Modeling the Behavior of Pac-Man Non-Player Characters Using Finite State Machines

Braanah Craig, Nicklaus Krems, and Harsha Kuchampudi

***Abstract*—This paper discusses an approach to modeling the behavior of Pac-Man Non-Player Characters (NPCs) using deterministic finite state machines and then visualizing the state machines in real-time by using the JavaScript data visualization library, Cytoscape.js.**

I. Introduction

FINITE state machines (FSMs) are mathematical models that can be implemented in hardware and software that help simulate sequential logic. When designing video games, finite state machines in the form of deterministic and non-deterministic finite automata are particularly useful for designing simplistic artificial intelligence (AI) models or for programming the flow of events in a game [2]. In the original Pac-Man, deterministic FSMs were used extensively to model the behaviors and actions of the ghost NPCs [2]. Although FSMs play a critical role in the Pac-Man game logic, there are few, if any, applications that attempt to model how the NPCs move and act in accordance with their implicit state machine based programming. For this reason, in addition to demonstrating how FSMs can be used in game development, that developing a real-time deterministic FSM visualization was especially compelling.

II. Game Logic

THE Pac-Man game contains five characters. One character is controlled by the player (Pac-Man) and the remaining four characters are NPCs controlled by the computer (the Ghosts). The goal of the game is for the player to maneuver Pac-Man through the generated maze and consume all the dots (known as "Pac-Dots") on the screen while avoiding running into the four ghosts. When all the dots are consumed, the player is able to move onto the next stage.

A. Properties of Ghost Non-Player Characters

During each stage, the four ghosts (named "Blinky", "Pinky", "Inky", and "Clyde") are programmed to wander the maze with the objective to kill Pac-Man. If Pac-Man touches a ghost, it loses a life and the stage resets with the ghosts being sent back home and Pac-Man is reset to the starting position. When Pac-Man loses all of its lives, the game is over. During the game, if Pac-Man consumes one of the flashing dots (known as "Power Pellets") on the board, the ghosts turn blue and run away from Pac-Man. If Pac-Man touches a ghost while it is blue, the ghost is "eaten" and teleports back home where it will respawn after some time. Based on these properties of ghosts, a deterministic FSM model is generated. There are 7 states that ghosts can be in:

- **Spawn:** The spawn state is where the ghosts are initialized prior to the game starting, or when Pac-Man dies and the game is reset, or when Pac-Man consumes one of the ghosts and the ghost respawns inside the home.
- **Idle:** The idle state occurs when the ghost is deciding what direction it should move in while it is not afraid.
- **Moving (from Idle):** The moving (from idle) state occurs when the ghosts finish making a decision on what direction to move in and then move in that direction. A ghost must not be afraid for this transition to take place.
- **Afraid:** The afraid state occurs when Pac-Man consumes one of the flashing dots. Ghosts will remain in this state until the effects of the Power Pellet wears off after a fixed amount of time. In this state ghosts decide what direction they should move in (to get away from Pac-Man).
- **Moving (from Afraid):** The moving (from afraid) state occurs when the ghost has finished making a decision on what direction to move in to get away from Pac-Man, and then moves in that direction.
- **Eaten:** When Pac-Man touches a ghost while it is in the afraid state (i.e. the ghost is blue due to the effects of the Power Pellet) the ghost is considered to be "eaten" and moves into the eaten state.
- **Teleporting:** This state takes place after a ghost is eaten and is then being transported back home. Once the ghost reaches home and re-materializes, it will respawn in the spawn state.

Using the above states in conjunction with the know behavior of the ghosts, the deterministic finite automata (DFA) in *Figure 1* is generated to model the ghosts' behavior.

