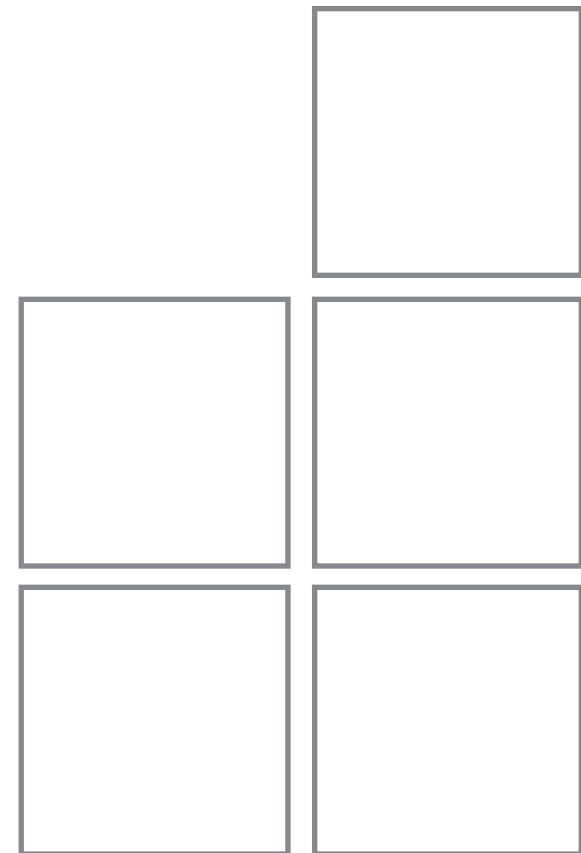




Recreating Nostalgic Computer Games in JavaScript

Sara Gorecki
@opheliasdaisies



Who I Am

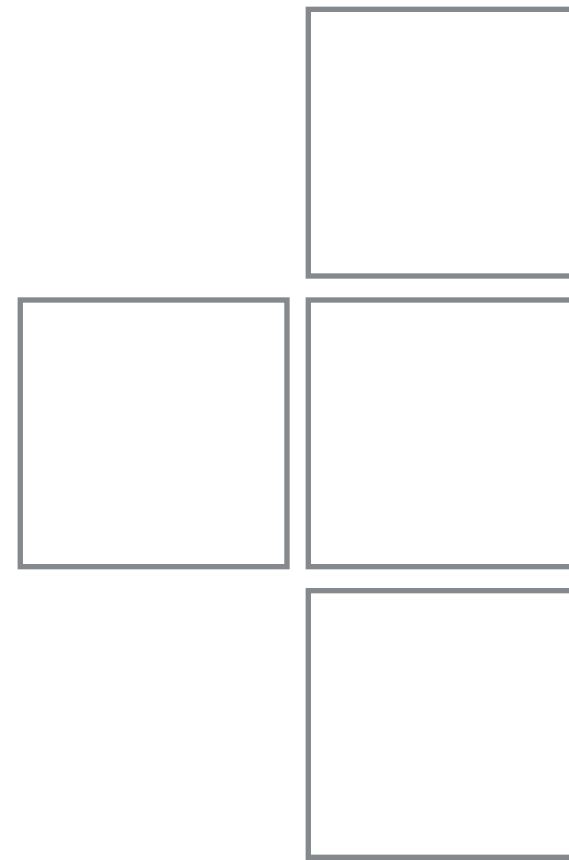
@opheliasdaisies

Node Engineer at Penton Media

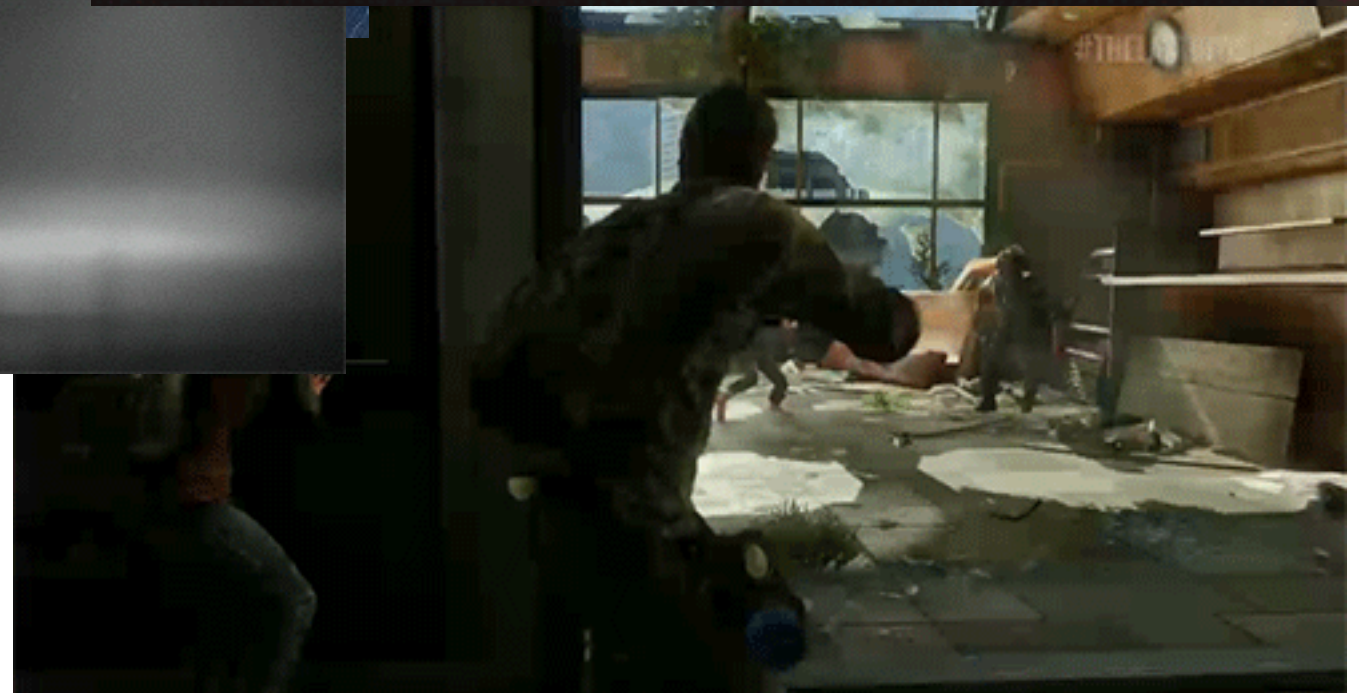
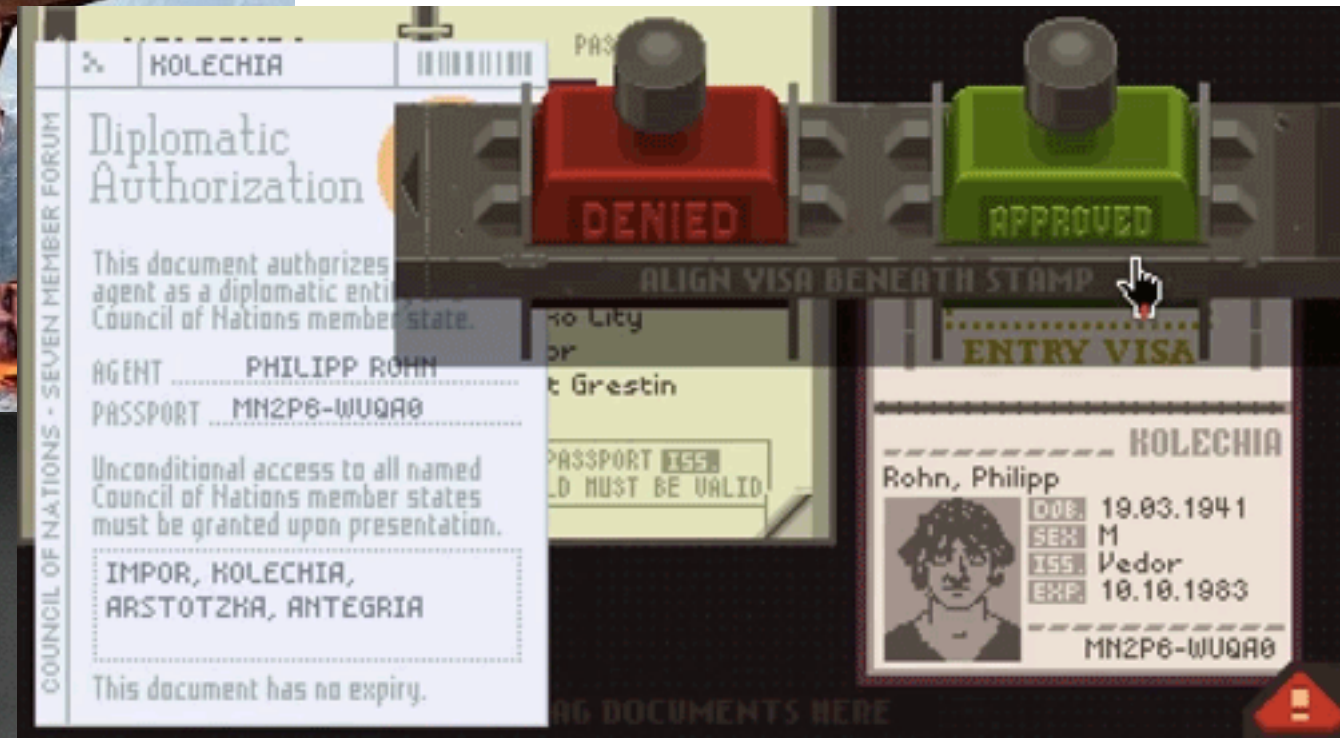
Flatiron School Graduate

Born and Raised in NYC

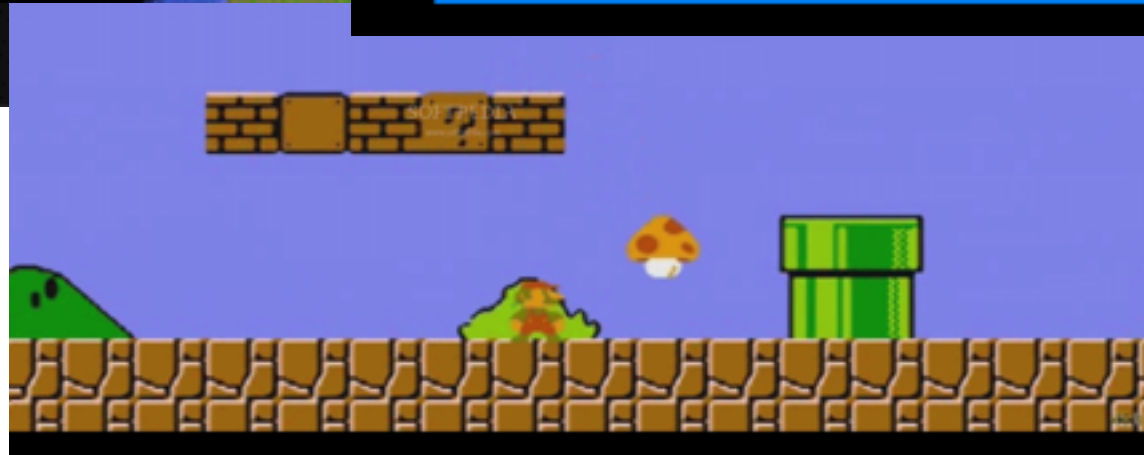
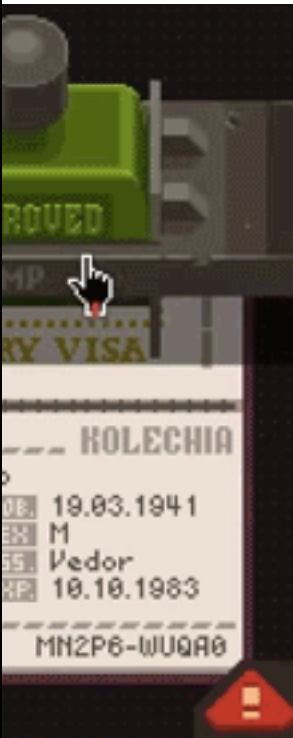
& I love video and computer games



Computer Games are Awesome!

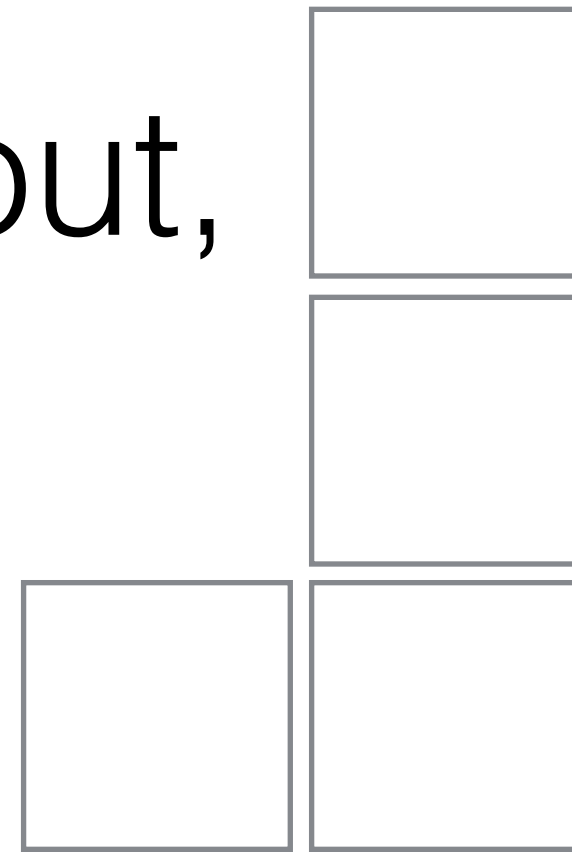


Computer Games are Awesome!

