Learning Games By Demonstration

Rahul Banerjee, Brandon Holt

December 13, 2012

Abstract

To enable the creation of simple 2D games without writing code, we propose a system that can learn the game logic from user traces demonstrating what should happen. Though our system is aimed at helping non-programmers design and build simple games, it could even be useful for game programmers to rapidly iterate through early prototypes while refining game logic. Learning programs from examples has been shown to be successful in specific domains where the space of possible programs is small and well-defined. We leverage domain knowledge of simple 2D games to compactly represent the state space and thereby allow fairly complex games to be expressed simply with a small vocabulary, and learned from a small set of user traces. We implement and evaluate a prototype system based on this approach and show three different games that can be built with it.

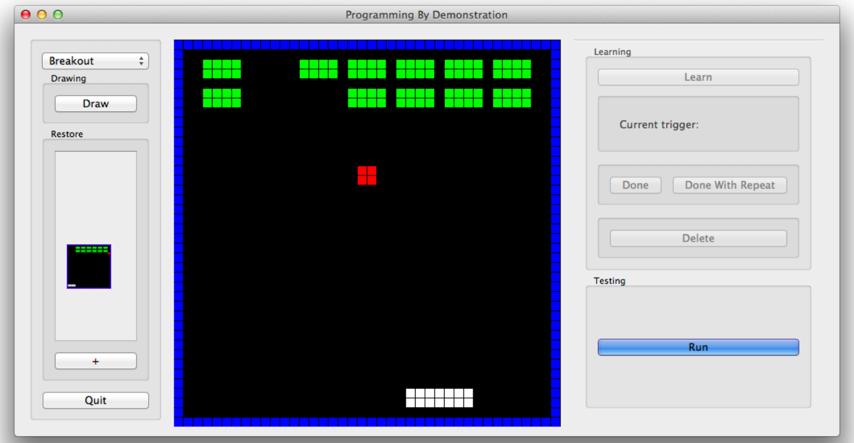


Figure 1: Teaching our prototype system how to play Breakout.

1 Introduction

Playing video games is fun and engaging, and building them, more so. However, not everybody knows programming well enough to create (or modify existing) games. Also, even programmers who want to rapidly iterate over game agent behavior would not relish the thought of coding up several programs or scripts to try out different scenarios. In our system, we aim to leverage Programming By Demonstration (hereafter referred to as PBD) to generate games by having the system infer the game logic from user-demonstrated examples.

1.1 Problem Statement

Let us consider the problem of implementing game logic for a 2D game in a non-scrolling world. Such games typically consist of:

- A 2D gameplay area,
- Objects in the game world, and
- Behavior for these objects. Some of them respond to the user's keypresses, others to collisions, while some are static (e.g. walls). They may respond by starting, stopping, or changing their motion, by dying or spawning, by changing a "score", by ending the game (win/loss), etc.

The system should display the gameplay area and objects to the user, then infer the behavior from demonstrations performed by the user, like moving objects around, deleting them, etc.

1.2 Related Work

PBD has been successfully applied in other domains prior to this work. Lau et al. [5] applied PBD to infer text-editing macros from user examples. They used Version Space Algebra (VSA) to efficiently represent and refine the hypothesis space (the set of all possible programs that are consistent with the user trace encountered so far).

Gulwani et al. [4] demonstrated a VSA-like approach for inferring Excel macros from input-output example pairs.

In [2], the authors leveraged domain knowledge to infer programs for common arithmetic operations by observing calculation traces. Our approach is closest to this work, where a small set of hand-selected program templates are fitted to user traces and the closest fit is deemed the best hypothesis.

There are other approaches to making game-building easier, like Stencyl [1], which is a set of customizable game templates, or Kodu [6], which is a GUI-based programming environment on the Xbox 360 aimed at young children. These systems present ways to define the behavior of a game, but neither use demonstrations or examples to generate any part of the game.

Figure 2: Main game loop.

```
while (GameRunning):
    # event handling
    for trigger in detectTriggers(gameState):
        action = LearnedActions[type(trigger)]
        action(trigger, gameState)
    # step each object according to its current behavior
    for o in gameState.gameObjects:
        o.update()
```

1.3 Overview

The rest of this report explains how we use PBD for learning games, followed by a description of our prototype implementation. We then give results from our pilot user study, provide directions for future improvement and round off with conclusions.

2 Approach to learning games

The basic idea behind any learning by demonstration system is to define a limited domain-specific vocabulary and grammar that will be able to encode useful programs but will be simple enough to learn with few examples. We leverage knowledge of how simple 2D games behave to determine how to represent games and define the space of learnable behaviors.

2.1 Game representation

We recognize that simple games typically consist of a number of independent objects that interact in the game world. The game state (`gameState`) at time t consists of a set of individual game objects $\{o_1, ..., o_n\}$.