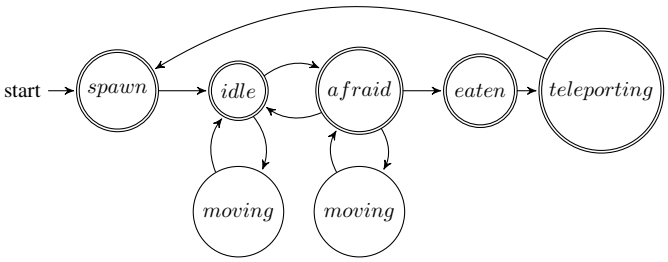

Fig. 1. Deterministic Finite State Machine for the Ghost NPCs

III. Modeling the Behavior of NPCs

AFTER designing the finite state machine for the ghosts, the next steps were to codify the sate machine, create a vizualization for the state machine, and link it to a live Pac-Man game so users can easily visualize how ghosts move through the state machine over the course of the game. For ease of accessibility, we designed our data visualization using HTML, CSS, and JavaScript. In order to create the data visualization there were four main tasks:

1) Obtain a copy of the Pac-Man game, preferably a copy of the game that has been ported to JavaScript.
2) Create a data abstraction layer so that inportant information could easily be polled and extracted when necessary.
3) Codifying the designed state machine described in *Figure 1* in JavaScript.
4) Creating a visualization for the state machine in HTML and CSS so end-users can see how ghosts move through the FSM.

The above steps along with our implementation are discussed in detail in the following sections.

A. The Pac-man Codebase

For our approach, instead of recreating Pac-Man or developing linking software with the original game released by Namco, we elected to model Lucio Panepinto's JavaScript clone of Pac-Man [3]. Panepinto's clone closely models the characteristics of the original game.

B. The Data Abstraction Layer

After obtaining a clone of Pac-Man, it was necessary to modify the source code so that movement data, along with state information could easily be monitored and extracted when necessary. In the source code (specifically in `js/ghosts.js`), Panepinto used globally scoped variables to monitor information about the ghosts' states, movements, and position:

```
// (GHOST_NAME) is a placeholder for the name of a
// specific ghost (i.e. BLINKY, PINKY, INKY, or
// CLYDE). The following set of variables are
// for each of the four ghosts.

// Tracks the ghost's X coordinate on the board
var GHOST_(GHOST NAME)_POSITION_X
// Tracks the ghost's Y coordinate on the board
var GHOST_(GHOST NAME)_POSITION_Y
// Tracks the direction the ghost is moving in
var GHOST_(GHOST NAME)_DIRECTION
// Tracks whether the ghost is moving
var GHOST_(GHOST NAME)_MOVING}
// Tracks the ghost's state (i.e. if it is Afraid,
// Alive, or Eaten)
var GHOST_(GHOST NAME)_STATE
// Tracks if the effects of the Power Pellet are
// wearing off and the ghost is almost not afraid
var GHOST_(GHOST_NAME)_AFFRAID_STATE
```

In order to ease the process of polling this information, it was necessary to design the `updateGhostDebugInfo` wrapper function (found in `js/ghosts.js`) which polls all the information at once and decodes the values into meaningful information.

C. Codifying State Machines for Non-Player Characters

After building the function to poll information about the ghosts' states, it is necessary to codify the state machine described in *Figure 1* so that the ghosts' actions in the game can be directly translated to transitions between states in the state machine. In order to codify the state machine, the *javascript-state-machine* library created by Jake Gordon was used [1]. In order to properly use the library, it was necessary to define a generic "factory" class that could be used to define state machines for each of the four ghosts. Within the class, the initial starting state is defined as the spawn state.

```
var Ghosts = StateMachine.factory({
  init: 'spawn',
  ...
});
```

After declaring the initial starting state, it was also necessary to define the states and valid state transitions for the state machine as defined in *Figure 1*.

```
...
transitions: [
      {
        name: 'spawn_to_idle',
        from: 'spawn',
        to: 'idle'
      },
      {
        name: 'idle_to_moving',
        from: 'idle',
        to: 'idle_moving'
      },
      {
        name: 'moving_to_idle',
        from: 'idle_moving',
        to: 'idle'
      },
      ...
  ],
  ...
```

After defining all the valid states and transitions for the state machine, it was necessary to add a data constructor for the class that would identify which ghost a partifular FSM was attached to.

```
...
data: function(ghost) {
  return {
    name: ghost,
    name_upper: ghost.toUpperCase()
  };
},
...
```

Finally, after finalizing the ghost factory class, a state machine was created for each of the four ghosts by creating a new `Ghost` object.

```
// Create state machines for each of the ghosts
var blinkyFSM = new Ghosts('blinky');
var pinkyFSM = new Ghosts('pinky');
var inkyFSM = new Ghosts('inky');
var clydeFSM = new Ghosts('clyde');
```

Finally, each of the generated state machines need to be linked to each ghost's actual state in the game. In order to accomplish this, a middleware function was written so whenever the information of a ghost is polled, the corresponding FSM also gets updated. The middleware function (`updateGhostFSM`) is called at the end of the `updateGhostDebugInfo` function mentioned in **Section B**. The `updateGhostFSM` function compares the current state of the ghost in the game to the ghost's present state in the corresponding FSM. If the states are different, the state machine triggers a transition to the correct state. At this point, the state machines for each of the four ghosts correctly align with the states of the ghosts in the game.

