

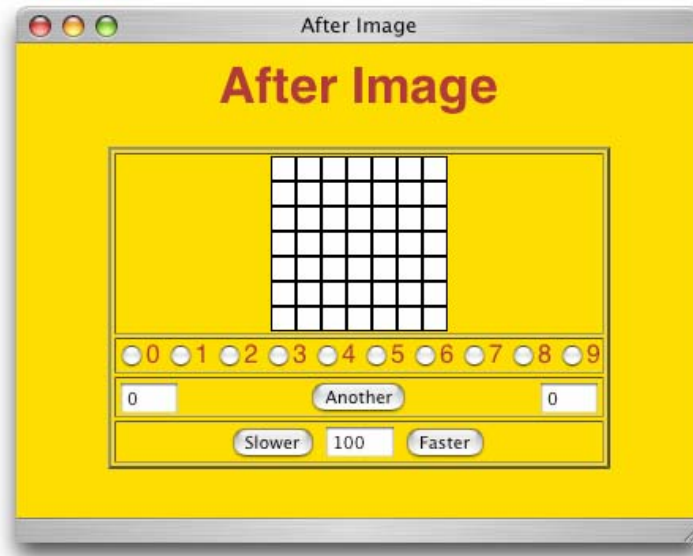
## Project 2: After Image

FIT100

Winter 2007

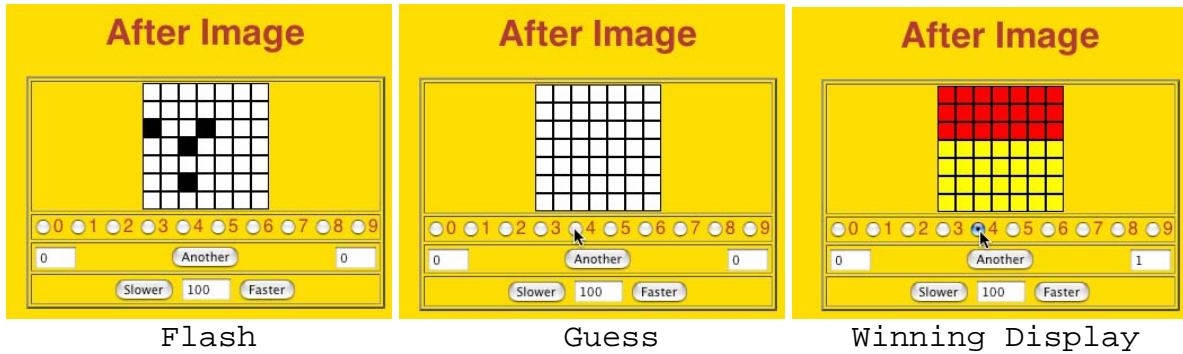
Have you ever stared at an image and noticed that when it disappeared, a “shadow” of the image was still briefly visible. This is called an after image, and we experiment with its effects in this project.

**Goal:** We will create a Web page to test whether we can remember the after image long enough to help ourselves perform a simple task. Specifically, we will create the following application:



This Web page works as follows. Three seconds after it starts, it flashes a random arrangement of black boxes in the 7x7 grid. The arrangement is displayed for 100 milliseconds, and then disappears. The user clicks on the radio button corresponding to the number of boxes, and gets scored. If the guess was right, the screen turns red and changes to yellow before going back to white. The score is counted in the right hand window. If the guess was wrong, the screen turns brown for two seconds. The score is counted in the left hand window. Then the user can try again by clicking on “Another”.

The buttons at the bottom control the speed of the flash. Clicking on Slower increases the time by 5 milliseconds; clicking on Faster decreases the time by five milliseconds.



These three sample images show the stages in a successful guess. The first shows the Flash of random black squares in the grid. In the second the squares have disappeared, and the user is making a Guess of “4”. Since this is the correct guess, the user is shown the correct display; the grid turns red, and then line-by-line from the bottom changes to yellow. If the Guess had been wrong, the screen would have changed to brown for two seconds. The user is then ready to try Another round.