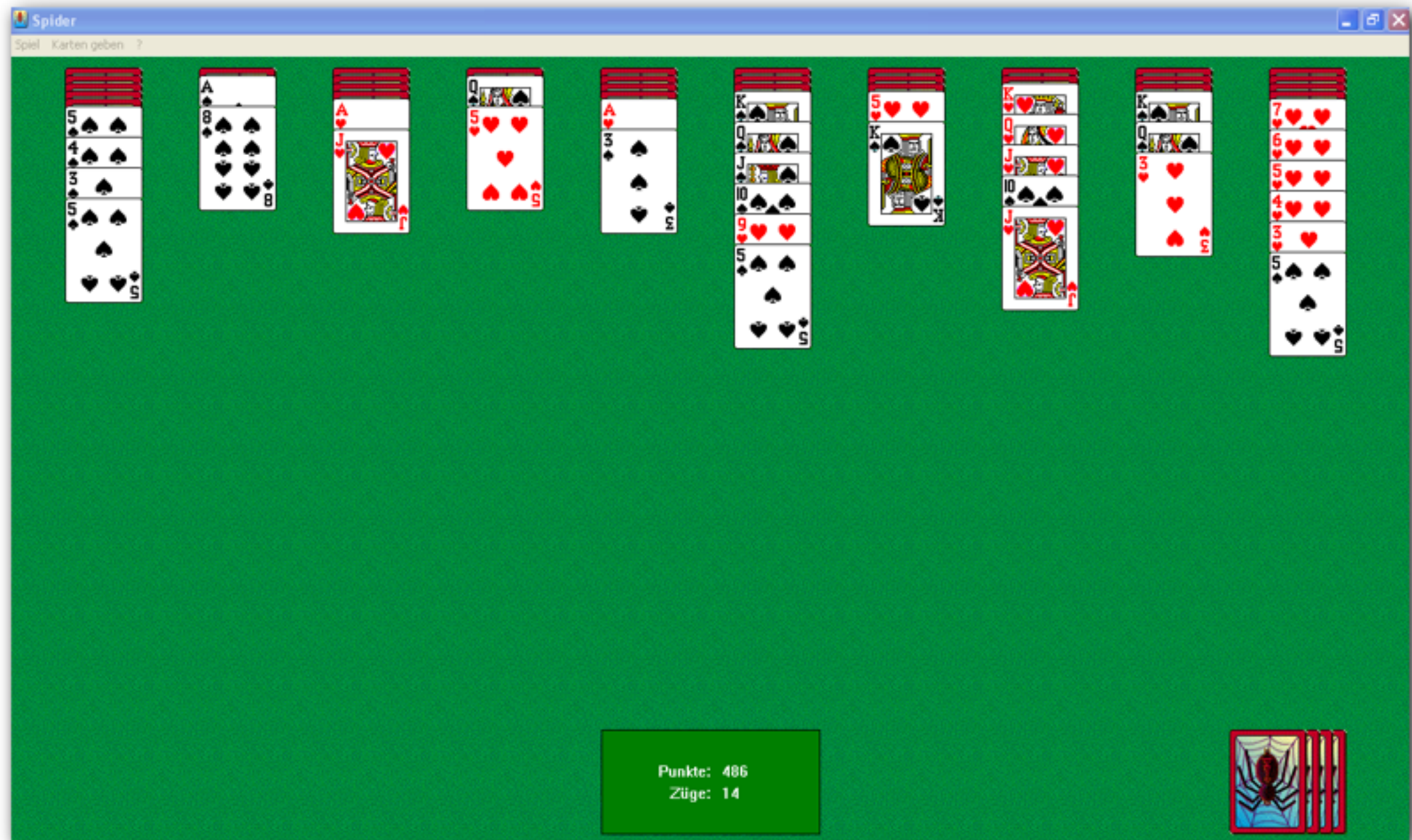


So what's this talk about, anyway?

- Logic patterns and planning a game
- Rendering your game
- The game engine event loop
- Managing repeating elements (pieces, cards, etc.)
- Perils and pitfalls
(mistakes are okay)

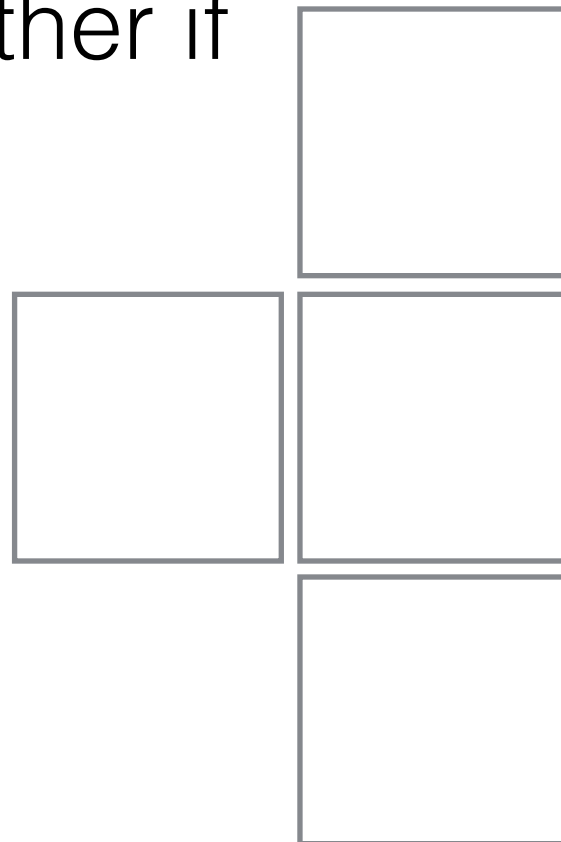


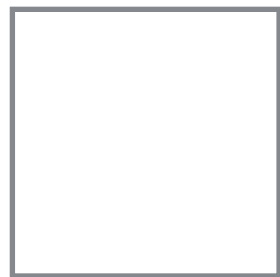
Let's Talk Spider Solitaire



The Rules

- Remove all cards from the table
- Cards can be stacked by value (ie. 2 on a 3)
- Cards of the same suit can be moved together if they are of consecutive, ascending values
- Additional cards can be dealt 10 at a time, but only if no columns are empty.





The Deck



- Two decks of cards, 54 dealt to start
- Three Difficulty Levels, based on the number of suits

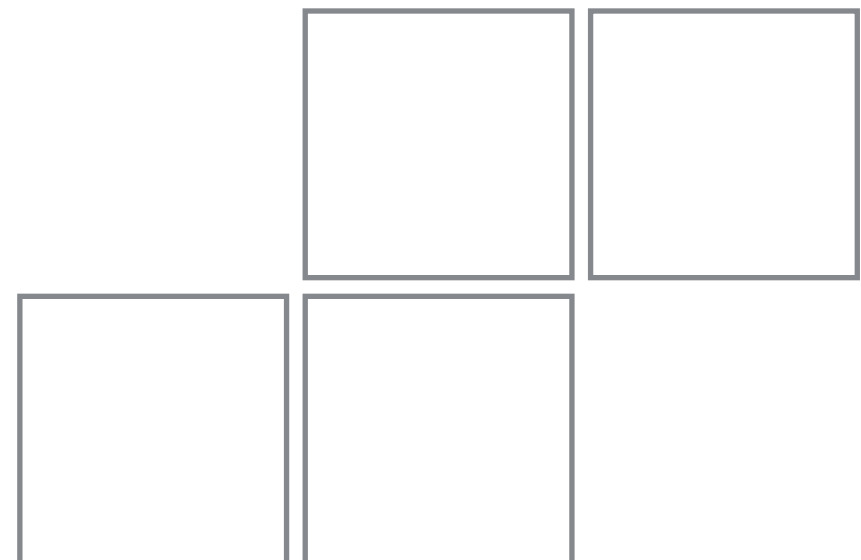
- Each card has:

- Value (1-13)

- Suit

- Face (A-K)

- Color



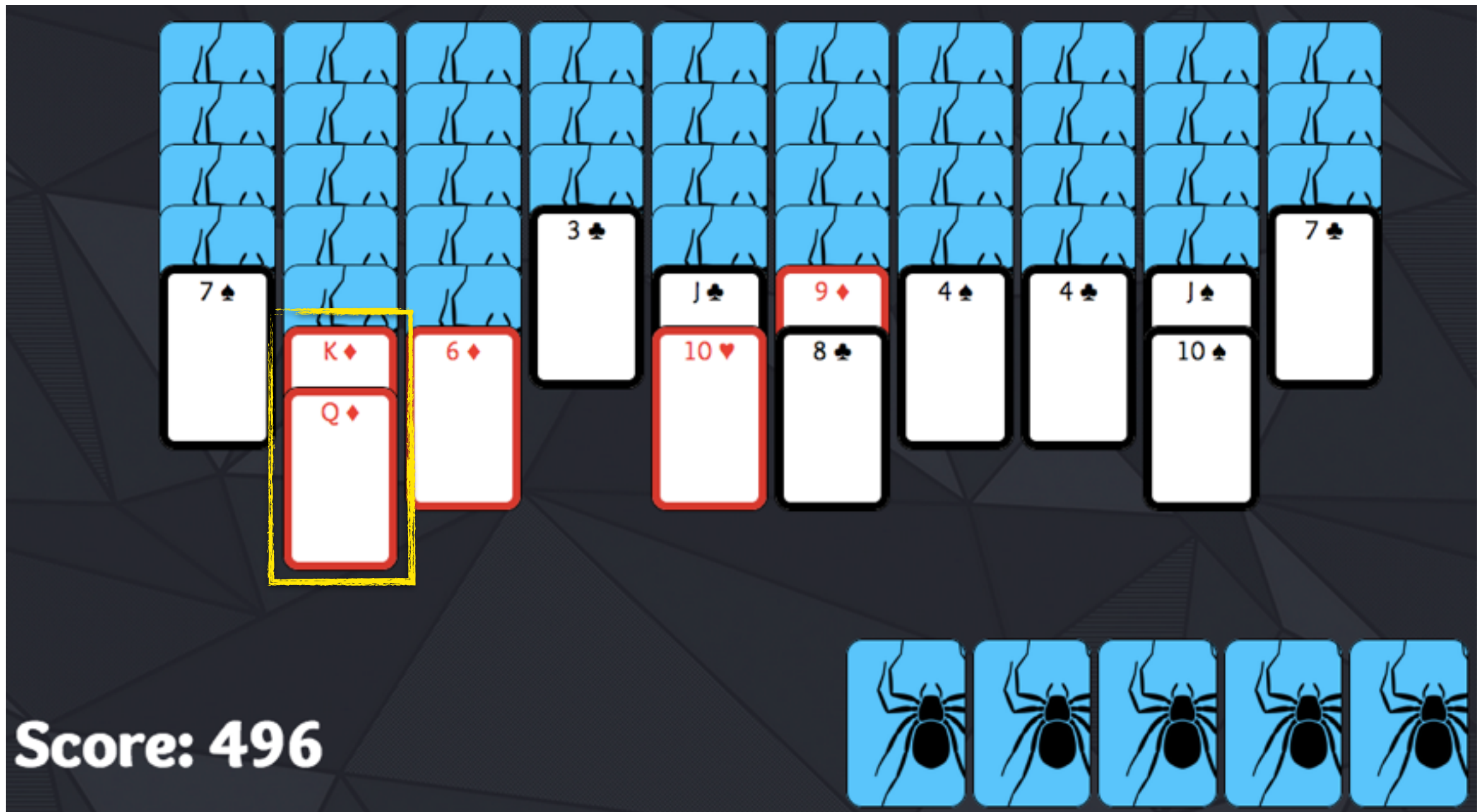
Planning the Game



- Didn't want to have to create cards each time
- Spider Solitaire has no complex timings or animations
- Everything in the game happens on page load or a click event



