SSUI Web: Assignment 3

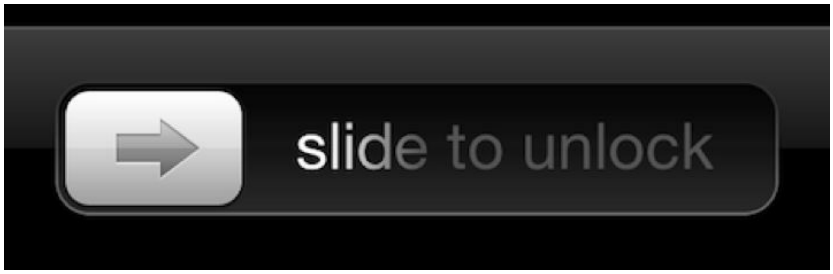
Finite State Machines

Part 1 and 2 Due: November 7th, 2016 @ 11:59pm

Part 1: Create a FSM Diagram

Overview, Learning Goals and Grading

This assignment aims to give you experience with the language of finite-state machine controllers. You will be creating a finite-state machine to describes the behavior of the "unlock slide" interactor on the original iPhone (depicted below).



For this assignment, you will study the behavior of this cross-platform implementation of the iPhone slider: http://demo.marcofolio.net/iphone_unlock/ (note that it does not function exactly like the original iPhone slider, but does ensure that everyone is making a state machine for the same interactor). Next, you will create a finite state machine diagram to represent the behavior of this slider. This assignment aims to help you understand:

- The correct notation for representing finite state machines **[25 pts]**
    - Not required to use guards, but notation must be correct if they are used
    - Correctly using start and final state (i.e., no links out of final state)
    - Actions only specified on transitions
    - Good use of variable names, with accompanying descriptions of what the variables represent
    - Correct use of higher and lower level events
- Complete in terms of the set of actions a user might take **[25 pts]**
    - Appropriate use of self transitions
    - Capturing all necessary events
    - Not capturing unnecessary events

What To Do

Create (or 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. In particular, the "slide to unlock" text has a highlighting animation, the button returns to the start position when not dragged all the way to the right. 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. In addition to your drawing, you should also submit a description explaining each action in the the state machine and any variables you used.

Turning Your Assignment In

Please provide your answer in a PDF and submit it on AutoLab by **November 7th, 2016 @ 11:59pm**. If your submission is hand drawn, then please make sure it is legible.

Part 2: Create a JS FSM Game Engine and Game

Overview, Learning Goals and Grading

In this part you will be implementing parts of a rudimentary game engine that uses a table-based finite state machine to describe the behavior of a canvas game. Similar to how you created a drawing library and then used it to create a doodle in the last assignment, in this assignment, you will be implementing the core functionality of a finite-state machine game engine and then showing how it can be used to create a simple "game." After you implement the game engine, it will accept a set of finite-state machines, which each describe the behavior of different Actors in your game. The engine will then use these state machines to power a canvas game. Thus, creating a game consists of building a set of finite state machine that describes the behavior of the game's actors and then passing it to your game engine to run the game.

This assignment is meant to give you experience with using finite-state machines to manage user interfaces. The rationale for using finite-state machines is that they simplify the task of building complex interactive elements. They provide a concise language for how interactive elements should respond to input over time. Also, this assignment is designed to give you experience with input handling and event delivery and dispatch.

This assignment aims to help you understand the following:

- How to implement event dispatch **[30 pts]**
    - How to implement events
    - How to implement multiple different dispatch policies (positional, focus, etc.)
    - Maybe how to pick dispatch policy
    - Reporting damage and redrawing
- How to implement a generic reusable finite-state machine engine **[30 pts]**
    - Transitions
    - Mapping events to state transitions and actions
    - Guards
- Separation of concerns between application architecture output and the application developer by creating your own game **[30 pts]**
    - How to define interactive behavior using finite state machines
    - How to use higher-level events to have allow actors to communicate.
    - Animation
- Style **[10 pts]**

Files Provided

Here is the starter code for A3 part 2. It contains the following files:

- game.js - JavaScript file that serves as the game engine, and is the root of the game.
- actor.js - JavaScript file that should hold an actor's information and its FSM
- actions.js - JavaScript file that has a list of predefined actions for use in the FSMs.
- statemachinetest.html - HTML file that goes with statemachine_test.js
- statemachine_test.js - Tester "game" for you to start from
- statemachinetest.css - CSS file that goes with statemachinetest.html
- statemachinetest.mov - a video of what the test game should look like.
- *.png - A set of icons to use as the actor images

Implement the Game Engine

Before creating your game, you must implement the core functionality of the game engine, defined by the Game, Actor, and Action classes. A game is defined by a Game object, which models a game using state machines on an HTML5 Canvas. A "game" is composed of several actors, which each have their own state machine. The game object also handles input and event dispatch (including focus and positional dispatch and dragging). In particular, when the game (i.e., the canvas) receives input, it needs to deliver the events to the appropriate actors.

The Actor objects will be built on a generic state machine implementation. Each actor will contain actor information (e.g., an image and x,y location) and will know how to draw itself

according to a simple absolute position layout. Additionally, each actor will have a state machine that describes its behavior. When an event is delivered to an actor, it will use its state machine to determine whether to take any actions and how to update its state. The actions specified by the state machine are events that trigger behavior in the canvas (e.g., animating an object).

Finite-State Machine Specification

For this project, an actor's state machine will be described as a JavaScript object (in JavaScript Object Notation, or JSON) indexed by state names and transitions, both of which must be unique. Stored at each state and transition is another JSON object that contains actions (a list of actions to execute on transition), endState (the state to transition to), and, optionally, predicate (which specifies a test that must be met for the transition to occur). Each action in the action list 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. State machine descriptions are also provided for you in statemachinetest.js:

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

```
                }],
                endState: "down"
            }
        }
    };
```

Specifically, the object breaks down as such:

- Finite-state machine [Object indexed by state names]
    - States [Object indexed by transitions / event.type] :
        - predicate: [Function] A predicate function (returns true / false) that indicates if the transition can proceed (*optional).*
        - 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 to transition to.

Game object

This class is responsible for managing drawing of the game board, the translation of low-level events to higher-level Action events, and the delivery of those events. Note, that drawing (and redrawing) is done globally from the game object. The game object has a method, `damageActor(actor)`, that can be used to trigger a redraw. The methods that you will need to 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 an 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(area, 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 an 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 finite-state machine, 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(startState, fms):` Sets the current FSM for the actor. Initializes the current state to the state specified by startState.
- `draw(context):` This method is responsible for drawing 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 an 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. Make sure to check that if there is a predicate on the transition it returns true.
- `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.

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 implementation

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 of events that the browser will send you. (by the way, 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 implement the listeners for the three natural events, as well as artificially create the two drag events during `dispatchDragFocus`

Testing the Game Engine

You are given one sample "game" (it's not really much of a game) to test your game engine on. This sample game is defined in statemachinetest.js and statemachinetest.html. There is also a short video of the behavior of this game in test.mp4.

Creating your own Game

After you've tested your game with the provided example, you need to build another game to test the game engine. To do this you might want to duplicate statemachinetest.js and statemachinetest.html and modify the Finite-State Machines specified in statemachinetest.js.

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

Create a zip archive containing the following files and upload it to AutoLab by **November 7th, 2016 @ 11:59pm**:

- 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. 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.
- Any other files necessary for running your test/game (e.g., images).