Homework Assignment 4: Fifteen Puzzle

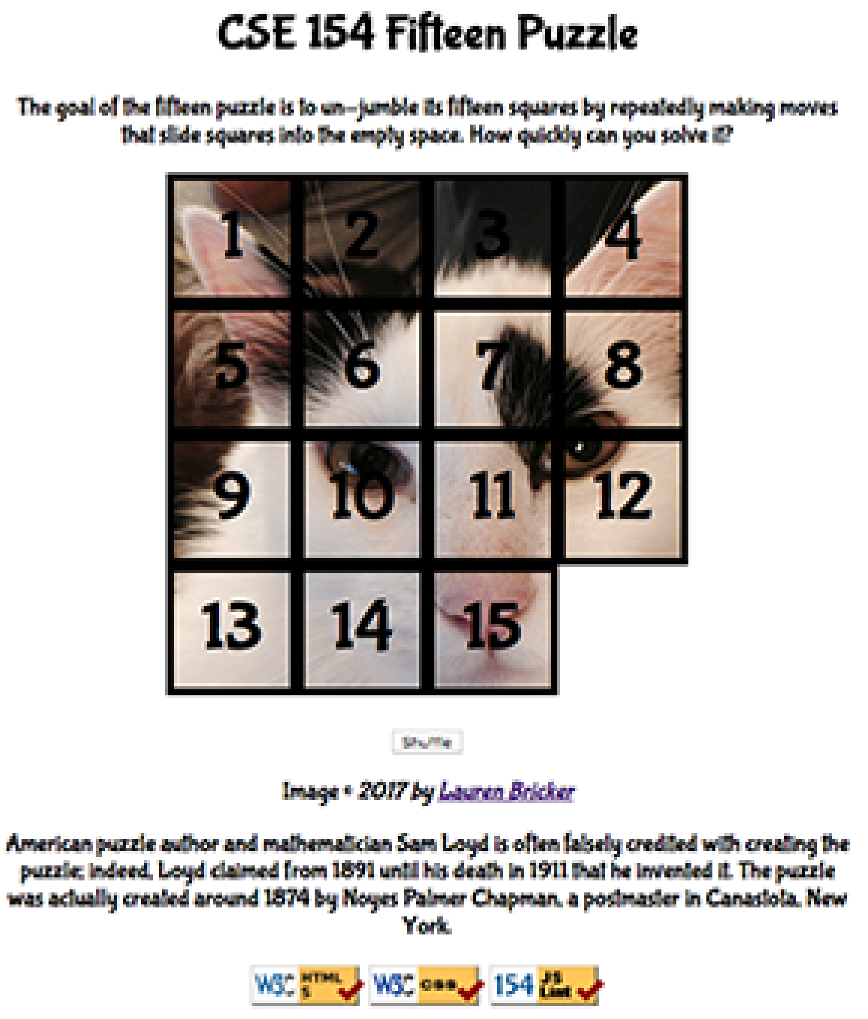
Due Date: Wednesday, April 25th

Overview

This assignment is about JavaScript's Document Object Model (DOM) and events. You'll write an interactive "Fifteen Puzzle" page, following the specifications provided below. The example output of this page is provided in the resources folder on the course calendar.

Background Information

The Fifteen Puzzle (also called the Sliding Puzzle) is a simple classic game consisting of a 4x4 grid of numbered squares with one square missing. The objective of the game is to arrange the tiles into numerical order by repeatedly sliding a square that neighbors the missing square into its empty space.



CSE 154 Fifteen Puzzle

The goal of the fifteen puzzle is to un-jumble its fifteen squares by repeatedly making moves that slide squares into the empty space. How quickly can you solve it?

Image © 2017 by Lauren Bricker

American puzzle author and mathematician Sam Loyd is often falsely credited with creating the puzzle; indeed, Loyd claimed from 1891 until his death in 1911 that he invented it. The puzzle was actually created around 1874 by Noyes Palmer Chapman, a postmaster in Canastota, New York.

Figure 1: Fifteen puzzle page: Expected output (rendered on Mac OSX)

Tips for solving a fifteen puzzle: First get the entire top/left sides into proper position. That is, put squares number 1, 2, 3, 4, 5, 9, and 13 into their proper places. Now never touch those squares again. Now what's left to be solved is a 3x3 board, which is much easier.

Learning Objectives

- Continue to practice all of the learning objectives from Homeworks 1, 2, and 3, including:
  - Carefully reading a specification.
  - Choosing appropriate CSS rules to style a page.
  - Reducing redundancy in your CSS while producing expected output.
  - Retrieving information from a user using events on the the Document Object Model (DOM) objects within your JS.
  - Modifying your web page using JS and DOM objects.
  - Producing quality readable and maintainable code including unobtrusive JavaScript that is modular and kept separate from the HTML/CSS.
- Responding to the user's mouse events on DOM objects
- Manipulating the DOM tree using JS functions.
- Manipulating DOM object styles through JS.