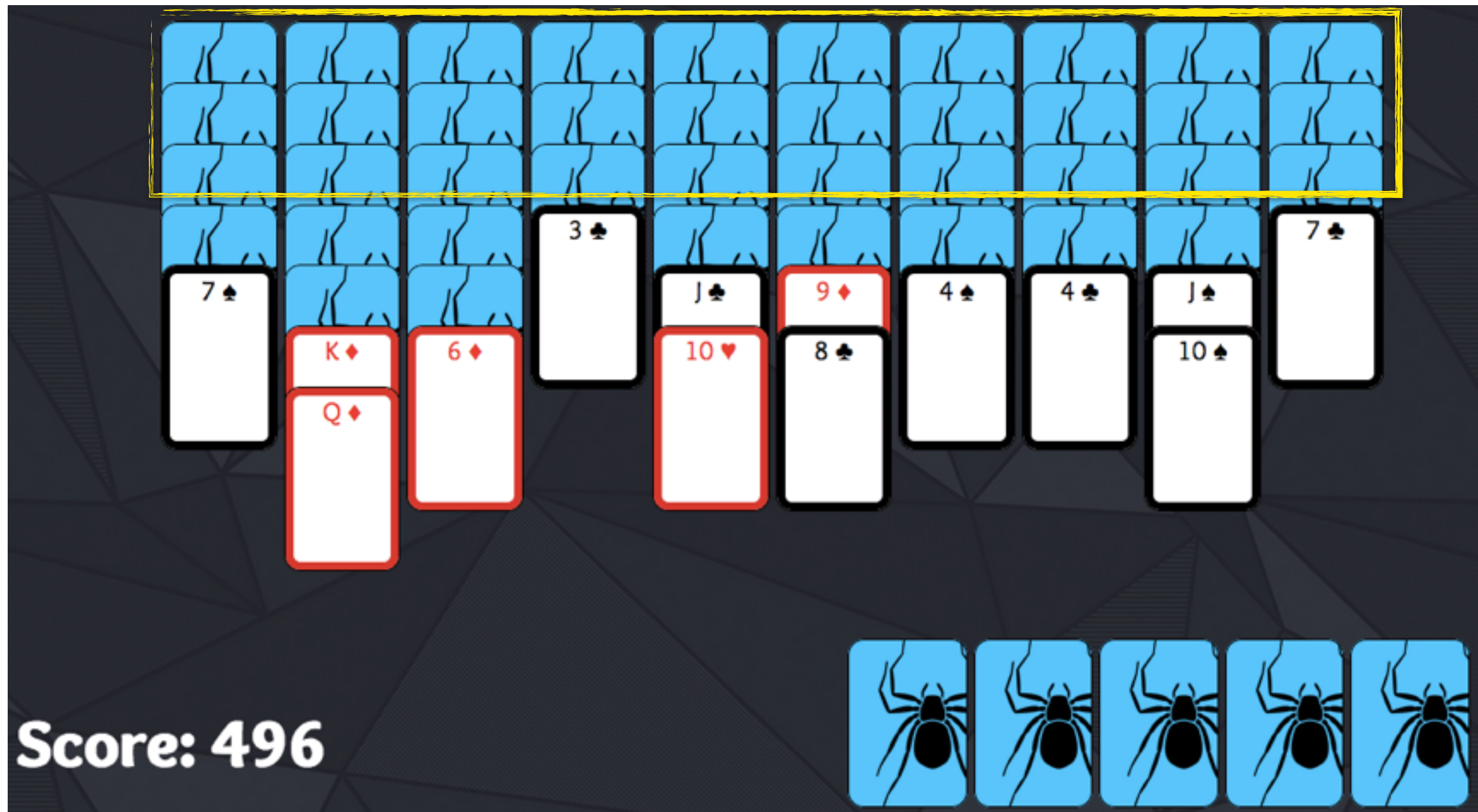
The States of Things



Active/Faceup Cards

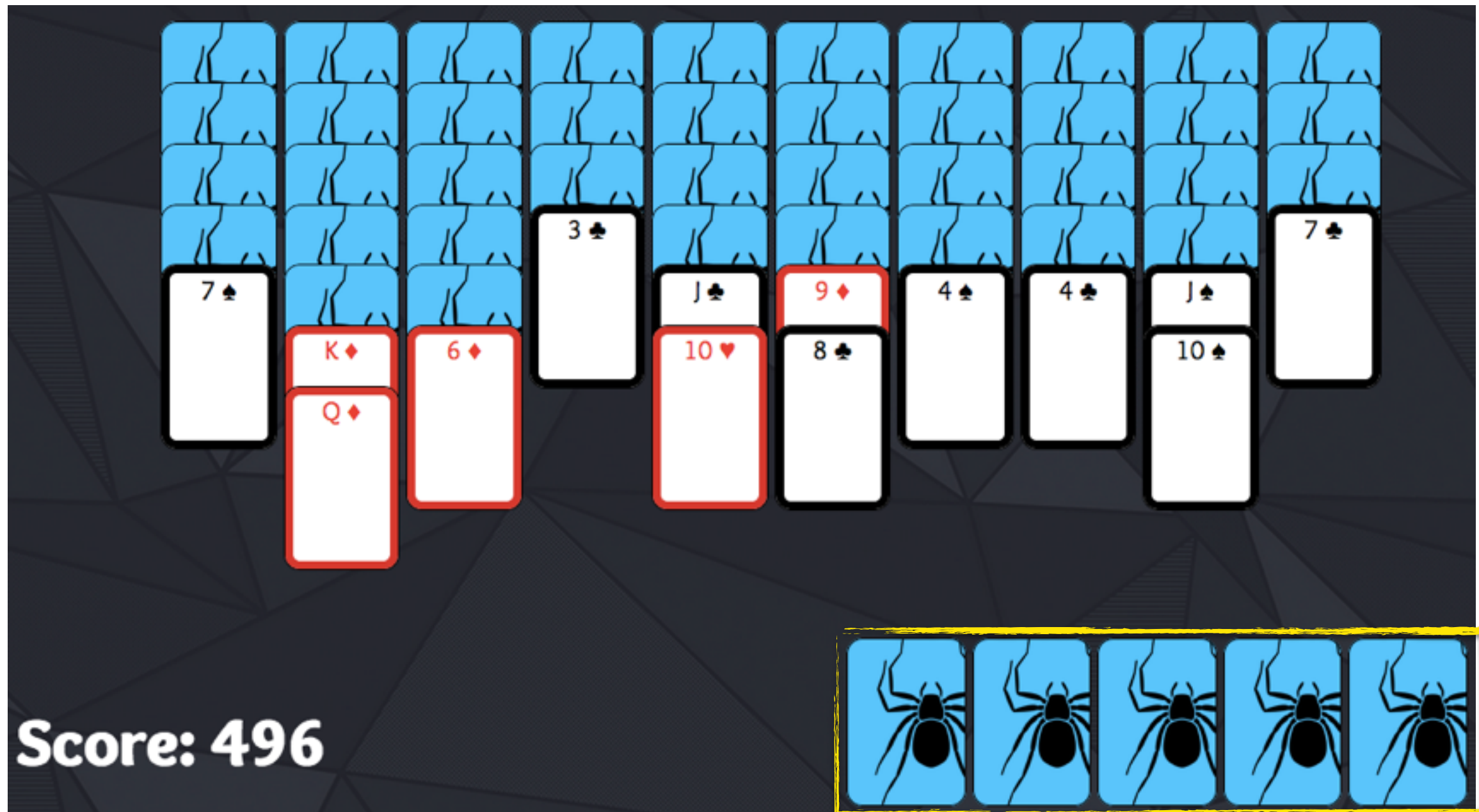
Selected Cards

The States of Things



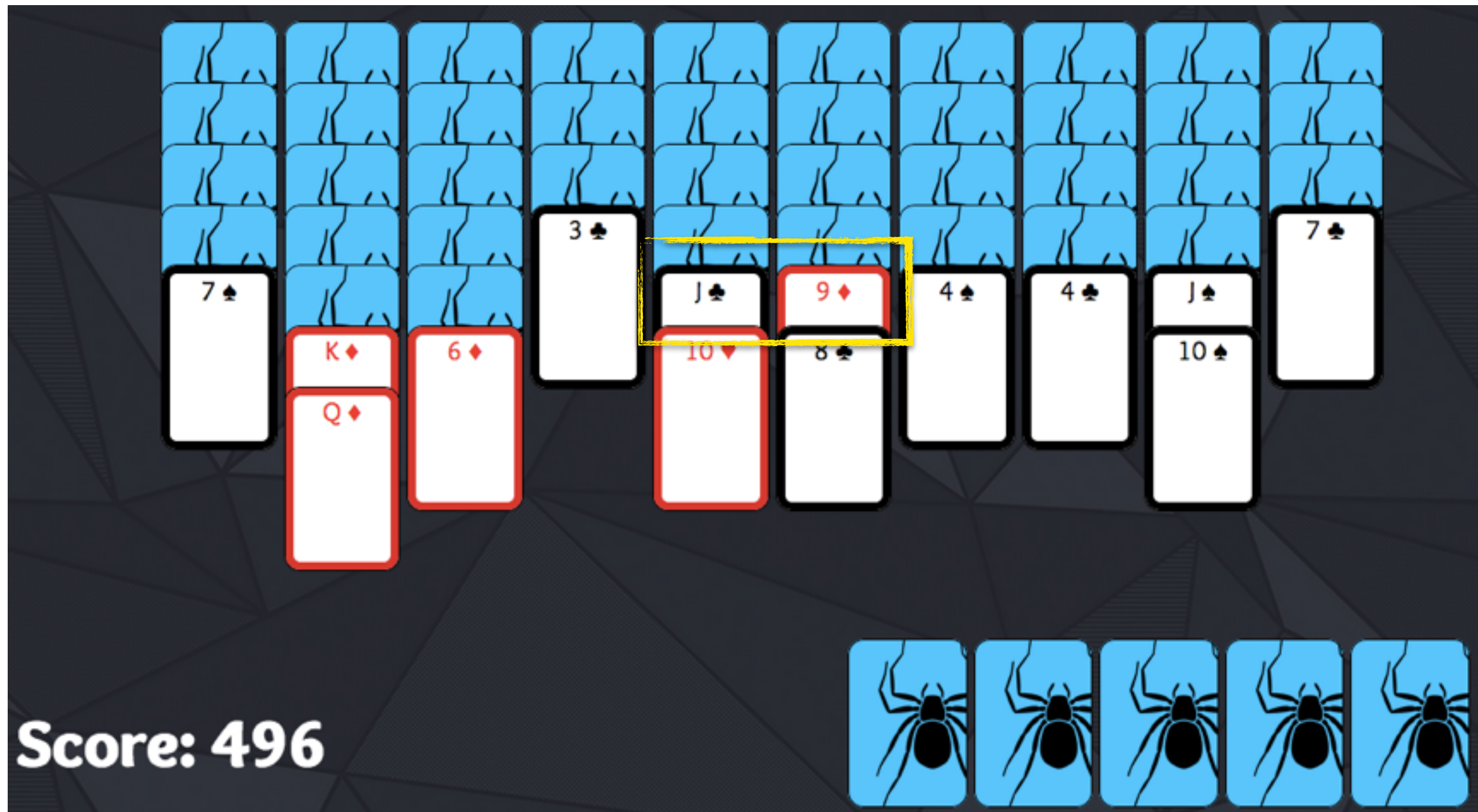
Facedown Cards

The States of Things



Reserve Cards

The States of Things



Blocked Cards

Let's Break It Down



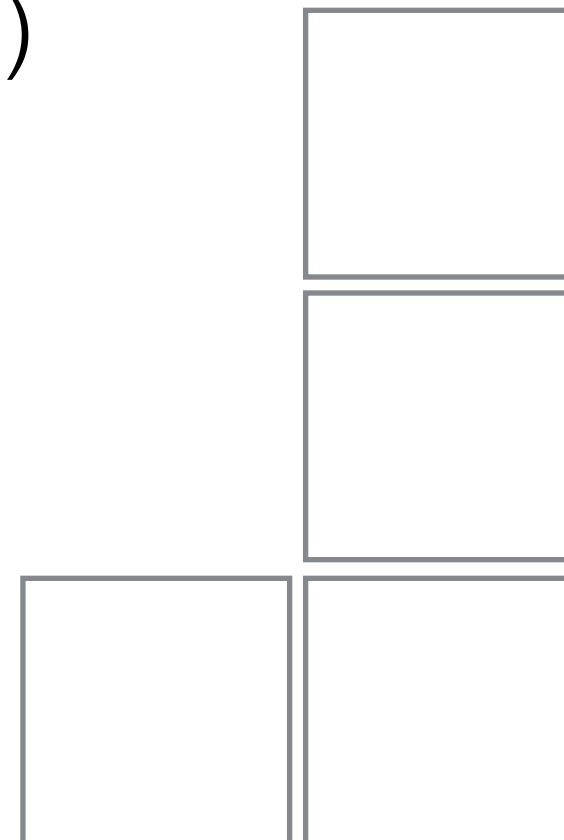
- \$(' .card')

- \$(' .faceDown')

- \$(' .reserve')

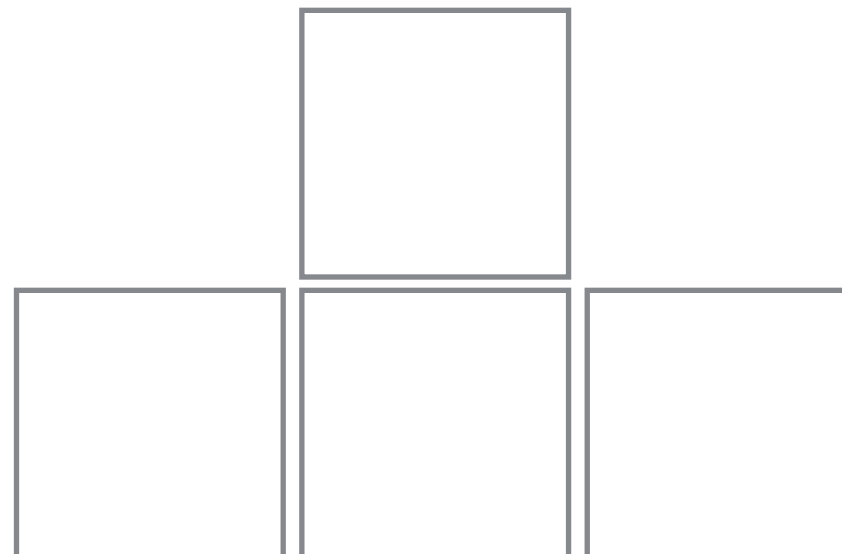
- \$(' .selected')

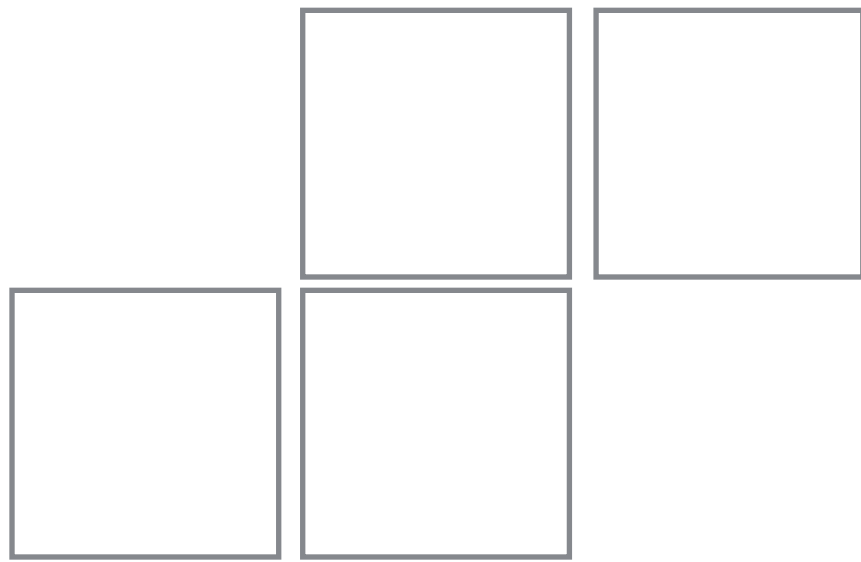
- \$(' .blocked')



Dealing the Deck

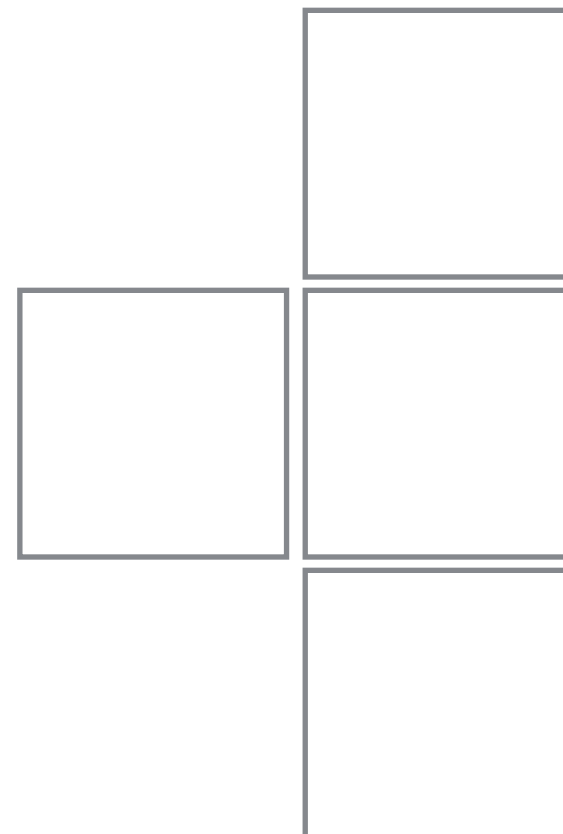
- YAML file had all card data
- Shuffled it and served it to the client
- Each column is a div with class column
- Each card is a div with child divs for the card properties
- Dealt cards facedown