In all parts of Project 2 you must comment the HTML and JavaScript that you write or change. The comments (`//` followed by an explanation) need not be long, but they must say what you’ve done in a way that is coherent to a human. *Uncommented HTML and JavaScript will not be graded.*

**Preparation.** The project requires the completion of Lab 6 and Lab 7, and reading of Chapters 18, 20 and 21. (Chapter 22 is optional, but may be useful since it solves a similar problem.) Further information can be found at [W3Schools documentation](#). It is possible to *start* the assignment using only the information from Lab 6.

**Advice.** As with all Web pages, it is best to work with both a browser and the editor open simultaneously, so that each change can be tested. Also, it is advisable that you read and understand the whole project before beginning to work on it. Once you begin, working through the assignment in the order given will make it easier to test. Finally, remember that there will be a two part turn-in, explained below.

**Overview.** Referring to the first picture, the project has the following physical components:

- the table,
- the (white) grid,
- the radio buttons,
- the Another button,
- the (left) Wrong and (right) Right textboxes,
- duration textbox, and
- the Slower and Faster buttons

Find them all.

In addition to the just-listed features of the page, there are the following actions:

- creating the grid
- coloring the grid
- displaying the random collection of black squares
- displaying the correct animation
- deciding if the guess is correct
- raising or lowering the time
- causing the first flash
- causing the Another flashes

Think about what causes each to happen and what happens.

Finally, there are five support files needed: `WhiteBox.gif`, `BlackBox.gif`, `RedBox.gif`, `YellowBox.gif`, `BrownBox.gif`. These are found in **Files and Stuff** on the class Web page.

A good strategy when solving problems that look difficult—and the strategy we will follow here—is to divide the problem into smaller pieces that are easier to solve. These are the components we will use:

- Create the Web page with all of the buttons and textboxes, but without the grid.
- Create the grid of squares.
- Figure out how to create random black squares.
- Figure out how to make them disappear.
- Figure out how to record the radio button that was clicked.
- Figure out how to decide if the answer is right or wrong.
- Create the correct animation
- Figure out how to change the time.

Where did these steps come from? The first two set up the physical features of the Web page. Then, the steps follow the flow of activity as someone goes through a round. This “divide and conquer” strategy is a standard approach and *you can use it whenever you have a complicated IT task to perform.*

Notice that although we developed the list by following through the process of the Web page, we don’t necessarily have to solve them in that order. In fact, we will solve them a little out of order.

#### **Notice**

This project has two goals: developing programming skills and problem solving skills. For problem solving purposes parts of this assignment tell you only generally what to do. If a task is only generally specified, do not panic. Think about it. It will be possible to figure out what to do *and that is part of the assignment.*

## Task Description, Part A

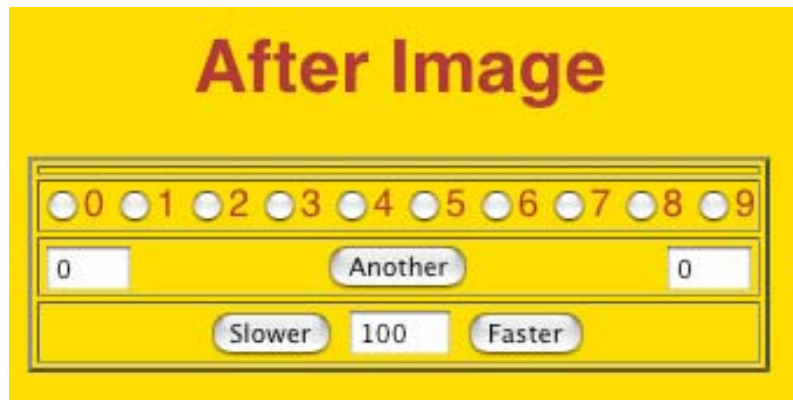
The task for part A has *only* one goal: to set up the physical components listed above.

1. *Create the Initial Page.* Set up a Web page that has an attractively colored background and contrasting text. (I have used a gold background with brown text; you must make different choices.) Select a specific font face. Include the heading. The initial page should have a centered table with one column and four rows. Check the page.