Your site will consist of four files. The first is `fifteen.html` which contains the interface that allows the user to play the Fifteen puzzle (the `HTML` will be provided for you). Your page will be styled by linking to a style sheet named `fifteen.css` and will be controlled by JavaScript code that you will write in `fifteen.js`.

You will also submit `background.jpg`, a background image of your own choosing, displayed underneath the tiles of the board. Choose any school-appropriate image you like, so long as each of the fifteen tiles can be distinguished on the board. You may use an cropped photograph you took yourself, a scanned in image you drew, or you may find an image online, provided it is **fair use** (labeled for reuse). As discussed later in this spec, make sure to save the image URL for the citation which will be added to the page dynamically. You must also not use the Mario image typically shown in previous versions of this assignment.

Files to Submit

In total, you will submit the following files:

1. `fifteen.css`: The stylesheet for `fifteen.html`.
2. `fifteen.js`: The JavaScript code for `fifteen.html`.
3. `background.jpg`: the background image, suitable for a puzzle of size 400x400 (px).

You will not submit any `.html` file, nor directly write any `HTML` code. You will need to download the `fifteen.html` file to your machine while writing your JavaScript code, but your code should work with the provided files unmodified as you will not be turning this `.html` file in. You will write JavaScript code that interacts with the page using the DOM. To modify the page's appearance, write appropriate DOM code to change styles of on-screen elements by setting classes, IDs, and/or style properties on them.

External Requirements

Your webpage *should match the overall appearance of the screenshots provided on the course website* and it *must match the appearance specified in this document*. We do not expect you to produce a pixel-perfect page that exactly matches the expected output image. However, your page should follow the specific guidelines specified in this document and match the look, layout, and behavior shown here as closely as possible.

Appearance Details

- All text on the page is displayed 'Bubblegum Sans' (imported from Google Fonts), falling back to the default cursive font on the user's machine. Any font size otherwise unspecified in this spec should be 14pt.
- Everything on the page is centered, including the top heading, paragraphs, the puzzle, the Shuffle button, and the W3C image links at bottom.
- The paragraphs on the page have a width of 80%.
- In the center of the page are fifteen tiles representing the puzzle. The overall puzzle occupies 400x400 pixels on the page, horizontally centered.
- Each puzzle tile occupies a total of 100x100 pixels, but 5px on all four sides are occupied by a black border. This leaves 90x90 pixels of area inside each tile.
- There is an image attribution listed under the Shuffle button. If this image is copyrighted by you, the author of this webpage, or someone who has granted you the right to use this image on your page, you must attribute this source with their copyright, like the following:

  Image ©*2017 by firstname lastname*

  If the image you use is fair use, you will still display a URL or other information about where you got the image, for example:

  Image from *http://www.thisisthesite.com/thisistheimage.jpg*

Behavior Details

- The `HTML` file given to you does not contain the fifteen div elements to represent the puzzle pieces, **and you are not supposed to modify the `HTML` file**; so you will have to create the puzzle pieces and add them to the page yourself using the JavaScript DOM. Before the `load` event fires on the window, the page should not contain any puzzle pieces.
- Once the window loads, the page should appear with the puzzle in its properly arranged order like in the screenshot on the first page of this spec, with 1 at top-left, 4 at top-right, 13 at bottom-left, the empty square at bottom-right, and so on.
- Each tile displays a number from 1 to 15, in a 40pt font that is centered both horizontally and vertically in the tile.
- Each tile displays part of the image `background.jpg`. Note that this file MUST be in the same folder as your page.
- The part of the image displayed by each tile is related to that tile's number. The "1" tile shows the top-left 100x100px portion of the image. The "2" tile shows the next 100x100px of the background that would be to the right of the part shown under the "1" tile, and so on.

- Your background image appears on the puzzle pieces when you set it as the background-image of each piece. By adjusting the `background-position` of each div, you can show a different part of the background on each piece. One confusing thing about `background-position` is that the x/y values shift the background behind the element, not the element itself. The offsets are the negation of what you may expect. For example, if you wanted a 100x100px div to show the top-right corner of a 400x400px image, set its background-position property to 300px 0px .

