Assignment 7

Contents

The goal of this assignment is to make a working web version of the game 2048. Some of the game mechanics for 2048 are described below, but **you should play the game a few times before starting the assignment in order to gain an understanding of how the moves work**.

This is the first assignment in COMP 426 which will have an autograded portion as well as a portion that is graded by hand. At a high level, we will use the MVC design pattern to organize the code; the Model will be autograded, while the View and Controller will be hand-graded.

1. The autograded part is the game engine, representing the logic to store the state of the game and transition between states as the player makes moves. The Model code will be encapsulated in a `Game` class with methods such as `move()`, `getGameState()` and others that can be used to interact with the state of the game. To help you test your Model code, the `a07` directory comes with a `run_in_console.js` script which allows you to play a game of 2048 using your `Game` class directly in the terminal console.

2. The hand-graded part of a07 is to design a web page that allows you to play 2048. When the web page loads, it should display a 2048 board and allow the user to make moves by using the arrow keys on the keyboard. When the player makes moves, the web page should handle the keypress event, use the logic in the `Game` class to update the board state, and update the html accordingly to show the new state of the game. The user interface should also display (1) the player's current score, (2) a button that allows the user to reset the game, (3) an indication when the game is over, and (4) and an indication when the player wins.

1. The Model

The Model code should be encapsulated inside a `Game` class and exported from `a07/engine/game.js`. The autograder requires your `Game` class to expose a few methods described below. Beyond these specifications, You may implement the 2048 game engine in any way you see fit. Feel free to define your own internal game state and/or methods to help you get the game logic right.

1.1 The GameState Object

The autograder will use a special object, called the `gameState` object, to load game states into your `Game` class. The `gameState` object completely specifies the current state of a game at any instant in time, including (1) the position of the tiles on the board, (2) the current score, (3) whether the game has been won, and (4) whether the game is over. The `gameState` object has the following structure:

```
gameState = {
  board: number[],
  score: number,
  won: boolean,
  over: boolean
}
```

Field	Type	Value
board	number[]	A one-dimensional array of (size squared) numbers that represents the value of each tile on the board. Represented as a flat array in **row major** order. 0 should be used where no tile is present.
score	number	The score of the game at the current instant in time. Initially, the score should be set to zero. Every time the player makes a move that combines two tiles, the combined value should be added to the score (*e.g. if two 128 tiles are merged to make a 256 tile, then you add 256 to the score*).
won	boolean	True if a user has combined two 1024 tiles to make a 2048 tile
over	boolean	True if the board is in a state such that no more moves can be made

Here is an example of a game and its corresponding `gameState`:

> Note that the board inside of `gameState` is a one-dimensional array. You may implement your board with any memory structure you like. The only requirement is that when you output a `gameState`, the board is converted into a one-dimensional, or flattened, array. The one-dimensional array representation in the `gameState.board` must be in **row-major** order.

```
// ascii version of the game board
 [ ] [ ] [2] [4]
 [ ] [2] [ ] [4]
 [ ] [ ] [4] [8]
 [2] [4] [8] [16]

// the internal gameState object
{
  board: [
    0, 0, 2,  4, 0, 2,
    0, 4, 0,  0, 4, 8,
    2, 4, 8, 16
  ],
  score: 80,
  won: false,
  over: false
}
```

1.2 Game Class

The Model code should be encapsulated in a class called `Game` and exported from `a07/engine/game.js`. It is important for the autograder that you are exporting the class.

Your `Game` class must meet the specification below in order to receive a score of 100 by the autograder.

Constructor

1. The `Game` class must **have a constructor** that takes a single integer argument representing the width/height of the game.
2. When a new `Game` object is created (e.g. `const game = new Game(4)`), a new game of size 4x4 should be set up, and the board should be initialized by randomly adding two tiles to the game.
3. Your game class must support arbitrary game board sizes (i.e. 2x2, 3x3, 4x4, 5x5, etc).

Adding tiles

Any time a new tile is added to the game, it should have a **90%** chance of being a **2** and a **10%** chance of being a **4**. It should be placed into a uniformly random free space on the board.

New tiles should be added to the game only in the following situations:

1. When the game is first initialized or reset, two tiles should be added to the otherwise empty board.
2. After a legal move occurs, one tile should be added to the board.

Moving

- A legal move is one that causes pieces to slide or collapse on the board.
- If a player tries to make a move that does not change the state of the board, no move should occur and no new tiles should be added to the board.
- Once a move causes a 2048 tile to be created, the state should reflect that the game is "won".
- Once no legal moves are available, the state should reflect that the game is "over".

Events

- `Game` objects must be observable, supporting three different events: `move`, `win`, and `lose`.
- The `move` event occurs every time a valid move is made, `lose` occurs when the game transitions into a state that is "over", and `win` occurs when the game transitions into a state that is "won".
- The `Game` object must expose three methods named after the events: `onMove(callback)`, `onLose(callback)`, and `onWin(callback)`. These methods allow external observers to register event handler functions that listen for the event.

