -2

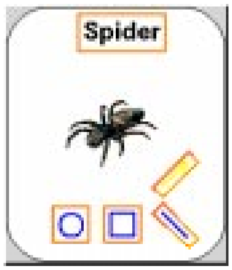
blocks: 0

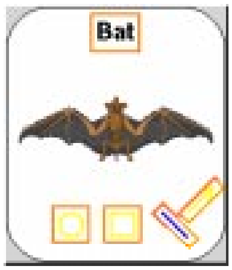
E.4 Participant Three's Saved Files

E.4.1 Task One: Safari



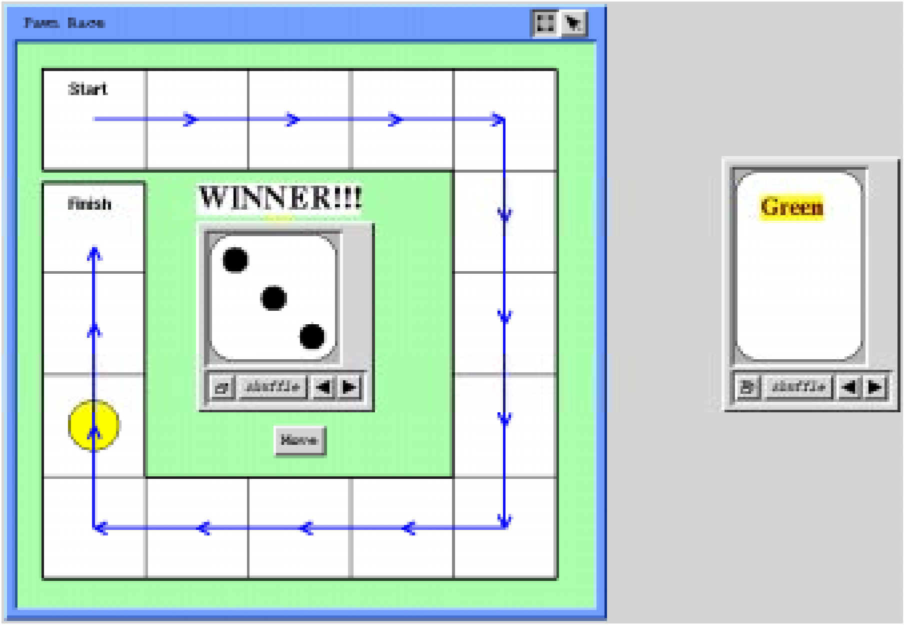






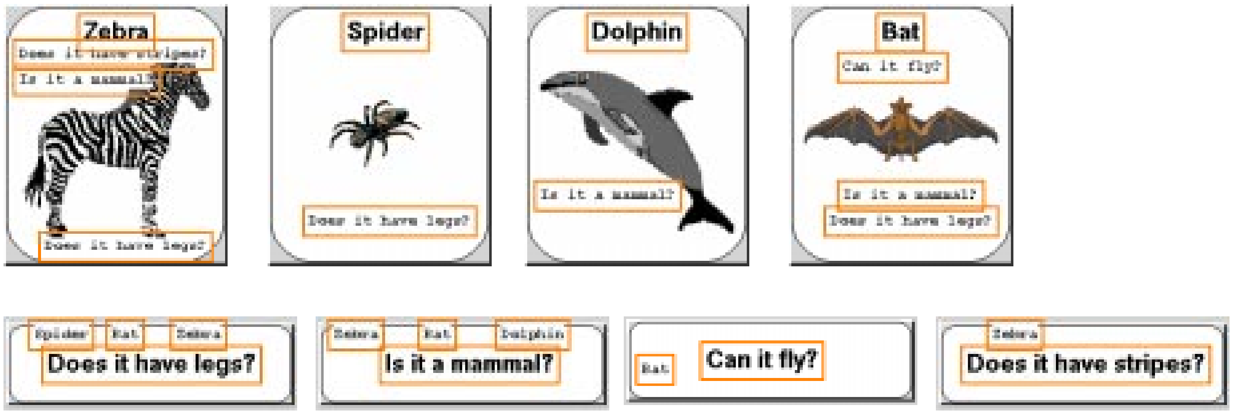
(participant 3 did not draw on the question cards)

E.4.2 Task Two: Pawn Race

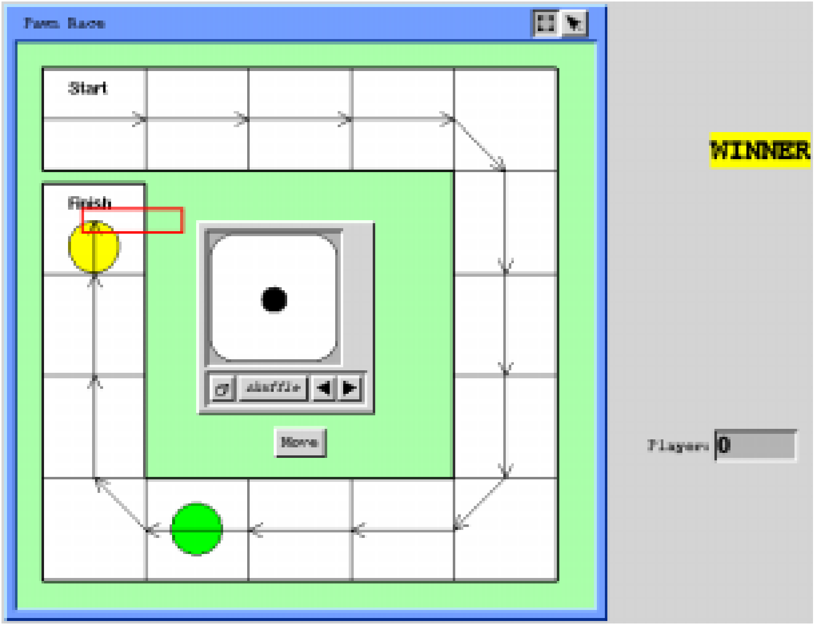


E.5 Participant Four’s Saved Files

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

Totally
Understandable
Method to
Assemble
Games

Great,
Another
Mind-numbing
User-interface
Tool

Green
Apples
Make the
Ultimate
Treat

Generating
Applications
Made
Unusually
Trivial

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

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.