2. *Radio Buttons.* Using the Lab 6 information on forms and the data from the textbook on radio buttons, set up the ten radio buttons in the second row of the table. Be sure that the spacing is as shown in the figure, though your browser may display it differently. Note that the `onClick` event handler for these buttons cannot be programmed yet, so specify it as `onClick= ''`, the last symbols being two single quotes, NOT one double quote. Notice that radio buttons must have a common name, and you should choose a name that makes sense to you.

3. *Another Button.* Using forms, create the elements of the third row, two textboxes and a button. Notice the alignment of these three elements and using the `align` attribute in the `<input ...>` tags, make the spacing as shown. As before, the `onClick` event handler cannot be programmed yet. Choose meaningful names for the textboxes.

4. *Row Four.* Continuing, set up the elements of the fourth row, making sure that the spacing is correct. Choose a meaningful name for the textbox. Be sure to initialize the value of the textbox to be 100. At this point, your page should look as follows:



5. *Got Boxes?* Get the five colored boxes used in this project and store them in the directory with the file `project2A.html`.

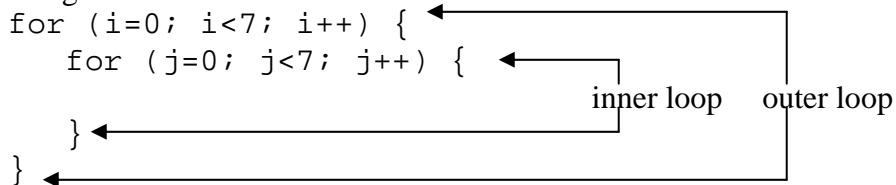
6. *Create the Grid.* The grid is a 7x7 array of copies of `WhiteBox.gif`. These white squares are placed into the first row of the table, centered. They *could* be placed by the tag

```
<img src=WhiteBox.gif>
```

but we would require 49 (=7x7) such lines which is too much work. Instead we will use two loops to do the work for us.

- a) Begin by inserting the `<script>` tags as the table data for the first row of the table. Then, declare two index variables, `i` and `j`, inside the tags.
- b) Next, write two World Famous Iteration loops, nested. (See page 617.) The `i` loop will be the outer loop, iterating 7 times, and the `j` loop will be the inner loop, also iterating 7 times:

```
for (i=0; i<7; i++) {
  for (j=0; j<7; j++) {
  }
}
```



The `i`-loop refers to the seven rows, 0 through 6, in order going down, and the `j`-loop refers to the boxes of each row, 0 through 6, in order from left to right.

- c) In the body of the inner loop, use `document.write()` to place a copy of the `<img src=...>` tag shown above. Remember that the argument to `document.write()`, i.e. the stuff inside the parentheses, which will be the `<img src=...>` tag, must be in quotes.
- d) After the end of the inner loop, i.e. after the closing brace of the inner loop (`}`), use `document.write()` to place a `<br>` tag. It'll have to be in quotes, too, of course. This tag ends a row of seven boxes and moves to the next row.

Check the textbook, p. 578-580, if you do not remember how `document.write()` works.

Your page should now look like the first figure in this document with the grid colored white.

Insert the timestamp code at the end of your HTML file:

```
<script language = "JavaScript">
var modified;
document.write("Last Modified: ");
modified = document.lastModified;
document.write(modified);
</script>
```

Store your file, named `project2A.html` in the `fit100` directory of your public Web space.

**Extra Credit to submit with Part A.** Find an explanation of the human after image effect on the Web, and write a short paragraph in your own words explaining the after image effect. Include the paragraph at the end of your After Image page and cite sources.

End of Part A.

## Task Description, Part B

The goal of task B is to finish the After Image application. This involves making the page “work,” mostly by creating event handlers for the inputs. Make a copy of the Part A solution, and rename it `project2B.html`. Follow these steps.

In all parts of Project 2 you must comment the HTML and JavaScript that you write or change. The comments do not have to be long, but they must say what you’ve done in a way that is coherent to a human. *Uncommented HTML and JavaScript will not be graded.*

7. *Prefetch*. It’s important to prefetch the images used for animations, so that they display quickly and do not have to be fetched over the Internet. This is a three step process, as outlined in the textbook.