Game objects can be thought of as finite state machines where each state has (at least) a position ((x, y) coordinates in our 2D game world) and behavior b, though they could be extended with arbitrary data. An example of a behavior is one that updates position according to a velocity. Typical game objects include walls, food pieces in Pacman, or the user-controlled “paddle” in Pong.

The program running a game is basically a combination of a game-object update loop and an event-handler that modifies game objects based on detected events. Each iteration of this game loop corresponds to advancing *game time*. *Behaviors* determine how all of the game objects are updated each time step (how each state machine is advanced). Game objects and the user of the game interact through *triggers*, which act to modify the behaviors of game objects. Figure 2 shows the main game loop that is used to run the game. This game logic is fixed. The “logic” for a particular game is learned by adding and updating the `LearnedActions` lookup table as detailed below. Though not implemented in

Figure 3: Game Representation

Parameterized trigger types:
(τ is an object type)

- $initialize < \tau >$
- $keypress < key >$
- $blocking\ collision < \tau_1, \tau_2 >$
- $touching\ collision < \tau_1, \tau_2 >$
- $separate\ after\ touch < \tau_1, \tau_2 >$
- ...

Actions available for learning:

- $replace(object, (x, y))$
- $bounce(object, \{x|y\})$
- $spawn(new\ object)$
- $delete(object)$
- $addPoints(n)$
- $winGame()$
- $loseGame()$

our system, a "level editor" is also necessary to allow creation of arbitrary game objects and level layouts.

2.2 Game vocabulary

Before explaining how game logic is learned, we must define the space of actions that game objects can take and under what circumstances they interact with each other and the user.

Triggers, as mentioned already, correspond to events that may occur while running the game, such as a key being pressed, or two objects colliding. The `LearnedActions` table is a map of trigger *types* to actions. This is to allow the system to learn a $trigger \rightarrow action$ pair for a single instance and generalize to the rest (e.g. a ball should bounce off of any wall tile). Trigger types are parameterized by other types depending on the kind of trigger. For instance, collisions are parameterized by the types of the objects colliding to allow different actions to be learned. A full list of triggers can be found in Figure 3.

Actions modify the state of game objects. Most often this means modifying the behavior of an object, such as *bounce* which inverts one component of an object's velocity if it is moving. Other available actions change the overall game state, such as deleting objects or ending the game as a loss.

When a trigger is detected for a game state, it is associated with specific object instances. This allows the game loop (Figure 2) to lookup the learned *action* and apply it the specific object instances the trigger was detected for. A completed action table defines how all of the game objects interact, making up a complete game. While this vocabulary restricts the set of possible games we can build, it also constrains the search space in a way that lends itself to learning by demonstration.

2.3 User Demonstration

Our system allows the user to demonstrate game object behavior, including:

- Motion or change therein (including stopping): The user drags the object to indicate the desired motion. Stopping is inferred from a no-movement drag.

Figure 4: Learning Algorithm

```
userTrace = <selected object trace, got via demonstration>
trigger = <trigger that initiated this user demonstration>
MatchedActions = <empty list>
for action in KnownActions:
    newGameState = copy of gameState
    # applying the action to the game state
    # gives us the new behavior
    action(trigger, newGameState)
    newBehavior = <behavior of object within newGameState>
    newTrace = <trace generated by executing newBehavior>
    # if it matches, then this is a candidate action
    if Similar(userTrace, newTrace):
        MatchedActions.append(action)
return MatchedActions
```

- Deletion of game objects: The user clicks the “Delete” button, then clicks on the object that must be removed.
- Game over (win/loss) and score changes: These are not implemented yet, but we plan to have dedicated buttons for them.

2.4 Learning From Traces

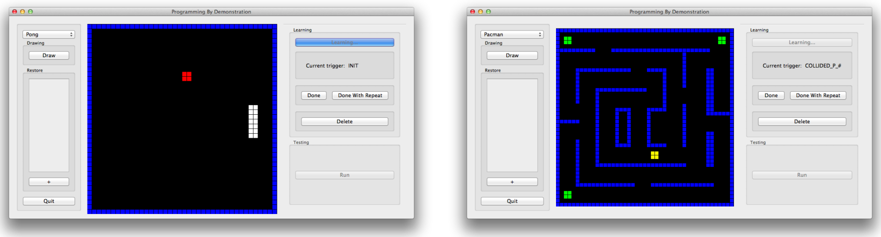
In response to some trigger, having recorded the motion or deletion traces from the user, our system compares the existing behavior against the new (user demonstrated) behavior. Starting from a small candidate set of actions (for motion, deletion, etc), we find the ones that transform the old behavior to match the new one, then pick the most general one (according to a hand-picked generality ordering).

