Gamut: Demonstrating Whole Applications

Richard G. McDaniel and Brad A. Myers
HCI Institute, School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
Tel: 1-412-268-3066
E-mail: { richm, bam }@cs.cmu.edu

ABSTRACT

Gamut is a new tool for building interactive, graphical software like games, simulations, and educational software. A developer can build entire applications in Gamut's domain using only programming-by-demonstration (PBD) and never has to look at or modify code to build any behavior. To accomplish this, we have developed a simple, streamlined interaction for demonstrating so that developers can create new examples quickly and can specify negative examples without confusion. Also, Gamut allows the developer to give *hints* to point out objects in a relationship that would be too time consuming to find by searching. Gamut automatically revises generated code using an efficient algorithm that recursively scans for the differences between a new example and the previous behavior. To correct the discovered differences, Gamut couples heuristic search with a *decision tree learning* algorithm allowing it to build more complicated behaviors than it could using heuristic search alone.

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

INTRODUCTION

Building a graphical application like a game has traditionally been a difficult task. Even when using modern prototyping tools like Visual Basic, the amount of programming a developer must perform is considerable. In our research with Gamut, we have experimented with programming-by-demonstration (PBD) techniques that can eliminate the need to write code altogether. In fact, we have used Gamut to build several applications including maze games (Figure 1), Tic-Tac-Toe (Figure 2), and a Turing machine simulation. As we add more features to recognize strings and numbers, more applications including video games such as PacMan and educational software like Reader Rabbit become possible.

A PBD system like Gamut uses inductive learning to generate a program from example demonstrations. First, the developer draws the various interface components and then demonstrates their behavior by mimicking the desired program response to input events. The behaviors that Gamut can infer includes conditional relationships where actions are controlled by objects which are not necessarily affected by the actions such as behavior controlled by palettes and state variables. And it can infer chains of relationships where objects refer to other objects which in turn refer to others. For instance, Gamut can generate the description for how to move a piece in a game like Monopoly: "move the current player's piece to the square the dice's number of squares away from the location where it started."

The goal for this research has been to push demonstrational techniques to build broader and more advanced behaviors. We have developed interaction techniques that simplify the demonstration process and allow the developer to give the system hints. The interactions provide a faster way to create examples and allow negative examples to be specified as easily as positive ones. Also, we integrated a decision tree learning algorithm [3] along with heuristic search to increase the kinds of relationships that Gamut can infer. This allows Gamut to generate conditional behavior that other systems would need the developer to add to the code by hand.

DOMAIN

Gamut is designed to make graphical games similar to board games. These applications are two-dimensional and usually have a structure where there is a static background image with pieces that move on top. Video game behaviors like chasing monsters and flying bullets are accomplished with timer widgets that automatically move pieces. Gamut can learn the rules to games but does not create computer opponents. Inferring computer opponent for Chess is well outside Gamut's ability, but a developer can create a Chess game that keeps human players from making illegal moves. Likewise, monsters and other moving objects that behave in a predictable but not strategic way can also be created.

The behaviors of board games provide a number of challenges for a PBD system:

- Games have many modes and states. Conditional behavior which use these states can be complex. In PacMan, the monsters normally attack, but turn blue and run away when the player makes PacMan eat a big dot.
- Objects are often referenced not as constant literals, but as relationships to other objects. For instance, the dot that disappears is the one overlapped by PacMan's image.
- References to objects can form long chains. In the previously mentioned Monopoly move, the final destination square depends on the number on the dice, the geometry of the board, the piece's original position, and the state of the player turn indicator.

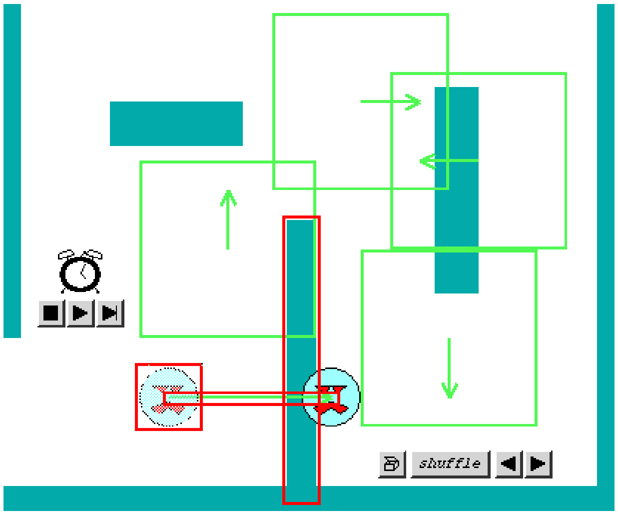

Figure 1: Here a monster (the circular face) is trained not to move through walls. The wall, the guide object arrow line the monster follows, and the temporal ghost of the monster are highlighted as hints. The four squares with arrow lines are cards which are put into the deck at the lower right to make the monster move randomly. The timer on the right causes the monster to move.