- a) Declare five variables, `white`, `brown`, `red`, `yellow` and `black` right after the previously placed declarations inside the first `<script>` tag.
- b) Initialize each variable to have as its value a new `Image`, as in  

```
white = new Image;
```
- c) Assign each color name its correct gif image, as in  

```
white.src = "WhiteBox.gif";
```

All five colors should now be prefetched.

8. *Color the grid*. In the application the grid must be colored several different colors. We will write a function with one parameter, `color`, that performs this operation. The function will be called `tint()`, giving it the form

```
function tint (color) {  
  
}
```

It can be declared at the end of the `<script>` tags in the first row of the table.

The `tint()` function works very much like the grid coloring operation in Step 6. It requires another pair of nested loops (the `i` and `j` index variables can be used again), but this time we will not be placing an `<img src=...>` tag. Rather, we will change the previously placed images that are stored in the `document.images` array of the Web page structure, as explained on pages 626-627 of the textbook.

- a) Place the nested loops in the body of the `tint()` function.
- b) To change the gif stored in the `document.images` array at, say, position *index*, we simply change its `src` field to the name of the new gif. For example, we write

```
document.images[index].src = "BrownBox.gif"
```

if we wanted to change it to the brown box. But we don’t always want to have a brown box. Instead, we want whatever color is passed to the `tint()` procedure,

e.g. `tint(brown)`, so we assign the source field of the `color` parameter that is passed in, as in

```
document.images[index].src = color.src
```

This is the body of the inner loop except we haven't programmed the *index* calculation yet.

- c) To compute the value of *index*, recall that the `document.images` array is one long list of the images of the Web page, listed in order that they were drawn originally. So, the 49 box gifs of the grid are numbered 0 through 48, starting at the top row leftmost box, and going to the last row rightmost box. So, if we know the row (*i*) and the column (*j*) numbers of the gif, which are the indexes of the nested loops from part (a), we can compute the position by multiplying the row number times the length of the row (7), and adding the column number, i.e.  $index = 7 * \text{row number} + \text{column number}$ . Fill in the proper index calculation for the `document.images` reference placed in part (b).
- d) Check to see if your `tint()` works, by placing a call to it after (not within) the loops written in Step 6. Try the call `tint(red)`, which should recolor the array. When testing is complete, remove this testing call.

9. *Generate A Random Pattern.* There are three ways in which the pattern is random. The total *number* of squares in the pattern is a random number between 0 and 9. Then, for each of those squares, the *row* and *column* of the square are chosen at random. We will use the `randNum()` function described in the textbook, pages 588-591:

```
function randNum ( range ) {  
    return Math.floor ( range * Math.random() );  
}
```

Include this code following the `tint()` function.

The process of creating the random pattern will be packaged into a function which we will call `flash()`. Write the skeleton for `flash()` and place it after `randNum()`.

The logic of `flash()` is simple:

- a) generate a random number from 0 through 9 for the number of black boxes to draw, `randNum(10)`, and assign it to a variable with a name of your choosing; you have to declare the variable, of course.
- b) set up a WFI loop which iterates once for each box we will draw, that is, the iteration limit is the number created in step (a).
- c) declare two new variables with names of your choosing, one for a row value and the other for a column value, and assign each of them a random value from 0 to 6.
- d) as with Step 8c above, compute the *index* position in the `document.images` array and assign its `src` field the value `black.src`.

- e) check your work by placing a call to the `flash()` function just before the `</script>` tag. Each time you refresh the page, you should get a different array of black squares.

10. *Make the first flash.* Our plan is that the `flash()` function should go off three seconds after the page loads. To do this we set a timer, as explained on pp. 622-623. Therefore, instead of calling `flash()` immediately, as in Step 9e above, we set a timer to call it after 3000 milliseconds, i.e. after 3 seconds. We replace the Step 9e call with

```
setTimeout("flash()", 3000);
```

This sets a 3 second timer, and when it “goes off,” `flash()` is called.

11. *Make the After Image.* The black boxes should disappear after 100 milliseconds (The user should control this, but we’ll fix this in the next step.) How do we make the black boxes disappear in 100ms? Set a timer for 100 milliseconds, and when it wakes up, call `tint(white)`. When should that be done? Immediately after the black boxes are drawn, that is, setting the timer should be the last step in `flash()`. Add this timer call as the last operation in `flash()`. Check that it works.