In future implementations, we plan to maintain all candidates (instead of picking one and discarding the rest). In that case, incorrect inferences can be corrected by demonstrating more traces, and successively shrinking the hypothesis space to only include the actions consistent with all examples. Since the UI is currently very restricted, and since this system was built in a very short time, we chose to pick only the best candidate action. Our learning algorithm is listed in Figure 4.

3 Prototype implementation

We implemented a simplified version of the above algorithm, with a functional UI using PyQt, incorporating certain simplifications due to time constraints.

Figure 5: Learning games from demonstration in our prototype.



(a) Pong (b) Pacman

(c) Breakout (shown in Figure 1)

3.1 On-demand demonstration

The current design takes a lazy approach to demonstrating *trigger* → *action* pairs. All objects begin with no movement (i.e., no behavior), and `LearnedActions` starts out empty. While running the game, whenever a trigger happens that does not map to an action yet, the game pauses and prompts the user to demonstrate what should happen. The user indicates completion of a demonstration explicitly by clicking a "done" button. Alternatively, if the intention is that a demonstrated action would continue indefinitely (such as continuing the motion of a flying ball), then the user indicates that by selecting a different button: "done with repeat". This gives our system a well-segmented trace, making the job of recognizing actions as simple as possible. Further, filling in `LearnedActions` lazily allows the user to "run" the game even though it is only partially learned.

3.2 Choosing the best candidate action

As Figure 4 shows, the set of candidate actions need not be a singleton. However, the only way to actually run the game would be to choose a particular action, or run all actions in parallel until they diverge. Ideally, we'd have a more capable UI which would allow the user to demonstrate multiple traces, or go back and correct the system when its playback is different from the user's expectations. In the absence of such a UI, we decided to rely on a generality ordering and pick the most generic action that would fit the given example.

3.3 Similarity of traces

There are several ways to compare traces. For deletion, making sure that objects deleted in both traces are of the same type, and have similar relative positions (e.g. touching the player) is sufficient. For motion, the user will not be able to draw the exact same path twice. So we constrained the user's click-drag motions to be discretized to the 2D game world's underlying grid. This made comparison of motion trajectories much more robust.

4 Evaluation

We evaluate our prototype with an informal user study. The goal of the user study is not to evaluate the user interface in particular, but rather to see if people besides ourselves are able to understand and use our model for learning games.

4.1 Method

Because our focus is not on having an intuitive user interface, our study focuses on getting participants comfortable enough using our prototype to give feedback about the overall approach to learning games. For each participant we explained how the game will be represented (by a map of triggers to actions) and how the user demonstrates what should happen in order for the system to learn these mappings. We then demonstrated how to make simplified versions of Pong and Pacman (as in the class presentation). Following our tutorial, the participants were given the pre-made board for Breakout and asked to demonstrate how to play the game. Requirements were that the paddle be manipulable by keys, the ball moved diagonally and bounced correctly, and the bricks were deleted when the ball hit them. Participants were encouraged to ask questions at any time during the tutorial or as they worked on teaching their own game.

4.2 Results

We found three Computer Science grad students to participate in our study. Two of them work at least partly on game-related research, one on machine learning and artificial intelligence. Despite biasing us toward users who had prior experience in this area, it resulted in providing us with interesting feedback about our approach.

All three participants were able to teach Breakout successfully, though two had to start over once because of a mistake that could not be corrected due to bugs in our implementation. Interestingly, each of the participants came up with a slightly different way to manipulate the paddle (directly with “left” and “right” controls, with “lazy” left and right that keep going after releasing, and with a single “toggle” direction). We believe this demonstrates a degree of flexibility in our system–more than one game can be made even with the same set of starting objects.

Overall, feedback was positive. The participants were able to look beyond the limitations of the current prototype and ask insightful questions about the design we envision. Participants asked many questions to help them understanding how the system learns behaviors before they felt comfortable trying themselves.

An issue came up a couple of times about how behaviors are generalized and which objects a demonstration applies to. In our current implementation, all instances of a type of object, such as walls, must have the same action. The long-term goal would be to incorporate multiple examples to disambiguate behavior and re-partition classes of objects that should behave in different ways.

However, this limitation caused confusions because it was not always clear why actions would generalize in some cases but not all. It was suggested by one participant that this could be made simpler by highlighting related objects when learning, and allowing behaviors to be copied between different objects.

The distinction we make between "done" and "done with repeat" required the most explanation and caused hesitation when demonstrating actions. With support for learning more general actions and learning from multiple traces, the need for such as distinction may go away. In particular, this distinction is closely tied with the interaction mechanism implemented for this prototype and could be completely different in another system such as will be described in the future work section below.

We also received a suggestion that we get ideas about interaction models from a low-fidelity "paper prototype". Participants would be asked to demonstrate a game to someone using actual paper cutouts of the game board and objects (possibly with the restriction that they cannot talk). From observing how these demonstrations were done, how ambiguous cases were handled, etc., we may come up with novel interaction schemes.