INTERFACE

We developed interaction techniques called nudges, hints, deck of cards, guide objects, and temporal ghosts.

Nudges: The developer uses two commands: Do Something! and Stop That! to create behavior. The commands "nudge" the system to make it do the right thing. The Do Something! nudge, tells the system that certain objects did not do what they were meant to do. The system asks the developer to manipulate objects to produce the correct result. The Stop That! nudge tells the system that a behavior was incorrect. The system undoes the actions affecting selected objects and considers that example as a negative example.

Hints: The developer can point out important objects by highlighting them (see Figure 1). When the system creates new behavior, it uses the hints to create code. By concentrating on the hinted objects at key times, the system can significantly reduce the amount of search it must perform.

Deck of Cards: Gamut uses a deck of cards metaphor as an important medium for storing and manipulating data. Developers can use the deck as though it were an actual deck such as the Chance deck in Monopoly, and they can use it to perform behavior involving randomness or lists of items. In Figure 1, the developer uses four cards to make a monster move randomly.

Guide Objects: Guide objects are objects that the developer can see but are made invisible when the application is run by a user. Guide objects originated in the Demo II system [1] and can be used to show alignments, directions, spacing, grouping, and other graphical relationships.

Temporal Ghosts: Quite often, the recent past state of objects is relevant to the behavior that acts on those objects. Gamut makes that state available by showing dimmed, translucent images of the changed objects (see Figure 1).

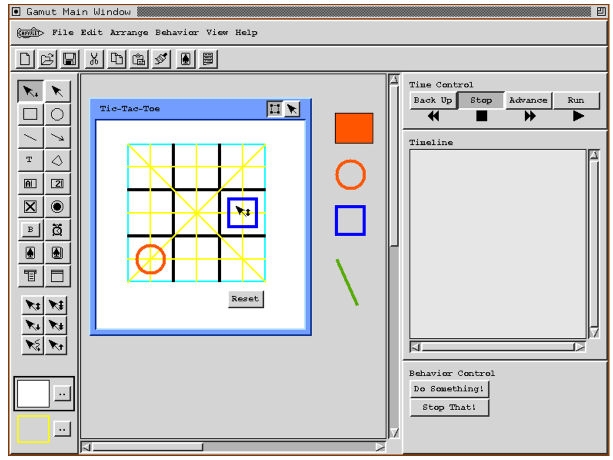

Figure 2: The Gamut main window while making Tic-Tac-Toe. Outside the window widget are prototype objects and an object to show the current player's turn.

INFERENCING

Gamut uses heuristic search to perform most of its inferencing. Rules are encoded as methods for a set of objects which represent code fragments. These rules search for the conditions where the fragments can be used. One set of methods is used to find the difference between a new example and the behavior that ran before. The difference methods are designed to find small changes in chains of descriptions like the Monopoly example. Once the differences are known, code generation can be applied to fix them.

It is not possible to have a built-in rule for every possible behavior a developer might want in a game. Therefore, some of Gamut's code fragments use decision tree learning [3] to form generic conditional expressions. Using a generic induction algorithm like decision trees allows Gamut to build behaviors that would be difficult using heuristic search.

IMPLEMENTATION

We built Gamut using the Amulet environment [2] and it runs on X Windows, Windows NT and 95, and the Macintosh. The remaining work includes implementing a few extensions and usability testing. We think Gamut shows that considerably more can be done with a PBD system than previously thought and we look forward to seeing how these techniques can help nonprogrammers build great software.

ACKNOWLEDGEMENTS

This research was partially sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326, and partially by NSF under grant number IRI-9319969. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

1. G.L. Fisher, D.E. Busse, D. A. Wolber. *Adding Rule-Based Reasoning to a Demonstrational Interface Builder.* Proceedings of UIST'92, pp. 89-97.

2. B.A. Myers, *et al*. "The Amulet Environment: New Models for Effective UI Software Development", *IEEE Transactions on Software Engineering*, to appear. http://www.cs.cmu.edu/~amulet.

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