D. Creating a Visualization for the State Machines

The final step, was to create a live visualization for the codified state machine. In order to create the visualization, the Cytoscape.js graph theory and network visualization library was used [4]. The code for each of the ghost FSM visualizations is found in the `js/viz` folder and configures each state as a node and describes the edges which connect nodes.

```
// Blinky's network graph
var blinkyGraph = cytoscape({
  ...
  // Elements in the Graph (i.e. states/edges)
  elements: [
    // States in the FSM
    { data: { id: 'spawn', label: 'Spawn' } },
    { data: { id: 'idle', label: 'Idle' } },
    ...
    // Edges in the FSM
    {data: { id: 'spawn_to_idle', source: 'spawn',
   target: 'idle' }},
    {data: { id: 'idle_to_moving', source: 'idle',
   target: 'idle_moving' }},
    ...
  ],
  ...
});
```

After configuring the visual layout of the network graph representing each FSM, the final step was to link the FSM visualization with the codified FSM described in **Section C**. When the codified FSM makes a transition to a different state, the corresponding state/transition in the visualization changes color to visually alert end-users that a state transition has taken place. In order to design this behavior, it was necessary to add some additional methods to the state machine factory described in **Section C**. The added methods are `onTransition` and `onEnterState` which are listening hooks that trigger when the codified FSMs detect that a transition to another state is taking place and changes the corresponding node/edge color to green.

```
// Class for Ghost state machines
var Ghosts = StateMachine.factory({
  ...
  // Functions/Watchers for the state machine
  methods: {
    // On the Transition
    onTransition: function (lifecycle) {
      var ghost_name = this.name;
      var ghost_tran = lifecycle.transition;
      ...
          // Change the arrow's color to green
          eval(`${ghost_name}Graph.elements("edge#${
    ghost_tran}")[0].style("line-color", "green")`);
          eval(`${ghost_name}Graph.elements("edge#${
    ghost_tran}")[0].style("target-arrow-color", "
    green")`);
          setTimeout(changeBackEdgeFunc, 1000,
    ghost_name, ghost_tran);
      ...
    },
    // On Entering a State
    onEnterState: function (lifecycle) {
      var ghost_name = this.name;
      var ghost_state = this.state;
        ...
        // Change the node's color to green
        eval(`${ghost_name}Graph.elements("node#${
    ghost_state}")[0].style("background-color", "
    green")`);
        setTimeout(changeBackNodeFunc, 1000,
    ghost_name, ghost_state);
        ...
    },
    ...
  }
  ...
});
```

Now, the data abstraction layer, codified state machines, and the data visualization components are all working together to produce a real-time representation of the in-game ghosts' states.

IV. Conclusion

In conclusion, the generated data visualization demonstrates how components of Pac-Man the underlying game logic can be represented using FSMs. In a broader scope, this visualization is valuable because it helps deconstruct how games can be seen through the lens of FSMs. This method for visualizing integrated, implicit state machines could prove useful for modeling other relatively simple FSMs.

References

[1] J. Gordon (2018) javascript-state-machine (master) [Source code]. https://github.com/jakesgordon/javascript-state-machine

[2] M. Mateas. Expressive Intelligence: Artificial Intelligence, Games and New Media. Lecture Notes in Computer Science AIIA 2007: Artificial Intelligence and Human-Oriented Computing, 2–2. DOI:http://dx.doi.org/10.1007/978-3-540-74782-6_2

[3] L. Panepinto (2018) Pacman source code (master) [Source code]. https://github.com/luciopanepinto/pacman

[4] P. Shannon et al. 2003. Cytoscape: a software environment for integrated models of biomolecular interaction networks. (November 2003). Retrieved July 11, 2018 from https://www.ncbi.nlm.nih.gov/pubmed/14597658