The following example shows how an event handler function might be registered with a `Game` object. The code creates a new `Game` object and registers an event handler function with the game. Every time a move occurs, the game object is responsible for calling the registered event handler function.

```
const game = new Game(4); // create a 4x4 game
game.onLose(state => {
        console.log('A move was made, and now the game is over!');
        console.log('Below is the final game state');
        console.log(state);
});
```

Your `Game` object is responsible for internally storing any event handler functions that are passed to it via `.onMove()`, `.onLose()`, and `.onWin()`. Then, any time an event occurs, your `Game` object is responsible for calling the stored event handler functions one at a time and passing in the current game state as a parameter.

Methods

The `Game` class must expose the following methods.

1. `setupNewGame()`: Resets the game back to a random starting position.

2. `loadGame(gameState)`: Given a `gameState` object, it loads that board, score, etc...

3. `move(direction)` : Given "*up*", "*down*", "*left*", or "*right*" as string input, it makes the appropriate shifts and adds a random tile.

   Notice how, when you play the game, tiles are only able to combine once per move. So, for example, if you had `[2][2][2][2]` and did a right shift, you end up with `[ ][ ][4][4]`, **NOT** `[ ][ ][ ][8]`. It would take another right shift to then combine the 4s into an 8.

4. `toString()`: Returns a string representation of the game as text/ascii. *See the `gameState` section above for an example*. This will not be graded, but it useful for your testing purposes when you run the game in the console. The `run_in_console.js` script uses the `toString()` method to print the state of the game to the console after every move.

5. `onMove(callback)`: Takes a callback function as input and registers that function as a listener to the `move` event. Every time a move is made, the game should call all previously registered `move` callbacks, passing in the game's current `gameState` as an argument to the function.

6. `onWin(callback)`: Takes a callback function as input and registers that function as a listener to the `win` event. When the player wins the game (by making a 2048 tile), the game should call all previously registered `win` callbacks, passing in the game's current `gameState` as an argument to the function.

7. `onLose(callback)`: Takes a callback function as input and registers that function as a listener to the `move` event. When the game transitions into a state where no more valid moves can be made, the game should call all previously registered `lose` callbacks, passing in the game's current `gameState` as an argument to the function.

8. `getGameState()`: Returns a accurate `gameState` object representing the current game state.

2. The View and Controller

Once your `Game` class is working, the next step is to hook up your newly finished Model to a View and Controller so that users on the web can play your game. For this portion of the assignment, you need to make a web page using JavaScript, HTML, and CSS that drives your `Game` class.

After the assignment due date, an LA will manually load your 2048 game page into a Google Chrome window and will try to play the game. Your user interface doesn't need to be perfect and there doesn't need to be animations, but the game must be playable for full credit. In particular, the following elements must be present in your user interface:

1. When the page first loads, the game must be initialized to a random starting state.
2. The player must be able to input moves by pressing the arrow keys on the keyboard.
3. The game board must be displayed on the page and updated as moves are made.
4. The score must be displayed on the page and updated as moves are made.
5. A button must be included in the user interface that allows the player to reset the game to an initial starting position.
6. When the game is over, the user interface must visually display that the game is over.
7. If the player wins the game, the user interface must visually display that the game is won.

Note: Your user interface is only required to work with a 4x4 board; no need to program other board sizes into your HTML.

3. Rubric

The autograded portion of the assignment (i.e. the `Game` class) represents 35% of the grade; the hand-graded portion (i.e. the user interface) will consist of the remaining 65%. **In order to receive ANY credit for the hand-graded portion, you must first fully complete the autograded portion.** In other words, an LA will not look at your user interface until after you have a working `Game` class.

Below is a breakdown of how the points will be distributed for this assignment.

35% -- The `Game` class works (autograded portion)

15% -- The user interface responds to arrow key presses and moves the game correctly

15% -- The user interface displays the game board and updates it correctly

10% -- The user interface displays the game score and updates it correctly

10% -- The user interface includes a working "reset" button which resets the state of the game when pressed

5% -- The user interface includes a brief set of instructions for newbies (this can be as easy as a single sentence, like in the reference game)

5% -- The user interface indicates when the game is over or won

5% -- The production-ready final product doesn't have any leftover debugging **alert()** or **console.log()** code

4. Submission

Submit your final 2048 code, complete with user interface and game class, using `npm run submit a07` from the command line just like we have been doing all semester. In particular, note the following:

1. Include an `index.html` file with your submission that loads the 2048 user interface and allows the game to be played. This is what the LAs will use to play your game when they hand-grade your assignment.
2. Do not include any DOM manipulation code in your `engine/game.js` file. Instead, make a separate `.js` file to hold your Controller/View code.
3. It's totally okay to use jQuery or any similar JavaScript libraries in your code. Just don't put them in `game.js`.

   When the autograder runs, it loads your `game.js` file **without a DOM**, so it won't have access to any of the usual browser-environment global variables such as `window` or `document`. As a result, you'll crash the autograder if you try to access these variables from within the `game.js` file. This is why you have to put your DOM manipulation code in a separate `.js` file.