12. *Event handlers for Slower/Faster.* In order to control how much time passes before the random black boxes disappear, declare a variable (I will call it `dura` for duration, but you need some other name), and initialize it to 100. Replace the 100 milliseconds in the timer call of Step 11 with that variable.

The event handlers for the two buttons will change `dura`. They are similar in operation: Slower will increase the value of `dura` by 5, and Faster will decrease the value by 5, and both should update the value in the textbox. I will give the textbox the name `tb`, but you used a different name when you defined the box above. Follow these steps to create the event handlers.

- a) Using the ideas from Lab 6, write an `onClick` event handler for Slower inside the two single quotes, which performs two operations:
  - i) Adds 5 to the duration variable, as in `dura = dura + 5`, and
  - ii) Assigns the duration variable as the value of the `tb` window; see Lab 6. The affect is to lengthen the amount of time the black squares are displayed.
- b) Create the event handler for Faster in the same way, except that the duration variable gets smaller by 5.

Try out the Web page and verify that the Slower/Faster buttons work properly.

13. *Event handler for Another.* The `onClick` event handler for the Another button is now easy to write. What should it do? It must call one function. Write that event handler.

14. *Event handlers for Radio Buttons.* The `onClick` event handlers for each of the radio buttons are similar to Another. They also call a function (it will be written next) that



checks to see if the answer is right. Though we haven't written it, we will use it in the event handler, passing the amount that the user guessed. So, for example, the event handler for the 0 radio button would be

```
onClick='check(0)'
```

Each of the radio buttons needs a similar event handler that passes its amount to `check()`.

15. *Testing if the Guess is Right.* Writing the `check()` function is simple. We give its structure

```
function check (guess) {  
  
}
```

and fill in the operations. The logic requires that we test the `guess` parameter to see if it matches the value randomly selected in Step 9a. This will require an `if`-statement, of course. If `guess` and the random number are equal, that is, our `==` test is true, we

- add one to the value in the right hand textbox (see Lab 6),
- tint the grid red, (we will color it yellow below), and
- set a timer to recolor the window white in 2 seconds=2000 milliseconds.

If the guess is wrong, i.e. our `==` test is false and we're the `else`-statement, we

- add one to the value of the left hand textbox (see Lab 6),
- tint the grid brown,
- set a timer to recolor the window white in 2 seconds.

Construct the function `check()` so that it performs this computation.

16. *Extra Credit: Animate the Correct Display.* To begin, change the timer that follows the red tinting so that instead of setting time, it calls `add1row(6)`.

The function `add1row()` has one parameter, the row to change to yellow. It does so, and then sets a timer to call itself in 200 milliseconds with one smaller row, unless the row is 0, i.e. this is the last row. In that case it sets the timer to tint the grid white. The function is

```
function add1row (row) {  
  
}
```

and the operation is as follows.

- a) The function begins by using a WFI to change the seven boxes of row `row` to yellow, in the same way as in Step 8b. The index `j`, used in other places can be used here as the index variable. Notice that only indexes for the seven boxes of a single row are changed.
- b) Next, check to see if `row` is equal to zero. If it is, set a timer for 200ms, and when it goes off, tint the grid white as usual.
- c) If `row` is not equal to zero, call `add1row()` again with `row-1` as its argument in the parentheses. Caution: the computation `(row-1)` cannot be inside any quotation marks, so it will be necessary to compose the first argument of the `setTimeout()` call using concatenation.

d) Check to see that the function works properly.

Be sure the Last Modified text is included, and place your project2B.html file in your fit100 directory in your public Web space, together with the five image files.

**Grading.** The two turn-ins are graded on the same criteria. Points will be assigned in the grading as follows:

- Are components of the page in place?
- Are requirements such as “choose a color” fulfilled?
- Does the page work as required?
- Is the HTML and the JavaScript well formed?
- Is the HTML and the JavaScript documented? (If your TA can’t understand it, it might not get graded.)