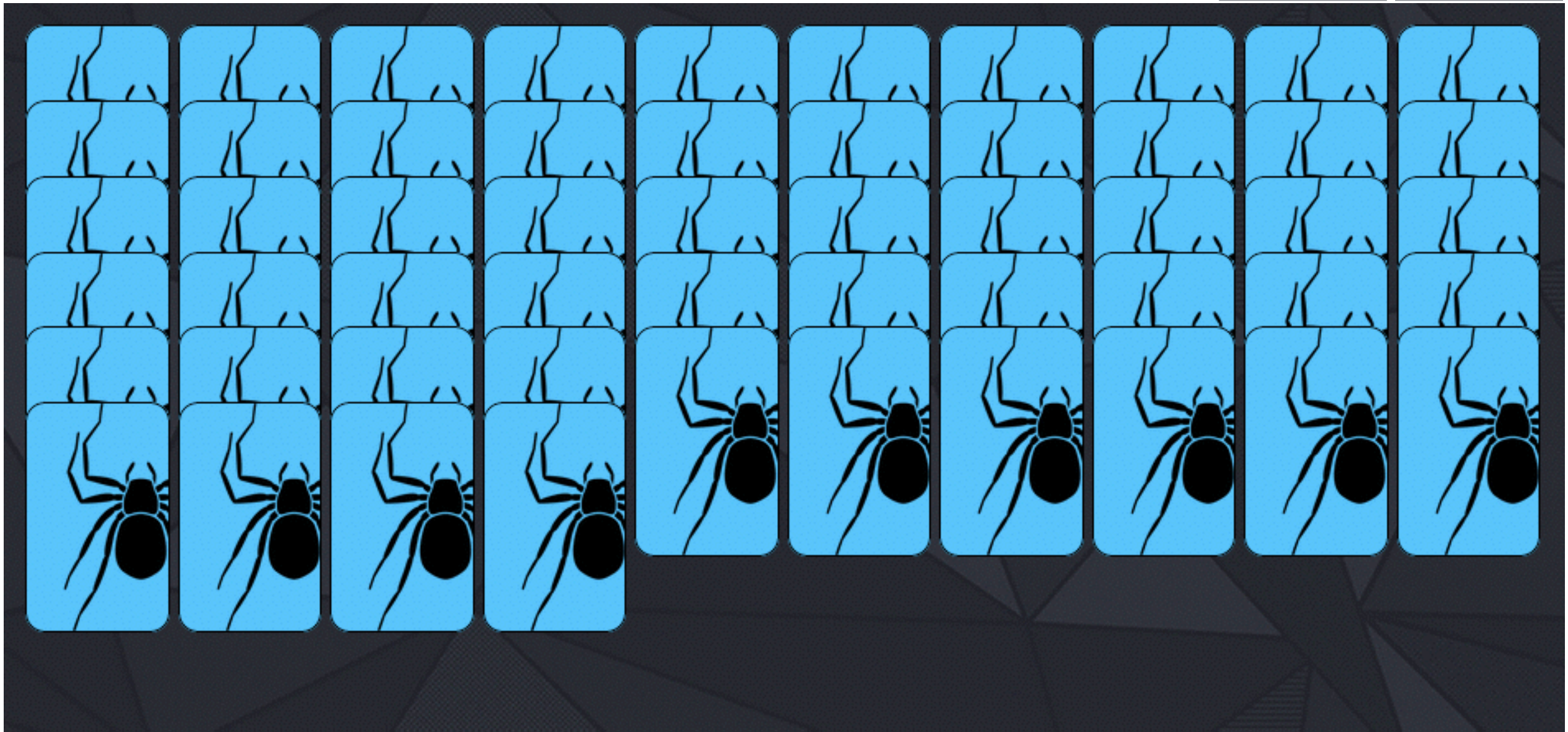


Partials

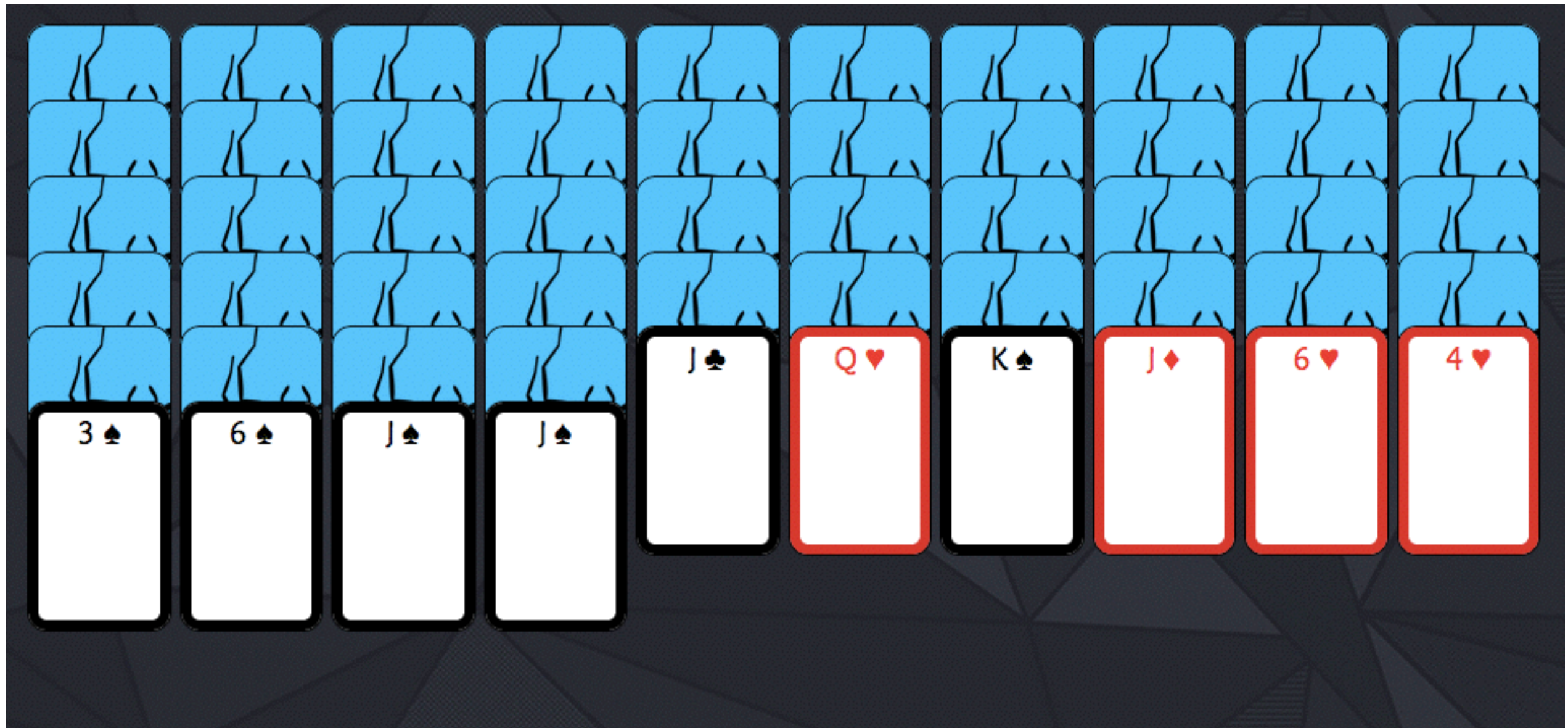
- Created partials so as to not re-create each card
- Each card div has children with the classes 'value', 'face', 'color', 'suit'
- 4 columns of 6 cards each
6 columns of 5 cards each



On to JavaScript!



On to JavaScript!



```
$(".board .column").on("click", ".card", function(){
```

```
var $this = $(this);
```

```
var $selected = $(".selected");
```

```
if(!$this.hasClass("faceDown") && !$this.hasClass("blocked")){
```

```
//for when nothing has been selected
```

```
if($selected.length == 0){
```

```
//select card and cards below
```

```
$this.addClass("selected");
```

```
$this.nextAll().addClass("selected");
```

```
} else {
```

```
var $firstSelected = $($selected[0]);
```

```
// for when cards have been selected
```

```
if($this.hasClass("selected")){
```

```
//remove selected if clicked on selected
```

```
$selected.removeClass("selected");
```

```
} else if (+$this.find(".value").text() == +$firstSelected.find(".value").text()+1){
```

```
var $blockedInCol = $firstSelected.prevAll(".blocked");
```

```
//turn uncovered card face-up
```

```
$selected.prev().removeClass("faceDown");
```

```
//unblock cards
```

```
unblockCards($blockedInCol);
```

```
//move selected card to new column if values allow it
```

```
$selected.appendTo($this.parent());
```

```
$(".card").removeClass("selected");
```

```
findCardsToRemove(this);
```

```
//block card when card is off-suit
```

```
if ($this.find(".suit").text().trim() != $firstSelected.find(".suit").text().trim()){
```

```
    $firstSelected.prevAll().not(".faceDown").addClass("blocked");
```

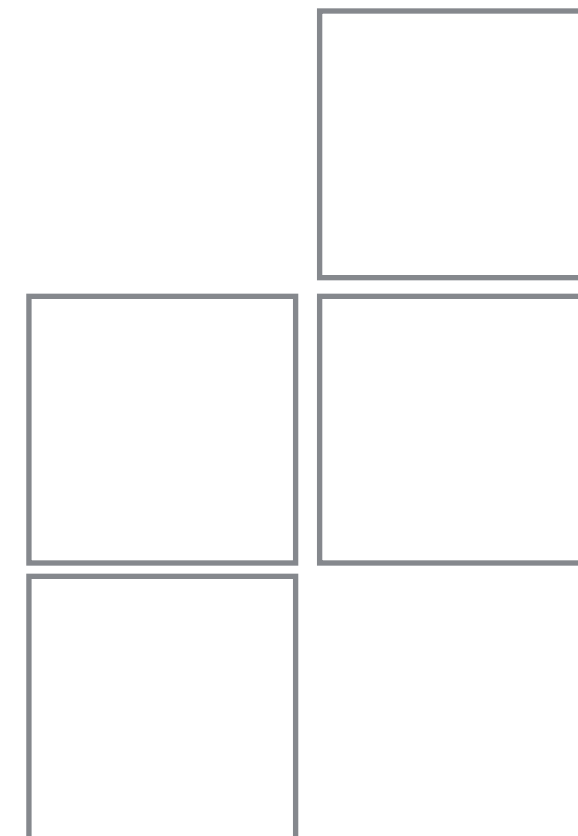
```
}
```

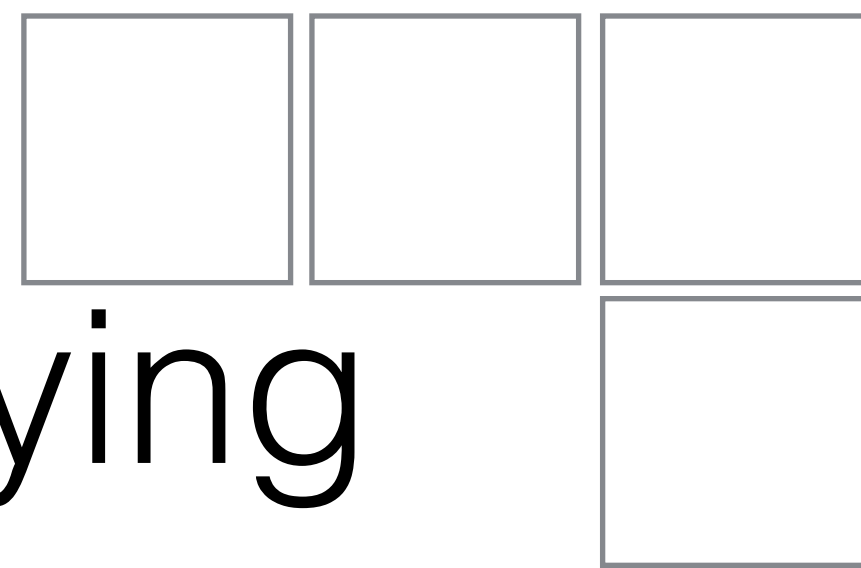
```
//decrease score by 1
```

```
$(".score").text(Number($(".score").text() - 1));
```



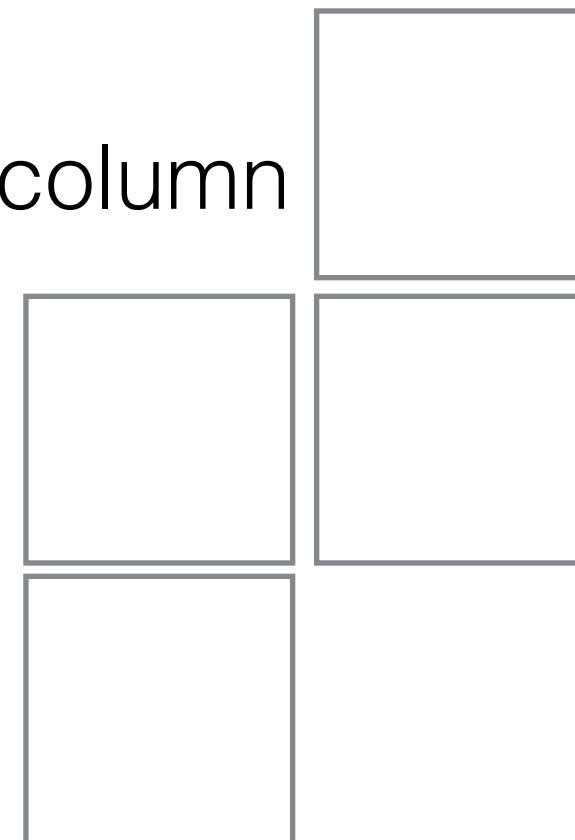
(Don't worry about it.)



