Project 3: State Machine Game

Due Dates

Part 1: 2015-10-25: 11:59pm
Part 2: 2015-11-1: 11:59pm

In this assignment you will be implementing parts of a rudimentary game engine that is implemented using a table-based finite state machine. This programming assignment is one demonstration of the value of finite state machines for managing user interfaces. The purpose of building a finite state machine is to simplify the task of building complex interactive elements such as resizable and draggable options. Because interactions with elements such as resizable icons occur over time, interactive elements must save their current state to reflect this input over time.

Files Provided

- game.js (/SSUI-Web-2015/assets/assignments/p3/game.js) - JavaScript file describes that serves as the game engine, and is the root of the game.
- actor.js (/SSUI-Web-2015/assets/assignments/p3/actor.js) - JavaScript file that should hold an actor's information and its FSM
- actions.js (/SSUI-Web-2015/assets/assignments/p3/actions.js) - JavaScript file that has a list of predefined actions for use in the FSMs.
- statemachinetest.html (/SSUI-Web-2015/assets/assignments/p3/statemachinetest.html) - HTML file that goes with statemachine_test.js
- statemachine_test.js (/SSUI-Web-2015/assets/assignments/p3/statemachine_test.js) - Tester "game" for you to start from
- icons.zip (/SSUI-Web-2015/assets/assignments/p3/icons.zip) - A set of icons to use as the actor images
- p3.zip (/SSUI-Web-2015/assets/assignments/p3/p3.zip) - All of the above, in a zip file.

Project Overview

You will be creating a Game object that models a game using state machines on an HTML5 Canvas. The game is composed of several actors, which each has their own state machine. Each state machine will react to events sent to the Canvas element, so the game engine has to appropriately dispatch events sent to it. The Actor Javascript object will contain actor information, as well its states and transitions. The start state of your state machine will be the first state in your state list. When input occurs and applies to an object, the state machine will need to transition to a new state according to the input, and execute any functions necessary.

Your state machine should be able to take in any valid state machine specification and behave correctly according to the specification.

StateMachine Specification

For this project, an actor's state machine will be described as a JavaScript object (in JavaScript Object Notation, or JSON). The state machine for this assignment will be described as a list of states. You can assume the states have unique names. Each state has a list of outgoing transitions (which you can also assume are unique; you won't have two transitions both from A to B). For this assignment, the first state in the list of state machines is the start state. Transitions specify a target state (the state to transition to if the input matches), and the input to transition on. Your action functions should take a parameter specifying the event that just happened, the runtime parameters, and the actor this state machine is attached to (so that your functions can make modifications to the elements themselves).

Below is an example of part of a state machine specification for a draggable actor. The state machine description is also provided for you in statemachinetest.js:

```
    var sampleDescription = { states: [{
        name: "start",
        transitions: [ {
            input: "mousedown", //Event type name
            actions: [{
              func: record_down_location
            }] //Actions  to be called
            predicate: function(event) { return true } //Predicate to be called (halts if it returr
            endState: "down" //End state name
        }]
    }, {
        name: "down",
        transitions: [ {
            input: "mouseup",
            actions: [{
              func: do_drop
            }]
            endState: "start"
        }, {
            input: "mousemove",
            actions: [{
              func: move_icon,
              params: {
                new_icon: 'test.png'
              }
            }]
            endState: "down"
        }]
    }
    ]};
```

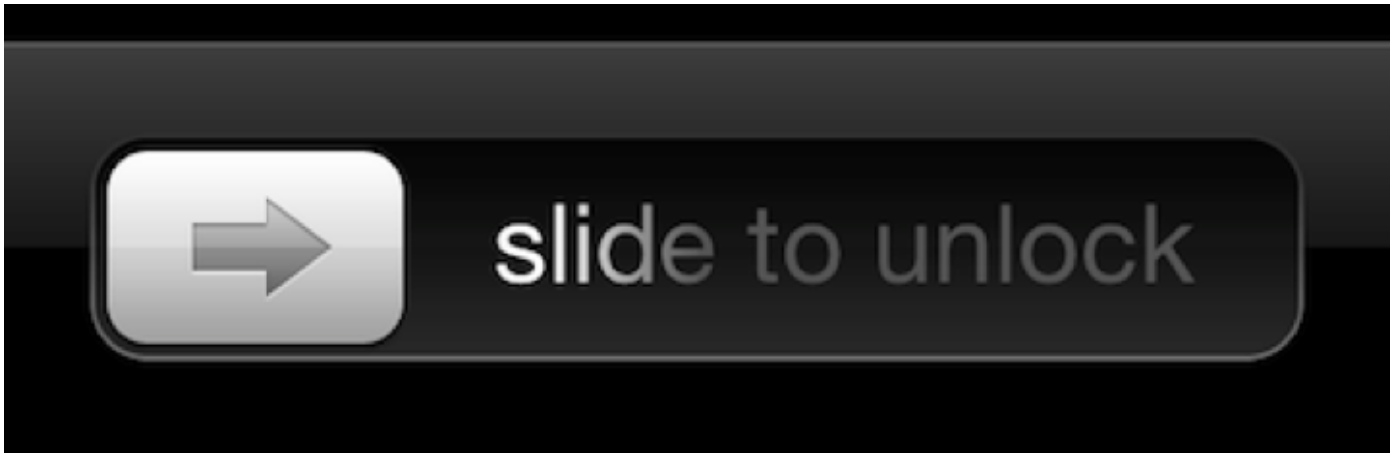
Specifically, the object breaks down as such:

- states: [Array] An array of "state" objects

- name: [String] The name of the state
- transitions: [Array] An array of transition objects from this state
  - input: [String] The input event that causes this transition
  - predicate: [Function] A predicate function (returns true / false) that indicates if the transition can proceed
  - actions: [Array] An array of actions objects that will happen during the transition
    - func: [Function] Transition function that will be called
    - params: [Object] A set of parameters that will be provided to the func at runtime
  - endState: [String] The end state of the
  - *other* - Additional properties that are event specific (like target for buttonpress)

Part 1

This assignment is to specify a finite state controller that describes the behavior of the "unlock slide" interactor on the iPhone (depicted below). You must use this cross-platform implementation as your source of information about how this works: http://demo.marcofolio.net/iphone_unlock/ (http://demo.marcofolio.net/iphone_unlock/) (note that it does not function exactly like an iPhone, but does ensure that everyone is making a state machine for the same interactor)



What To Do

Draw the state diagram for the full dynamic behavior of the "unlock slide" interactor on the iPhone.

Note that this interactor has a number of subtle features where parts of the visual appearance that change over time. These need to be captured in your finite state controller. You can assume that you get an event that marks the passage of time (at whatever interval is convenient). You can also assume higher level events (such as entering a region). You can also make use of guards on your transitions if you prefer. You can also assume the existence of an `animate_text()` method which animates the text in parallel to any other input and a `return_arrow()` which animates the motion of the button with the arrow on it back to its home position. We will revisit the implementation of animation in a later assignment after learning more about it.

Turning Your Assignment In

Please provide your answer in a PDF or Microsoft Word (.doc, .docx, or .rtf) and submit it on Blackboard. Be sure to provide both the full state machine and an explanation of what each action does.

Part 2

The second part will be the actual implimentation of the game engine, as well as an additional test for the game.

Game object

This class is responsible for managing drawing of the game board, the translation of low- level events to higher-level FSMEvents, and the delivery of those events. The methods it should implement are:

Constructor:

Params - canvas: Game canvas element to draw on and watch events for.

The constructor needs to attach to the canvas, and then listen to canvas events, correctly dispatching them as they come in.

Methods:

- `onDraw(Canvas canv)` - This method is responsible for drawing all Actors.
- `actorsUnder(x,y,width,height)` - Find and return the list of actors whose bounds overlap the given rectangular area. The actors (if any) in the list should be in reverse drawing order. That is, the actors drawn later should appear earlier in the

list.

- `pointDispatch(event)` - Dispatch the given event to one actor under the given x,y position. When multiple actors are under the position we offer it to them in reverse drawing order. As soon as a actor takes the event (returns true from its deliverEvent() method) we stop offering it to others so that only one actor gets the event.
- `areaDispatch(event)` - Dispatch the given event to one actor whose bounds overlap the given rectangle. When multiple actors are overlapped we offer it to them in reverse drawing order. As soon as a actor takes the event (returns true from its deliverEvent() method) we stop offering it to others so that only one actor gets the event.
- `dispatchDirect(event, actor)` - Dispatch the given event directly to the given Actor.
- `dispatchToAll(event)` - Dispatch the given event to all actors in reverse drawing order. This dispatch does not stop after the first actor accepts the event, but instead always continues through the list of all actors.
- `dispatchTryAll(event)` - Attempt to dispatch the given event to all actors in reverse drawing order stopping as soon as some actor takes the event (returns true from its deliverEvent() method).
- `dispatchDragFocus(event)` - Dispatch the given event to the current drag focus object (if any). If there is no current drag focus or the current drag focus object rejects the event (returns false from its deliverEvent() method), this method returns false. All events which contain an x,y position will have their x,y position adjusted by (- *grabPointX, -grabPointY) prior to being delivered (or more correctly a copy of the event will be adjusted and delivered). In this way the position indicated in the event will reflect where the top-left corner of the dragged actor should be placed, rather than where the cursor was (which will normally be inside the actor; specifically at a distance of (*grabPointX, _grabPointY) from the top-left of the object).