  The following is a complete listing of the exact background-position values each location on the board should have:

0px 0px	-100px 0px	-200px 0px	-300px 0px
0px -100px	-100px -100px	-200px -100px	-300px -100px
0px -200px	-100px -200px	-200px -200px	-300px -200px
0px -300px	-100px -300px	-200px -300px	

- Centered under the puzzle tiles is a Shuffle button that can be clicked to randomly rearrange the tiles of the puzzle. See the "Shuffle Algorithm" section of this spec for more details about implementing the shuffle behavior.

- The `HTML` file you are given also does not have the citation information displayed on the screen, so you will need to create and add these DOM elements to your page programatically within your `.js` file.

All other style elements on the page are subject to the preference of the web browser. While the screenshot in this document was rendered on Mac OSX, default cursive fonts vary quite a bit from system to system, so it is possible that a screenshot of your page will look somewhat different than the expected output.

Playing the Game

- When the mouse button is pressed on a puzzle square, if that square is next to the blank square, it is moved into the blank space. If the square does not neighbor the blank square, no action occurs. Similarly, if the mouse is pressed on the empty square or elsewhere on the page, no action occurs.

- When the mouse hovers over a square that can be moved (neighbors the blank spot), its border and text color should become red. The mouse cursor also should change into a "hand" cursor pointing at the square (do this by setting the `CSS` cursor property to `pointer`.) Once the cursor is no longer hovering on the square, its appearance should revert to its original state. When the mouse cursor hovers over a square that cannot be moved, it should use the system's standard arrow cursor (set the cursor property to default) and it should not have the red text or borders or other effects (you may find it useful to use the `:hover CSS` pseudo-class to help deal with hovering.)

- The game is not required to take any particular action when the puzzle has been won. You can decide if you'd like to pop up an alert box congratulating the user or add any other optional behavior to handle this event.

Shuffle Algorithm

Centered under the puzzle tiles is a Shuffle button that can be clicked to randomly rearrange the tiles of the puzzle. When the Shuffle button is clicked, the puzzle tiles are rearranged into a random ordering so that the user has a challenging puzzle to solve.

The tiles must be rearranged into a solvable state. Some puzzle states are not solvable; for example, the puzzle cannot be solved if you swap only its 14 and 15 tiles. Therefore your algorithm for shuffling cannot simply move each tile to a completely random location. It must generate only solvable puzzle states if you want full credit.

We suggest that you generate a random valid solvable puzzle state by repeatedly choosing a random neighbor of the missing tile and sliding it onto the missing tile's space. Roughly 1000 such random movements should produce a well-shuffled board. Here is a rough pseudo-code of the algorithm we suggest for shuffling.

```
for (~1000 times):
    neighbors = []
    for each neighbor n that is directly up, down, left, right from empty square:
        if n exists and is movable:
            neighbors.push(n)
    randomly choose an element i from neighbors
    move neighbors[i] to the location of the empty square
```

Notice that on each pass of our algorithm, it is guaranteed that one square will move. Some students write an algorithm that randomly chooses any one of the 15 squares and tries to move it; but this is a poor way to shuffle because many of the 15 squares are not neighbors of the empty square. Therefore the loop must repeat many more times in order to shuffle the elements effectively, making it slow and causing the page to lag. This is not acceptable.

A few requirements about the shuffle algorithm:

- You are not required to follow exactly the algorithm above, but if you don't, your algorithm must exhibit the desired properties described in this section to receive full credit.
- **Your algorithm should be efficient**; if it takes more than 1 second to run or performs a large number of tests and calls unnecessarily, you may lose points.
- For full credit, your shuffle should have a good random distribution and thoroughly rearrange the tiles and the blank position.
- Your shuffle algorithm will need to incorporate randomness. You can generate a random integer from 0 to K, which is helpful to randomly choose between K choices, by writing, `parseInt(Math.random() * K)`.

Some development hints about the shuffle algorithm:

- We suggest first implementing code to perform a single random move; that is, when Shuffle is clicked, randomly pick one square near the empty square and move it. Get it to do this once then work on doing it many times in a loop.
- In your shuffle algorithm, or elsewhere in your program, you may want access to the DOM object for a square at a particular row/column or x/y position. We suggest you write a function that accepts a row/column as parameters and returns the DOM object for the corresponding square. It may be helpful to you to give an id to each square, such as "square_2_3" for the square in row 2, column 3, so that you can more easily access the squares later in your JavaScript code. (If any square moves, you will need to update its id value to match its new location.) Use these ids appropriately. Don't break apart the string "square_2_3" to extract the 2 or 3. Instead, use the ids to access the corresponding DOM object: e.g., for the square at row 2, column 3, make an id string of "square_2_3".