5 Future

While the simple prototype implemented so far can actually learn games, the interaction mechanisms are unintuitive and the set of games that are possible to learn is much smaller than our proposed model could support. Future work could extend this functionality in a number of interesting ways.

5.1 Advanced learning

Even if the proposed learning model was fully implemented, it would not support learning complicated compound actions. A more advanced learning algorithm could do recursive matching to learn more complex actions from the simple vocabulary we described. Further, by adding a concept of learning how objects relate (spatially or otherwise) to each other, pattern-based triggers could be learned as well. In the simplest case, this could allow us to learn multi-body behavior like in Conway's Game of Life, but would significantly add to the complexity of the search problem. To handle the exponential explosion of state, future work would most likely need to use VSA or similar techniques to keep the problem tractable.

5.2 Continuous space

One of the most frustrating aspects of using the prototype from this work is the discrete grid that objects are confined to moving in. It makes demonstrating gestures difficult and reduces the quality of the output. Allowing continuous movements is preferable, but making motions look natural and reproducible requires intelligent smoothing. The goal of smoothing trajectories would be to come up with the "simplest" explanation for the curve, whether it's a straight line or regular curve of some sort. This is basically an extension of learning the most general explanation for a behavior, similar to some related work on solving constraints to choose the best-looking construction when drawing geometry [3].

5.3 Timeline interaction

The "on-demand" demonstration used in this work does not allow learning to be refined by demonstrating more than one example. One possibility for future work would be to incorporate the idea of a "timeline" control to allow the user to move around through the trace. While the timeline is at the end, the game runs completely from its `LearnedActions` table. However, if time is "rewound", the trace is replayed instead. While "back in time", any actions the user demonstrates that diverge from the recorded trace can be assumed to be teaching new behavior (or refining existing learned behavior).

5.4 Physics-based learning

The current design assumes nothing about the way objects behave; even collision with walls can be bypassed easily. However, another property of many games is some semblance of physics, such as gravity in platformers. This gave us the idea that adding "physicality" constraints and learning physics-based actions may result in realistic-feeling games much more easily. For example, a "jump" action in a Mario-like game could be learned as just an impulse on the Mario character, and the rest of the way Mario behaves, such as landing on platforms and being blocked by walls happens automatically. Learning the "most physically accurate" action is another kind of generality ordering that could help simplify teaching complex games easily.

6 Conclusion

This work applies the concept of programming by demonstration to a new domain: game design. We leverage knowledge about game programming to narrow the problem space to where it is possible to learn game logic from a small number of examples. Even with a very small vocabulary, limited learning capability, and a simple interaction model, we were able to successfully build a few different 2D games. Designing other games than are shown in this paper should only require making new game objects and game boards for them. Building this prototype allowed us to debug the design of the game representation and learning model on concrete examples and allowed us to show the idea to people and get feedback. This work can inform another iteration of the design that has a more capable user interface, larger vocabulary of actions and triggers, and expanded learning capabilities.

A Appendix

A.1 Who did what?

Initial ideas and high-level design were done by Rahul for his own research prior to the class project. Both Brandon and Rahul were involved in sketching out the triggers and actions vocabulary and working out the details of the learning systems.

The prototype implementation was created for the AI class project. Rahul and Brandon both worked on the Python code, with Rahul focusing primarily on the learning modules and Brandon doing most of the GUI maintenance.

A.2 Code

Code for the project is included in the submission. It consists of a small number of Python modules that define the learning system and make up the graphical user interface. The only external code we use is the PyQt framework for our cross-platform GUI, which can be downloaded from here: `http://www.riverbankcomputing.com/software/pyqt/download`. With PyQt installed, the program can be run with the command:

```
python mainapp.py
```

From within the GUI, different starting boards can be loaded with a dropdown menu on the left. Learning controls are on the right side. *Warning: this code is a very rough prototype and is likely to break often during use. Please let the authors know if you encounter problems running it and would like some help.*

References

[1] Stencyl. `http://stencyl.com/`.

[2] E. Anderson, S. Gulwani, and Z. Popovic. Programming by demonstration framework applied to procedural math problems. (Unpublished).

[3] S. Cheema, S. Gulwani, and J. LaViola. Quickdraw: improving drawing experience for geometric diagrams. In *Proceedings of the 2012 ACM Annual Conference on Human Factors in Computing Systems (CHI 2012)*, pages 1037–1064. ACM, 2012.

[4] S. Gulwani, W. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[5] T. Lau, S. Wolfman, P. Domingos, and D. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1):111–156, 2003.

[6] M. MacLaurin. The design of Kodu: a tiny visual programming language for children on the Xbox 360. In *ACM Sigplan Notices*, volume 46, pages 241–246. ACM, 2011.