Actor object

This class is responsible for managing the actor's FSM, responding to events, and drawing the actor:

Constructor Parameters (in this order):

- params: A params object containing values for the actor:
  - height: height of the actor,
  - width: width of the actor,
  - x: left position of the actor,
  - y: top position of the actor,
  - img: image to display for the actor,

Methods:

- `setFSM(fms)` : Sets the current FSM for the actor. Resets the startstate of the actor to the first state of the FSM provided.
- `draw(context)` : This method is responsible for the actor's image. A actor may or may not have a image. If there is no image, the actor will not have anything visible on the game board (but that does not mean it is inactive). If a image is drawn, it should be drawn with the top left corner at the x,y coordinates of the Actor. However, the intrinsic dimensions of the image should be respected, and the Actor width and height should not cause the image to be clipped, nor transformed in any other way.
- `deliverEvent(event)` - This method should be called by your code to deliver an event to a particular Actor. The method returns true if the event is consumed and false if it is not. The Actor should only consume an event if it has a transition with

an event that matches that event.

- `makeFSMTransition(event, transition)` - This method should be called by your code whenever a Transition is being followed. This method should make sure that any Actions on the transition are carried out (which may also require redraws) and should maintain what state has been transitioned to.
- `matchEvent(event, transition)` - This method should match a particular event to a transition. Most events have fairly simple matching based on the called for event and the transition applied, however the message and buttonpress events also need to check that the message sent is the one called for, and the button pressed has the right ID.

Action object

This object contains a list of actions that should be supported by your actors. See the starter code for more information about each action's implimentation

State Machine Input Events

Your state machine will need to translate raw input events into strings like "mousedown", "mouseup" for interpretation by the actor's state machine. The canvas needs to capture these events, and then they need to be dispatched appropriately. These are the actions that need to be captured:

- mousedown: The mouse button was pressed down over this object.
- mouseup: The mouse button was released over the object.
- mousemove: The mouse has moved over the object.

In addition to these, the game engine has to create some artificial events depending on the state of the engine and certain transitions. These include:

- animstart: Fired whenever an animation begins. Created as part of `runAnimation`.
- animmove: Fired whenever an animation is in progress.
- animend: Fired whenever an animation completes.
- buttonpress: Created by external buttons.
- dragmove: Fired whenever a mousemove is detected on an actor that has the drag focus
- dragend: Fired whenever a mouseup is detected on an actor that has the drag focus

Relevant spec: here is a full list (https://developer.mozilla.org/en-US/docs/Web/Events) of events that the browser will send you. (btw, it says the parameter is case insensitive, but I found it to be case sensitive.) Your state machine has to respond to the three natural ones above, as well as the 7 artificial ones. You must impliment the listeners for the three natural events, as well as artificially create the two dragevents during `dispatchDragFocus`

Testing Your State Machine

You are given one sample game to test your game engine on. In addition to this, you need to build another game test using the same game engine. You don't need to expand the game engine in anyway to recieve full points on this test.

Bells and Whistles

Completing the above requirements and having well-documented code with no errors will get you 45 out of 50 points, which is an A. You must complete one of the below 'bells and whistles' to get full points. You can complete as many of these as you want, but you will get no more than 10 points total. Different bonuses are worth different numbers of points, based on difficulty. You may receive up to 5 points for bells and whistles, bringing you up to up to 5 extra credit points.

1. Extend the game engine (up to 5 points): There are many ways to add more functionality to the game engine. This could include new actions, recognizing additional events, or even adding more actor types. Be creative! This can be part of the additional game you create for testing.
2. Allow the same state machine to be attached to multiple actors, and show an example of when this might be useful. (3 points): You will only get points for this if you show an example of where attaching to multiple elements is actually useful. You should add an extra test for this.
3. Implement a probabilistic actor state machines, and show an interesting example of your probabilistic state machine being used (5 points): Each transition also has a probability attached to it. Instead of transitioning when an event happens, your state machine transitions with some probability. This can make your transitions behave somewhat randomly (and in interesting ways!). One useful example of this would be to generate semi-random animations (when you would transition randomly at every timer tick). Implement your probabilistic state machine and make a test that illustrates an application of this state machine (you should add an extra test for this).
4. Write a visualization for your FSMs (5 points): Make a visualization of all of your states & transitions, including highlighting the active state. You may want to consider using the drawing library you built in P2.

Show and Tell

The course instructor will be picking the most interesting state machine tests (i.e. the most interesting state machines you created) and showing them to the class. They may also be put on a public website. Please let me know if you do not want your solution shown in the README file in your assignment.

Turning Your Program In

This project has the same procedure as Projects 0, 1, and 2. Make sure to include the following files:

- actor.js: Completed Actor class
- game.js: Completed game class
- actions.js: Completed actions object
- statemachine-test2.html and .js: Your additional game engine test.
- README.txt: Describe your project as in previous projects.
- If you do any bells and whistles, also include the testing files for those bells and whistles. Make sure your README file includes any extra documentation about any extra things you did. Also, include any questions or difficulties you had. If something doesn't work, please also include this in the README file.

Grading

- Turn in is correct, code has no syntax errors: 5 pts
- Formatting, Comments and Coding style: 5 pts
- Correct implementation of Actor class: 10 pts

- Correct implementation of Game class: 15 pts
- Correct implementation of Actions object: 5 pts
- At least one additional test that more completly tests the game: 5 pts
- Bells and whistles: up to 5 pts