Development Strategy

Students generally find this to be a tricky assignment. Here is a suggested ordering for tackling the steps:

1. Make the fifteen puzzle pieces appear in the correct positions without any background behind them.
2. Make the correct parts of the background show through behind each tile.
3. Write the code that moves a tile when it is clicked from its current location to the empty square's location. Don't worry initially about whether the clicked tile is "movable" (whether it is next to the empty square).
4. Write code to determine whether a square can move or not (whether it neighbors the empty square).
5. Implement a highlight when the mouse hovers over movable tiles. Track where the empty square is at all times.

Hints

- Use absolute positioning to set the x/y locations of each puzzle piece. The overall puzzle area must use a relative position in order for the x/y offsets of each piece to be relative to the puzzle area's location.
- You can convert a string to a number using `parseInt`. This also works for strings that start with a number and end with non-numeric content. For example, `parseInt("123four")` returns 123.
- Many students have bugs related to not setting their DOM style properties using proper units and formatting. The string you assign in your `JS` must exactly match what it would be in the `CSS` file. For example, if you want to set the size of an element, a value like 42 or "42" will fail, but "42px" or "42em" will succeed. When setting a background image, a value like "`foo.jpg`" will fail, but "`url(foo.jpg)`" will succeed. When setting a background position, "42 35" will fail but "42px 35px" will succeed. And so on.
- We suggest that you do not explicitly make a div to represent the empty square of the puzzle. Keep track of where it is, such as by row/column or by x/y position, but don't create an actual DOM element for it. We also suggest that you not store your puzzle squares in a 2-D array. This might seem like a good structure because of the 4x4 appearance of the grid, but will likely make it more difficult to keep it up to date as the squares move. Furthermore, it is easy to write 2-D array code that ends up being poor style because of how difficult it is to maintain the proper state of the arrays.
- Many students have redundant code because they don't create helper functions. You should write functions for common operations, such as moving a particular square, or for determining whether a given square currently can be moved. The `this` keyword (see lecture slides) can be helpful for reducing redundancy.

Internal Requirements

- For full credit, your page must use valid HTML5, CSS3, and JS and successfully pass the W3C HTML5 and W3C CSS3 validators and our JSLint (https://courses.cs.washington.edu/courses/cse154/18sp/jslint/jslint.html) with no errors.
- Your `CSS` and `JS` should also maintain good code quality by following the guide posted on the class web site. We also expect you to implement relevant feedback from previous assignments.
- Your`.js` file must be in the module pattern and run in strict mode by putting `"use strict";` within your file.
- Your JavaScript code and `CSS` files should have adequate commenting. The top of both files should have a descriptive header describing the assignment. You are to use the JSDoc commenting style for each function and module-global variable in your `JS` file (see the code quality guide on our website for more details on how to write JSDoc). Additionally, complex sections of code should be documented. You may use inline commenting within `JS` functions.

- Format your code similarly to the examples from class. Properly use whitespace and indentation. Use good variable and method names. Lines of code should be fewer than 100 characters long.

- Variables should be localized as much as possible. Minimize the use of module-global variables. Do not ever store DOM element objects, such as those returned by the `document.getElementById` or or `document.querySelectorAll` functions, as global variables. For reference, our own solution has only four module-global variables: the empty square's row and column (initially both are set to 3), the number of rows/cols in the puzzle (4), and the pixel width/height of each tile (100). The last two are essentially constants.

- If a particular literal value is used frequently, declare it as a module-global "constant" `IN_UPPER_CASE` and use the constant in your code.

- You should make an extra effort to minimize redundant JavaScript code. Capture common operations as functions to keep code size and complexity from growing. You can reduce your code size by using the `this` keyword in your event handlers.

- Separate content (`HTML`), presentation (`CSS`), and behavior (`JS`). Your `JS` code should use styles and classes from the `CSS` rather than manually setting each style property in the `JS`. For example, rather than setting the `.style` of a DOM object, instead, give it a `className` and put the styles for that class in your `CSS` file. **Note:** Style properties related to x/y positions of tiles and their backgrounds are impractical to put in the `CSS` file, so those can be in your `JS` code.

- Use unobtrusive JavaScript so that no `JS` code, `onclick` handlers, etc. are embedded into the `HTML` code.

- You should not use any external JavaScript frameworks or libraries such as jQuery to solve this assignment.

Academic Integrity

As with any CS homework assignment, you may not place your solution to a publicly-accessible web site, neither during nor after the school quarter is over. Doing so is considered a violation of our course academic integrity policy. As a reminder: The University of Washington has an entire page on Academic Misconduct on their Community Standards and Student Conduct Page. Please acquaint yourself with the University of Washington's resources on academic honesty, and in particular how academic misconduct will be reported (which has been changed for 2017).