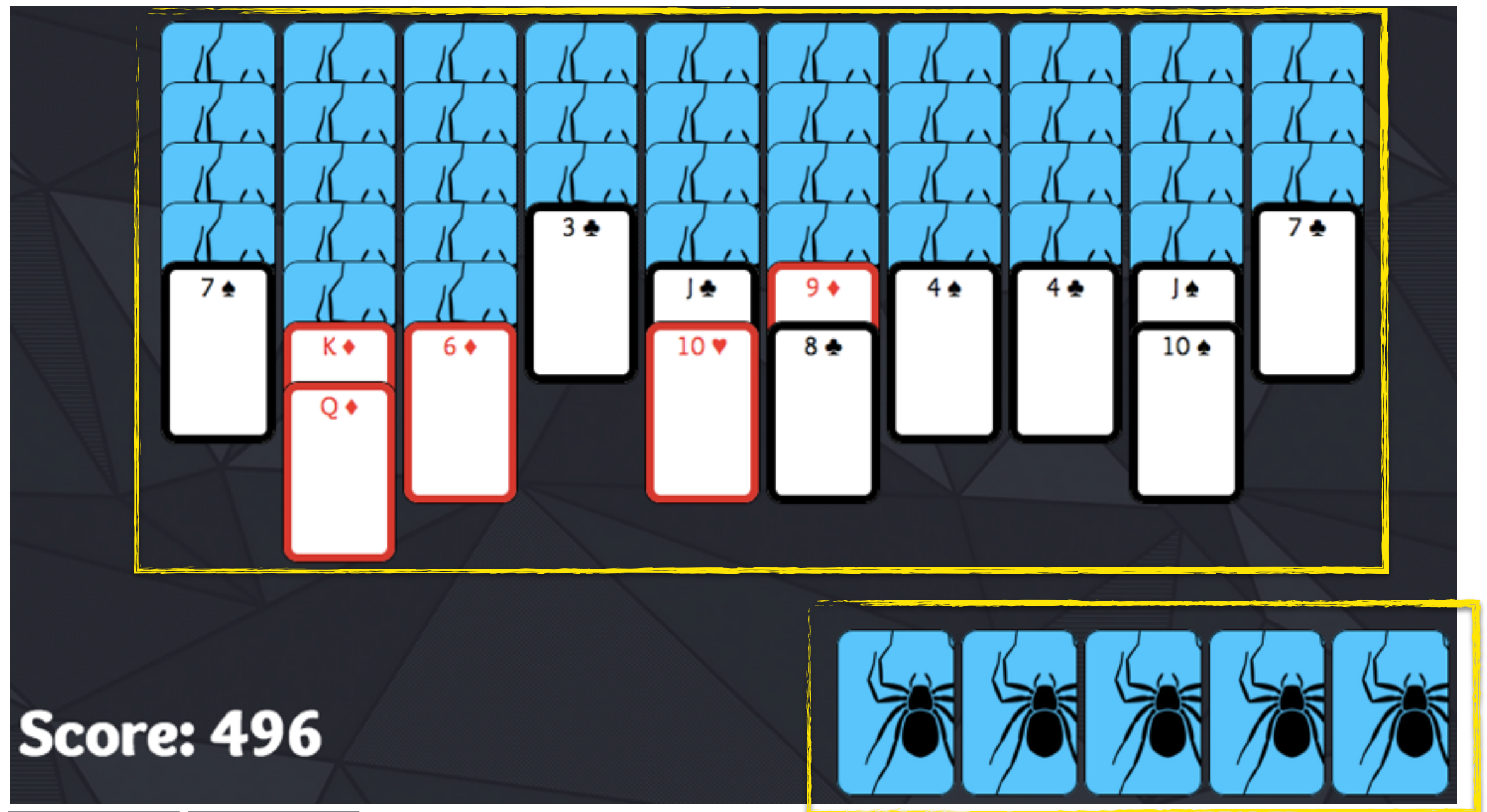


So Let's Start Playing

- IF a card on the board is clicked and it does not have the class `faceDown`, and it does not have the class `blocked`
 - and IF the length of `selected` cards is 0
 - Then we can add the `selected` class to the clicked `card`, and any cards below it in the column
 - ELSE the selected card is moved, or not
 - This is where ALL the game logic lives!

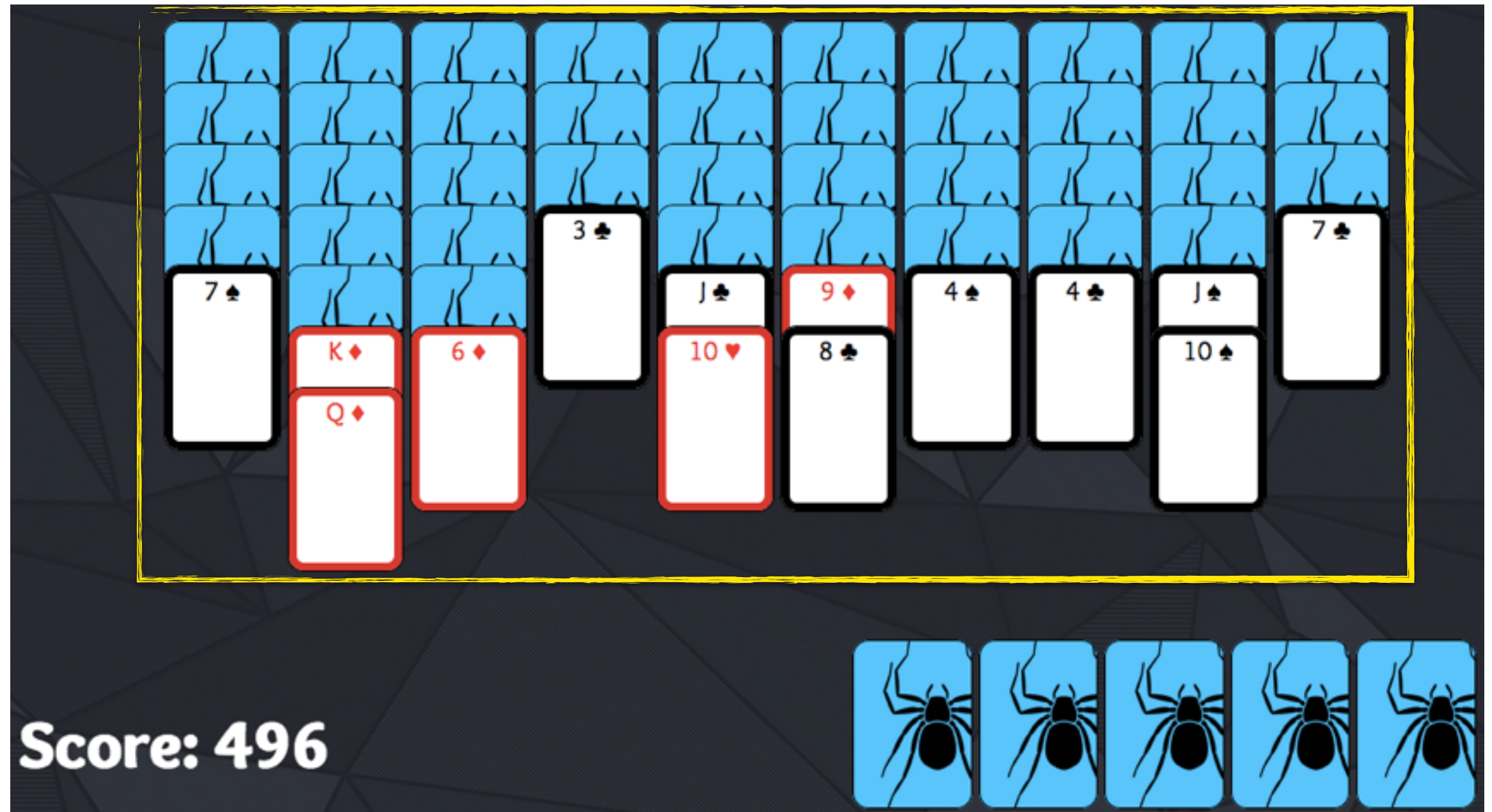


Or... maybe not



`$('.board .column .card')`

Or... maybe not



```
$('.board .column').on('click', '.card', function());
```

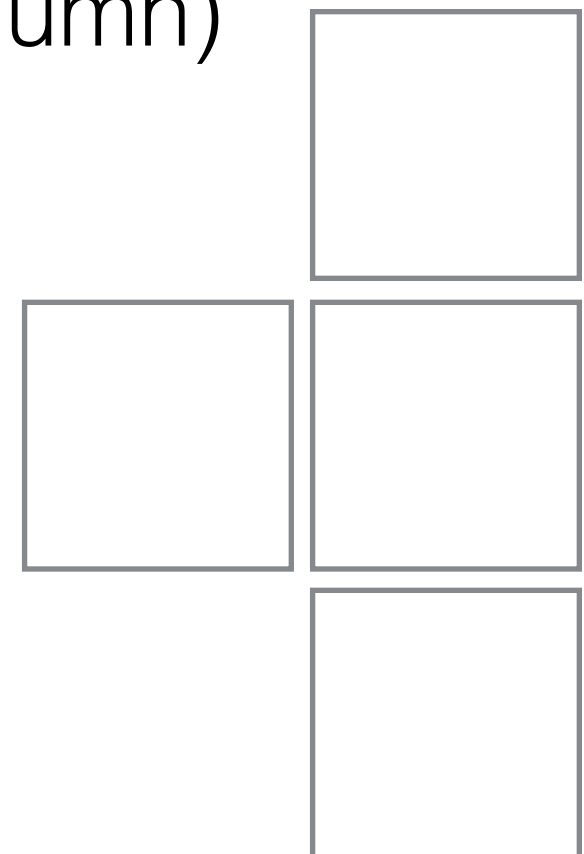
So what about the else? The



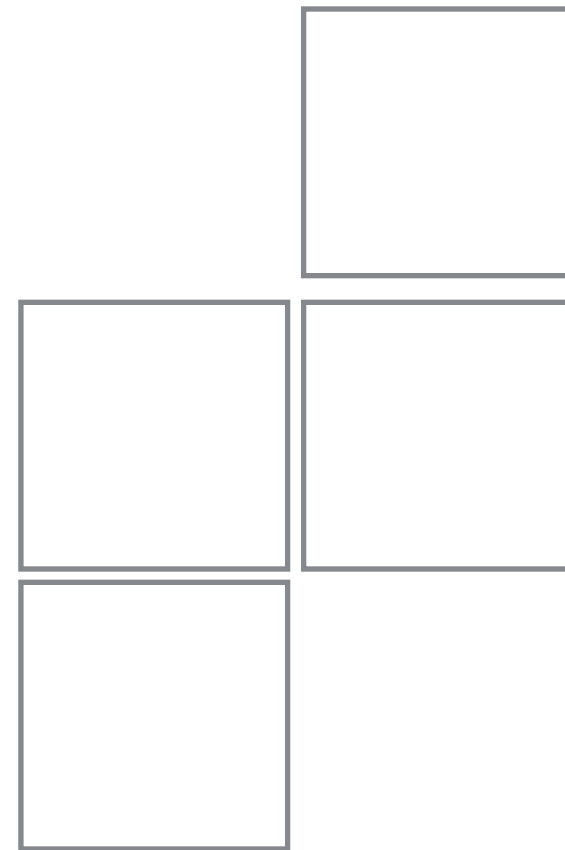
rest of the game logic.



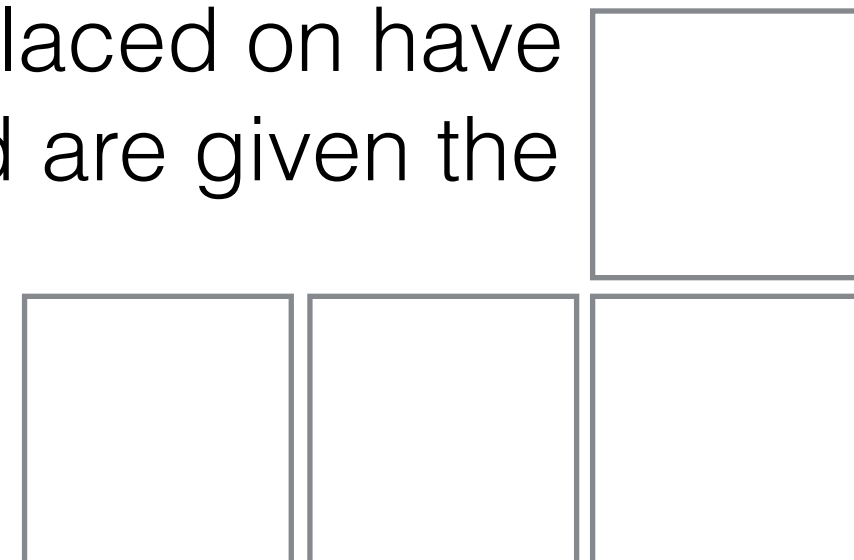
- When cards are selected, you can move them between columns by clicking again.
(remove class selected and appendTo column)
- Remove facedown class from the card underneath a card that has been moved



Validating Moves

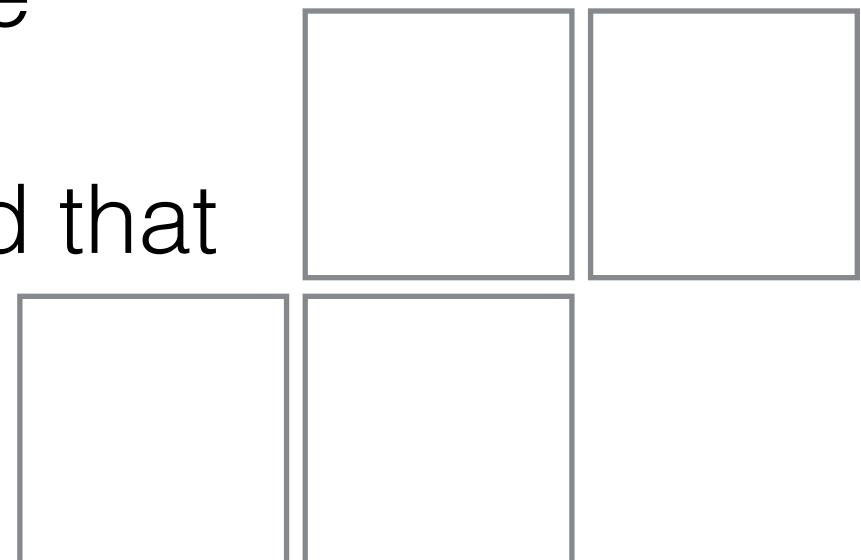


- Find the numeric value (1-13) of the selected card with the greatest value.
- The card clicked should have a value equal to the selected card +1.
- If not, definitely an invalid move.
- If the selected card and the card it is placed on have different suits, all cards above selected are given the class 'blocked'.
(But only if not facedown!)



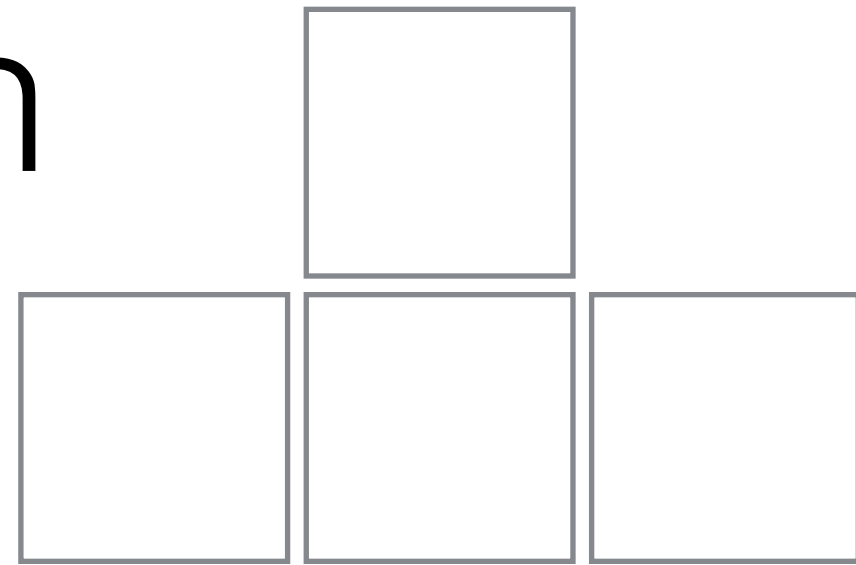
What is blocked must be
unblocked

- When a card is moved off the column, look at all the cards in that column with the 'blocked' class
- The top card always will become unblocked
- Look at each card going up the column, check the suit and value
- Stop once there's a card that can't be unblocked



How to Win

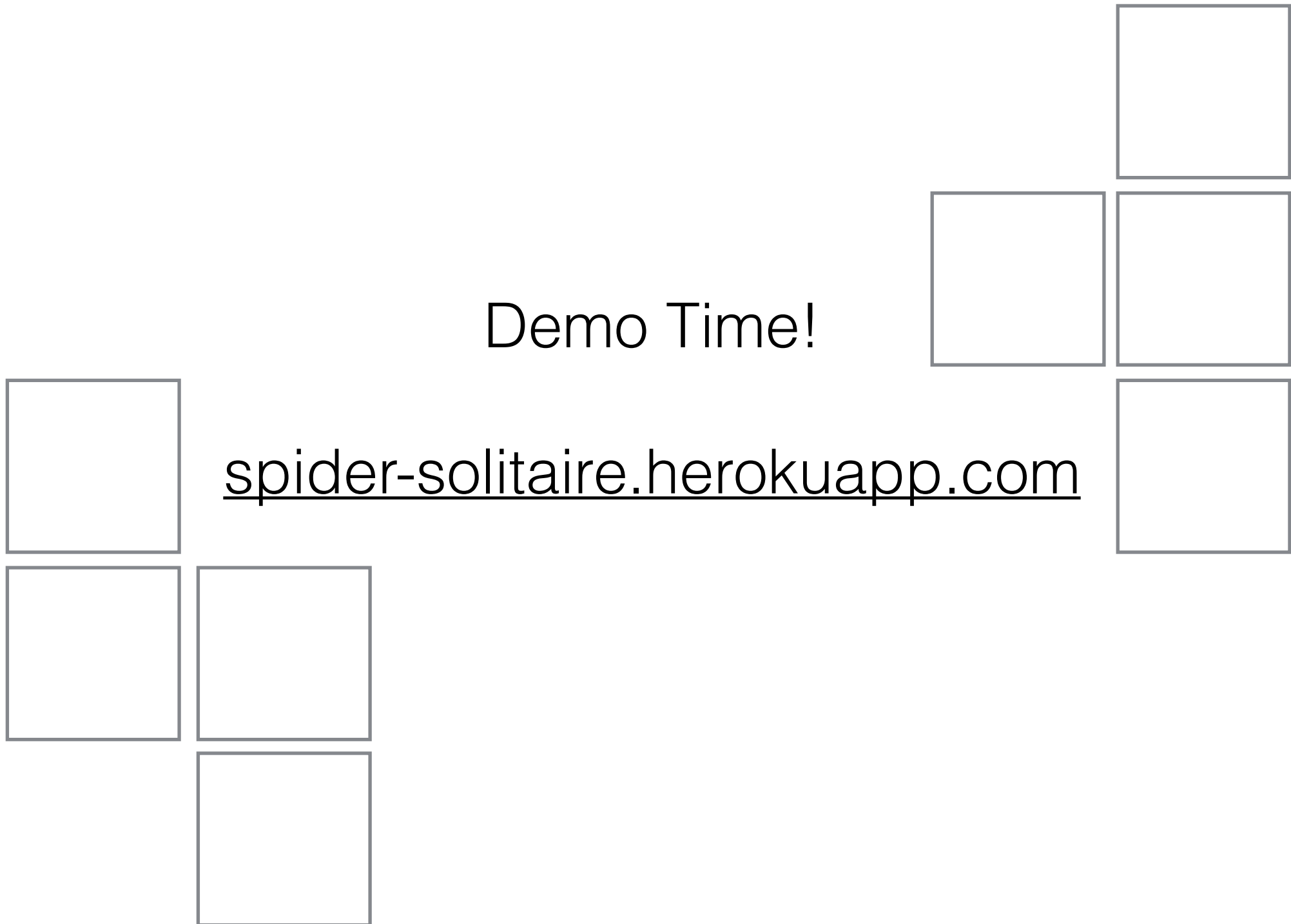
- Remove all the cards!
- When a card is moved, check the column for a King
- If there's a King, check each descending card for a value and suit match
- If you reach an Ace, congratulations!
- (Don't forget to turn the card under the King face-up.)



Let's Play!

Demo Time!

spider-solitaire.herokuapp.com



So... What now?



- By building Spider Solitaire, I learned how to make a DOM-based game.
- Appreciated the power of partials and templating.
- Learned how to think about general game logic.
- Made a DOM-based game!



So... What now?



- What if you wanted to make a game that included animation?
- What if you wanted to make a game that had multiple things happen at once?
- What if you wanted to make a game that relied on timing?



Time for..... Tetris!



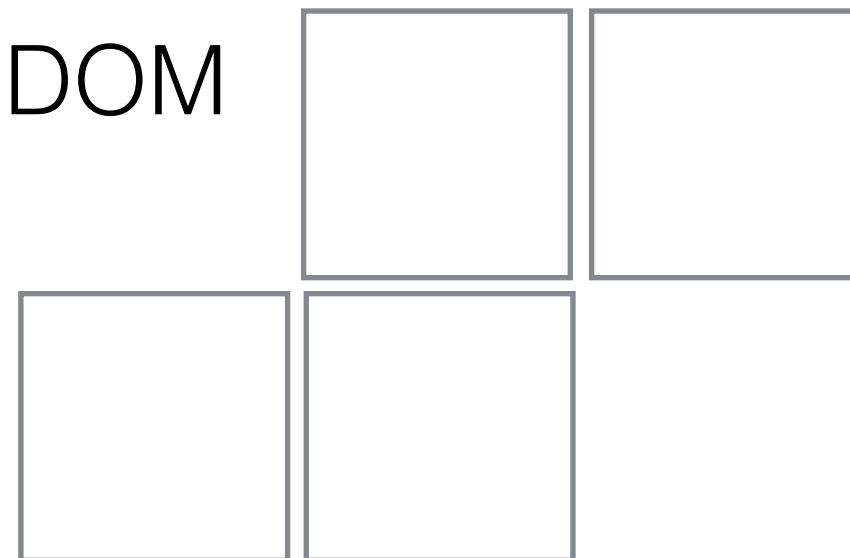
The Rules

- Pieces are randomly generated
- Pieces move down at regular intervals
- Pieces can stack
- Pieces can rotate and move left/right on player input
- Pieces cannot overlap other pieces or move off the grid

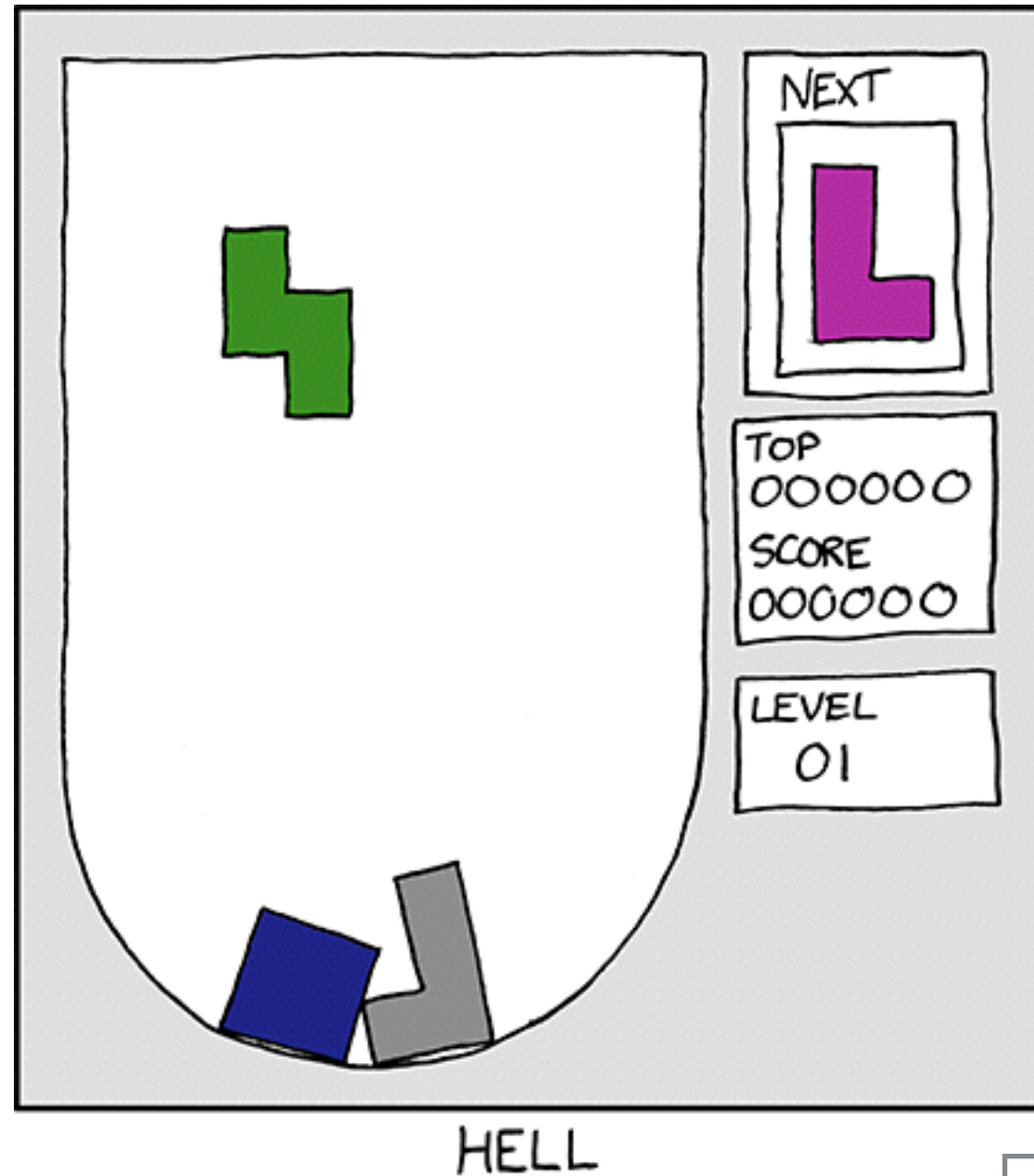


The Approach

- New challenges
- Constant animation
- Multiple things happen at once
- New approach
- Separate the game logic from the DOM
- Have an event loop for the game

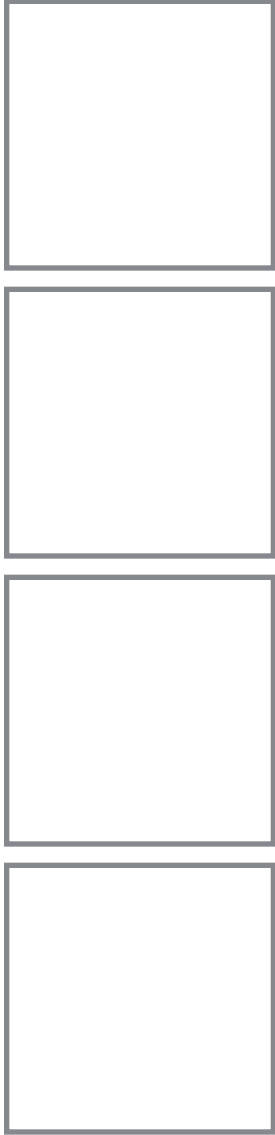


The Pieces and the Board



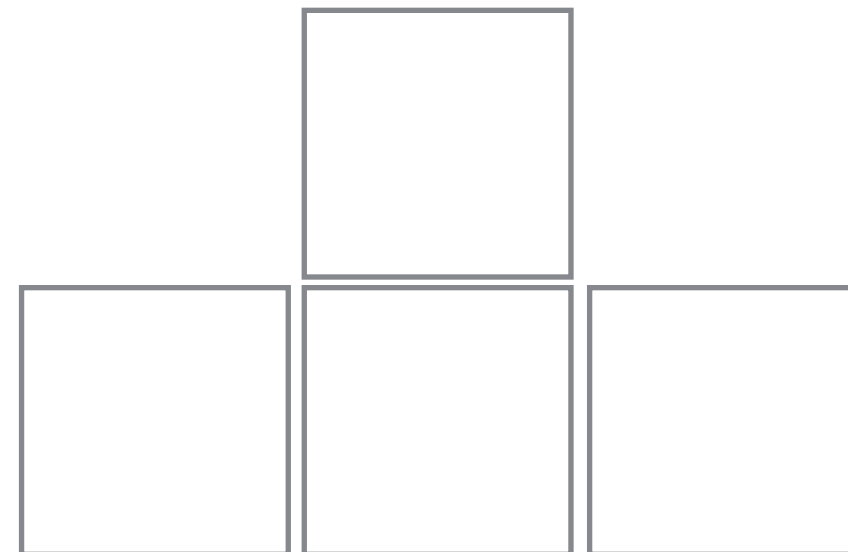
The Pieces and the Board

The Piece Constructor

- 
- Has a shape, chosen at random
 - Defaults to the configuration at Index 0
 - Is aware of the board it belongs to
 - Knows its starting coordinates on the board (midpoint on row 0)
 - Tracks if a piece is frozen

The Board Constructor

- The grid - a two-dimensional array, initially all undefined
- Has an active piece
- Knows if game is active
- Controls the speed of the game, both rendering frequency and the speed at which pieces fall



Rendering The Board



Tables! (jk)



Rendering The Board

- Handlebars templates!

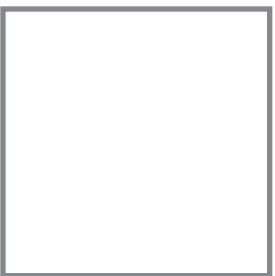


- In the Board constructor we compile the board

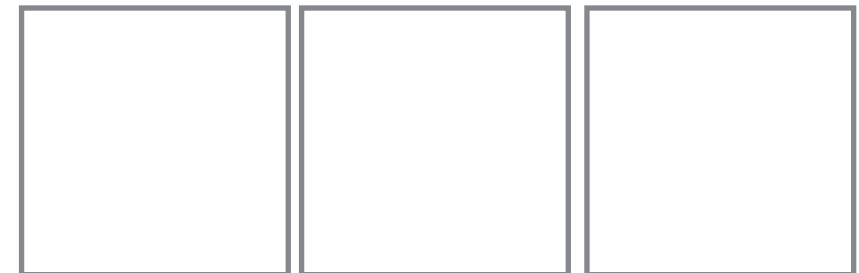
- `this.boardTemplate =
 Handlebars.compile($('#board').html())`

- `$('#.board').html(this.boardTemplate(this));`

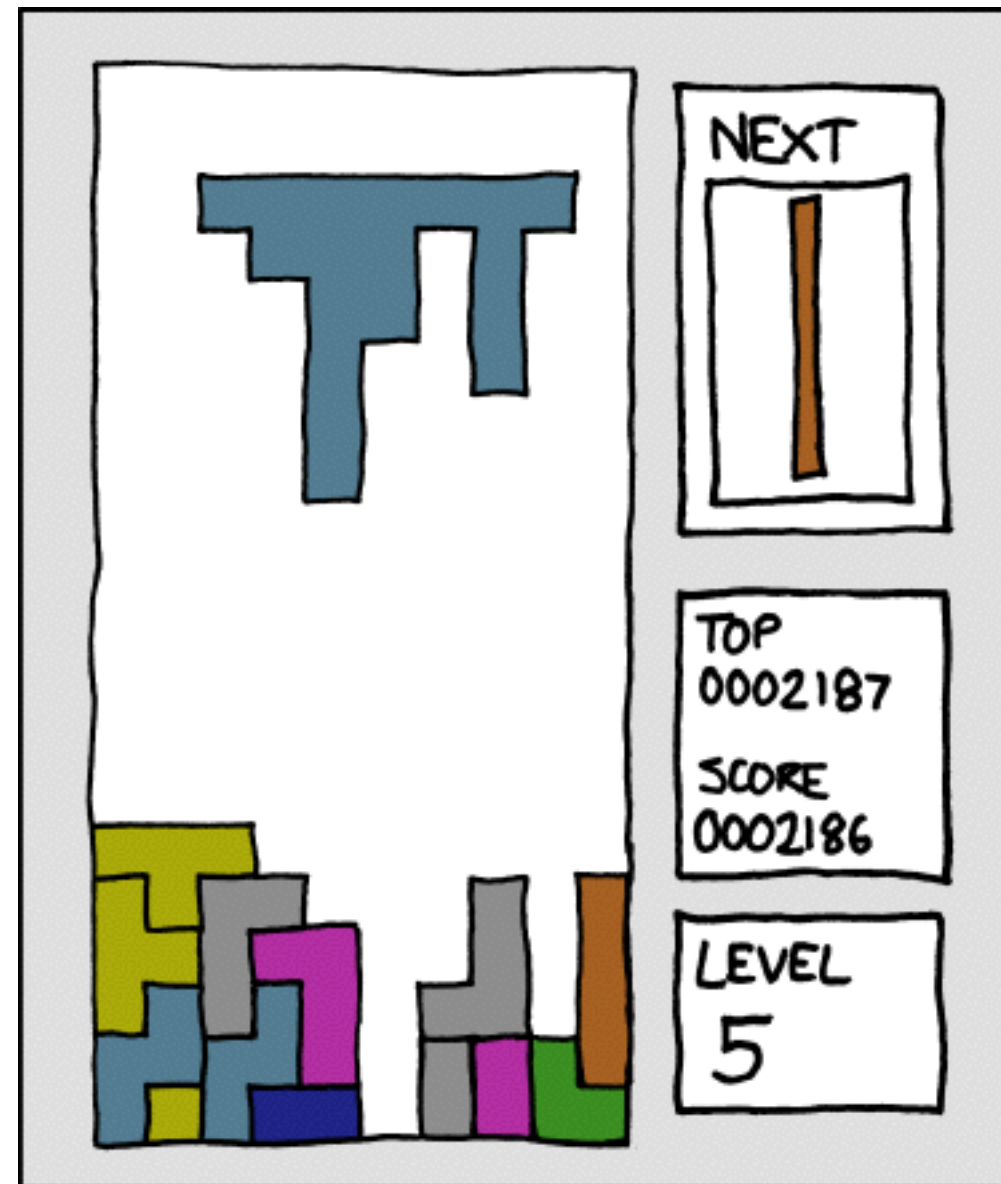
- In our html we iterate over our grid with `{{#each }}`



- `{{#if . }}` apply the class 'active'

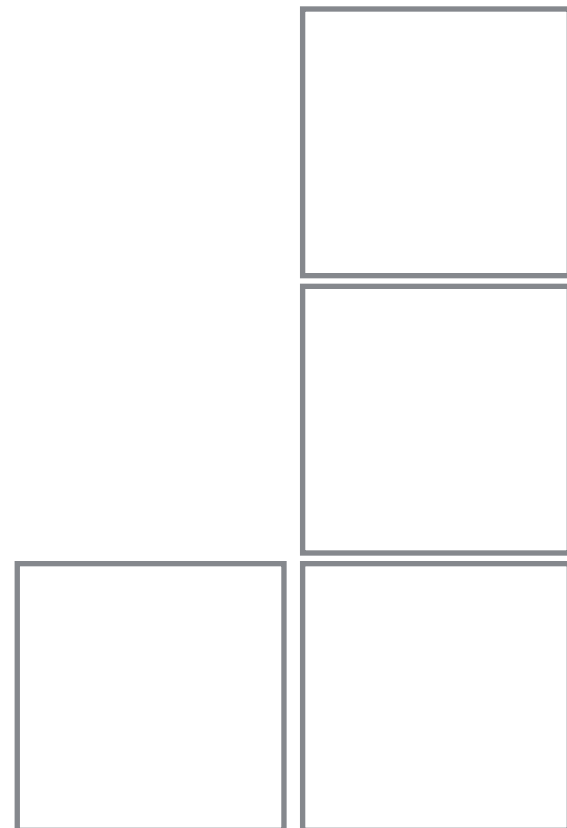


The Pieces

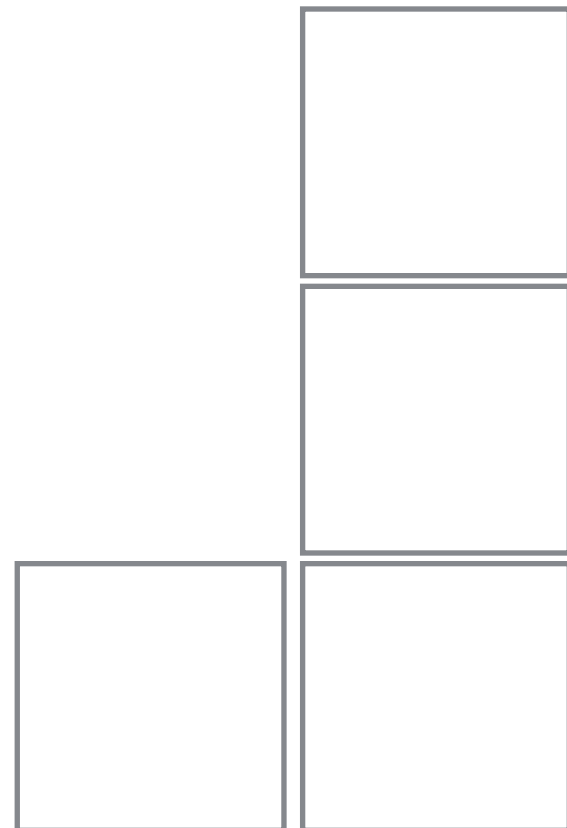


HEAVEN

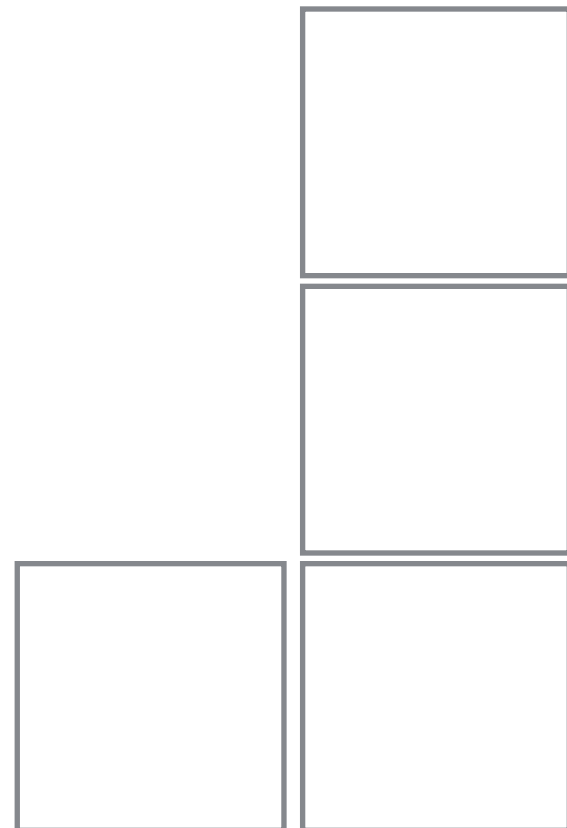
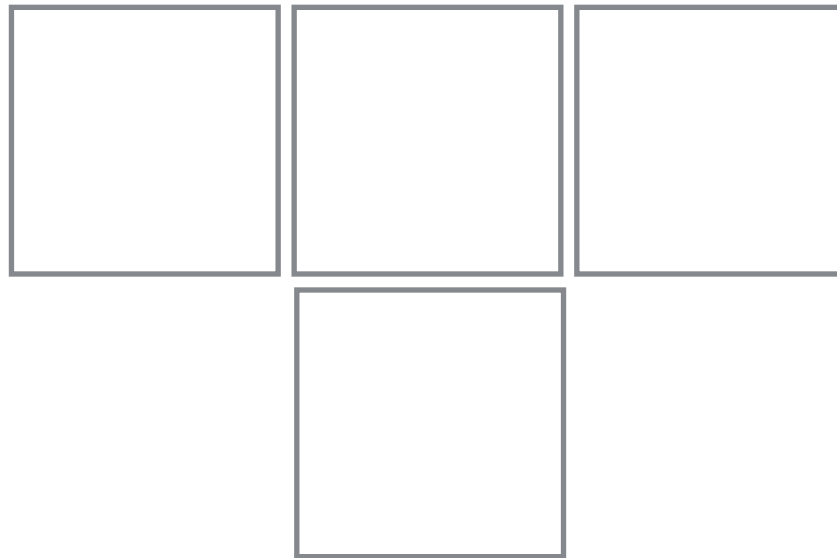
The Pieces



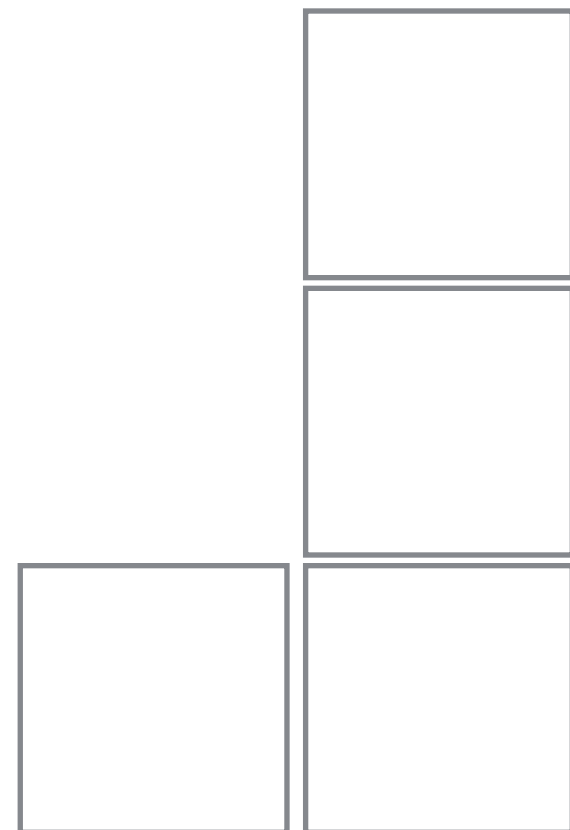
The Pieces



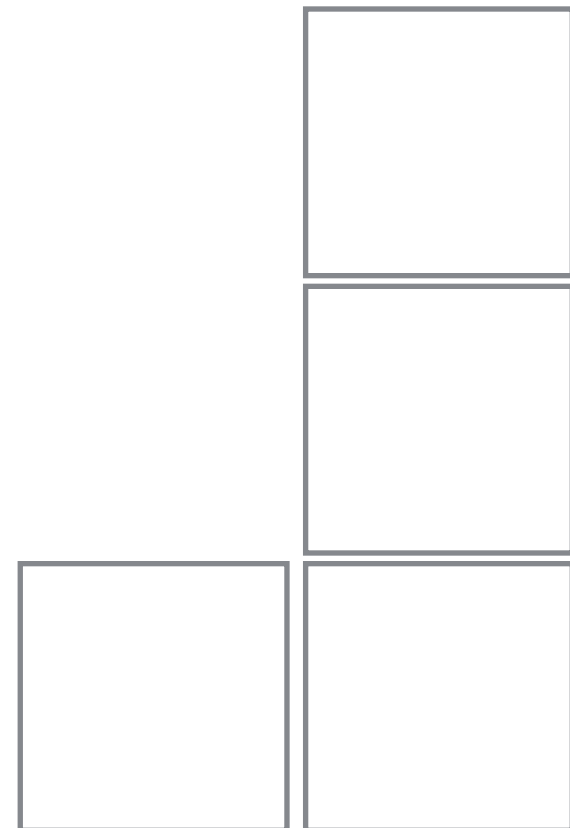
The Pieces



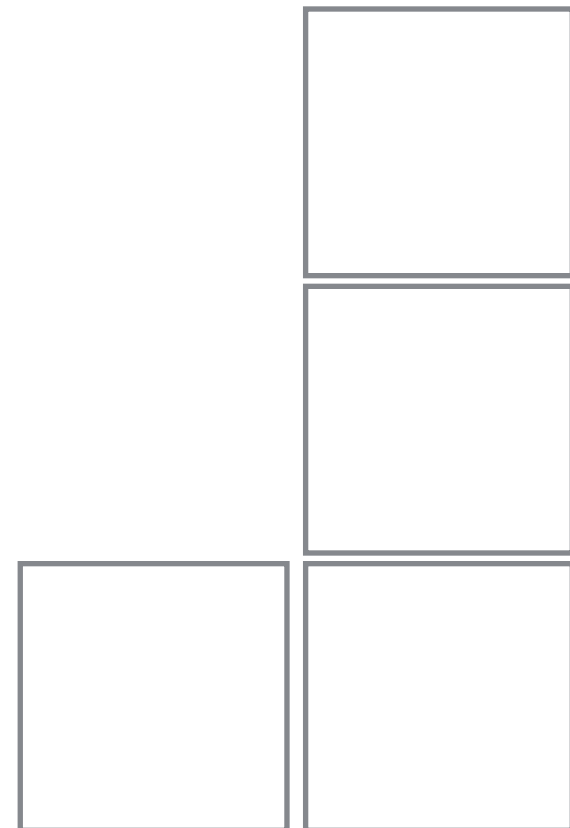
The Pieces



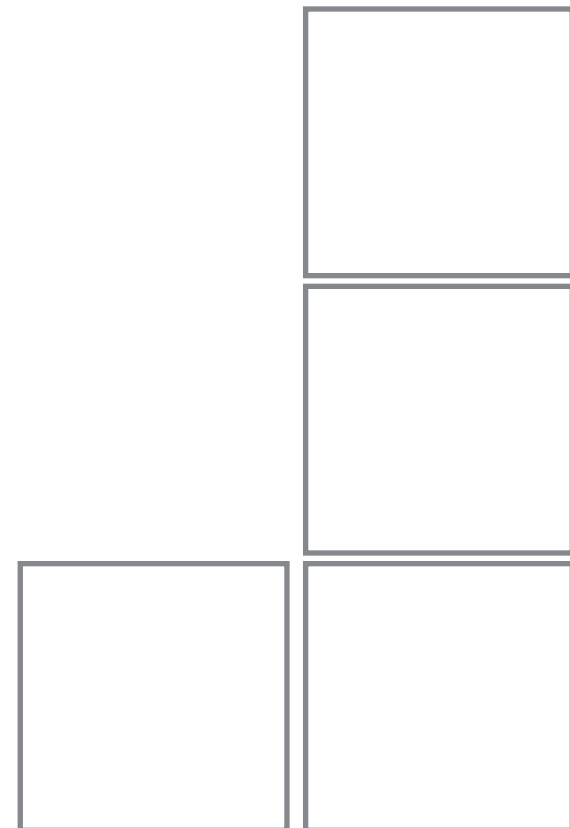
The Pieces



The Pieces



The Pieces



The Game Loop

- There are two event loops happening in Tetris

- The rendering of the game

- The movement of the pieces



- Can use setInterval to call a 'tick' function - I set the interval for 50 milliseconds



- Inside that 'tick' function, I call a function to render the grid as it exists at that moment



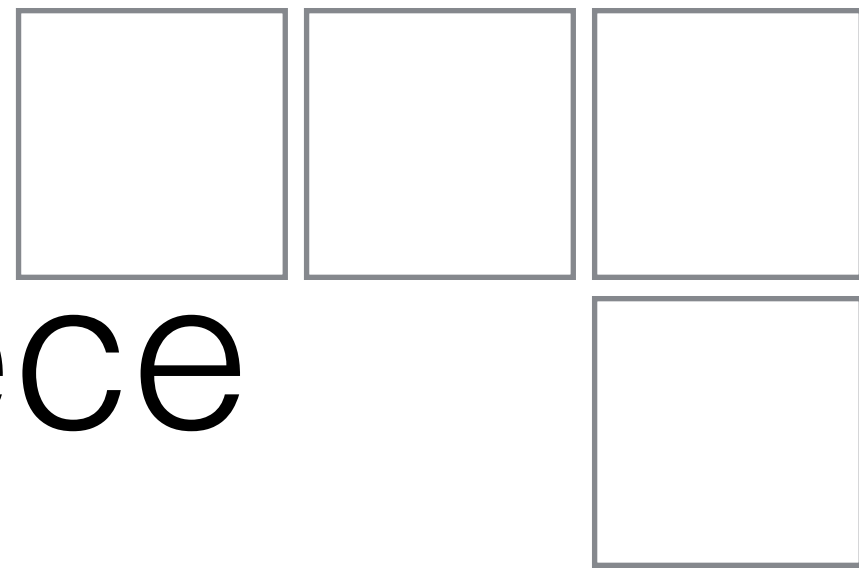
Ticking Along



- There are also two tick functions
 - The board tick
 - The piece tick



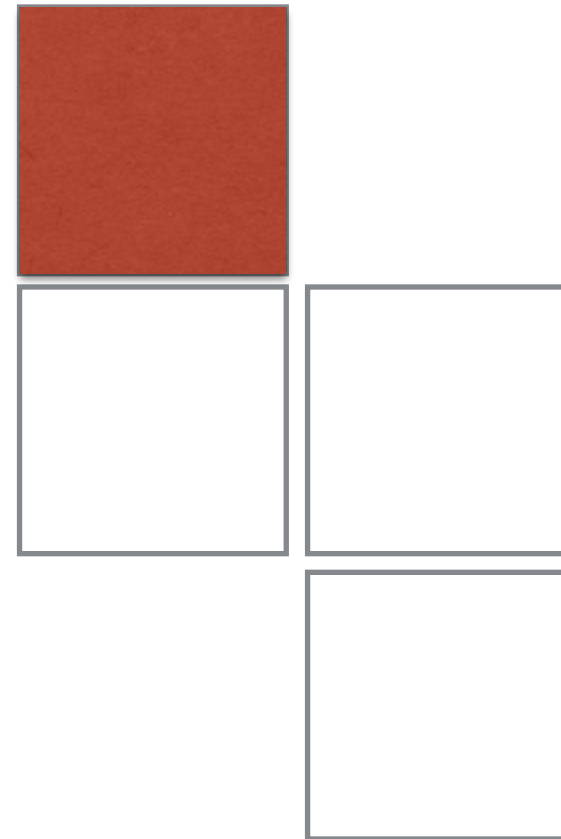
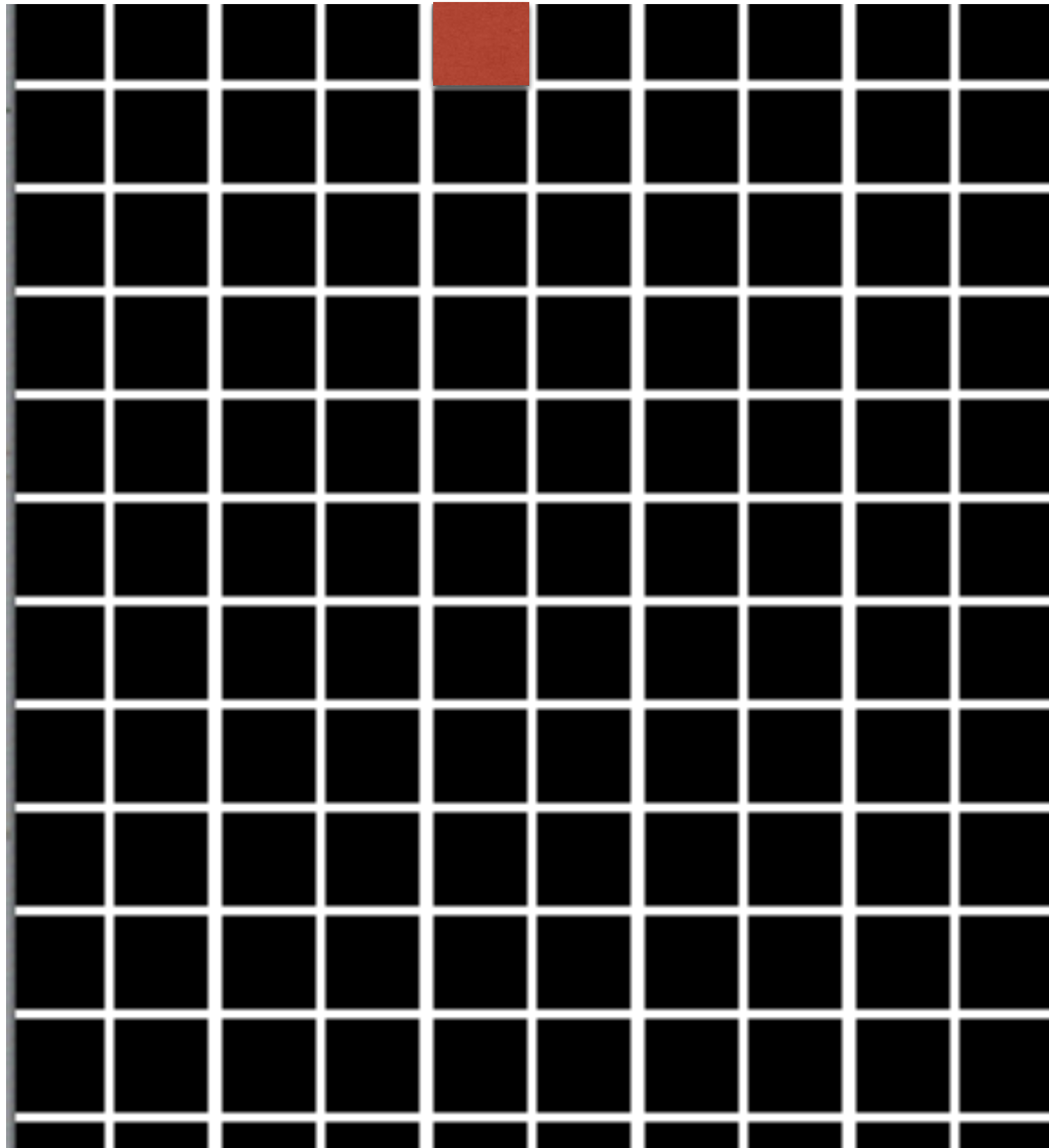
Adding a Piece



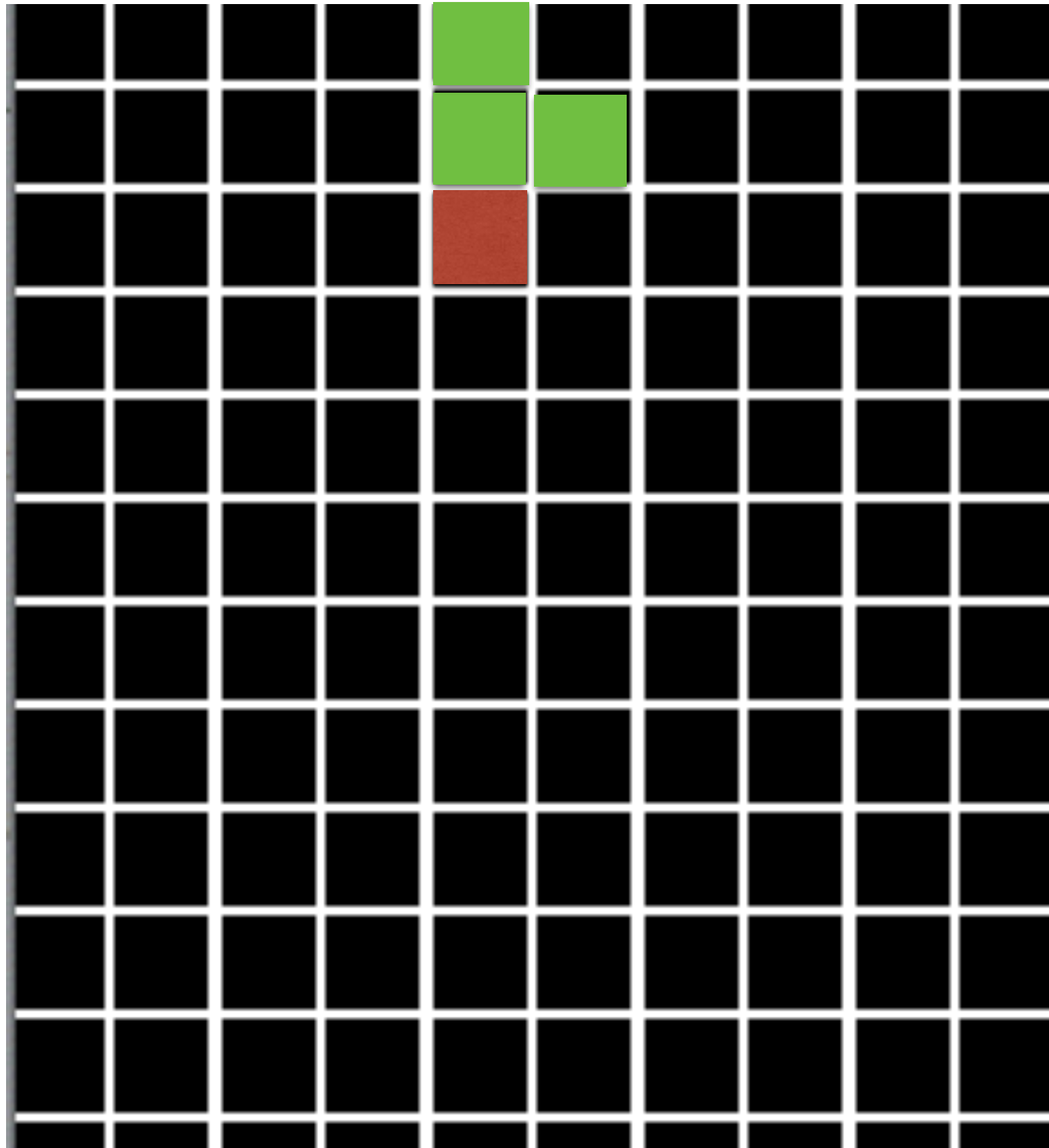
- The board checks to see if it already has an active piece, if not it generates a new Piece
- We look at the piece, and iterate over each row and column
- If that location is undefined, skip it. Otherwise, find the corresponding location on the board.



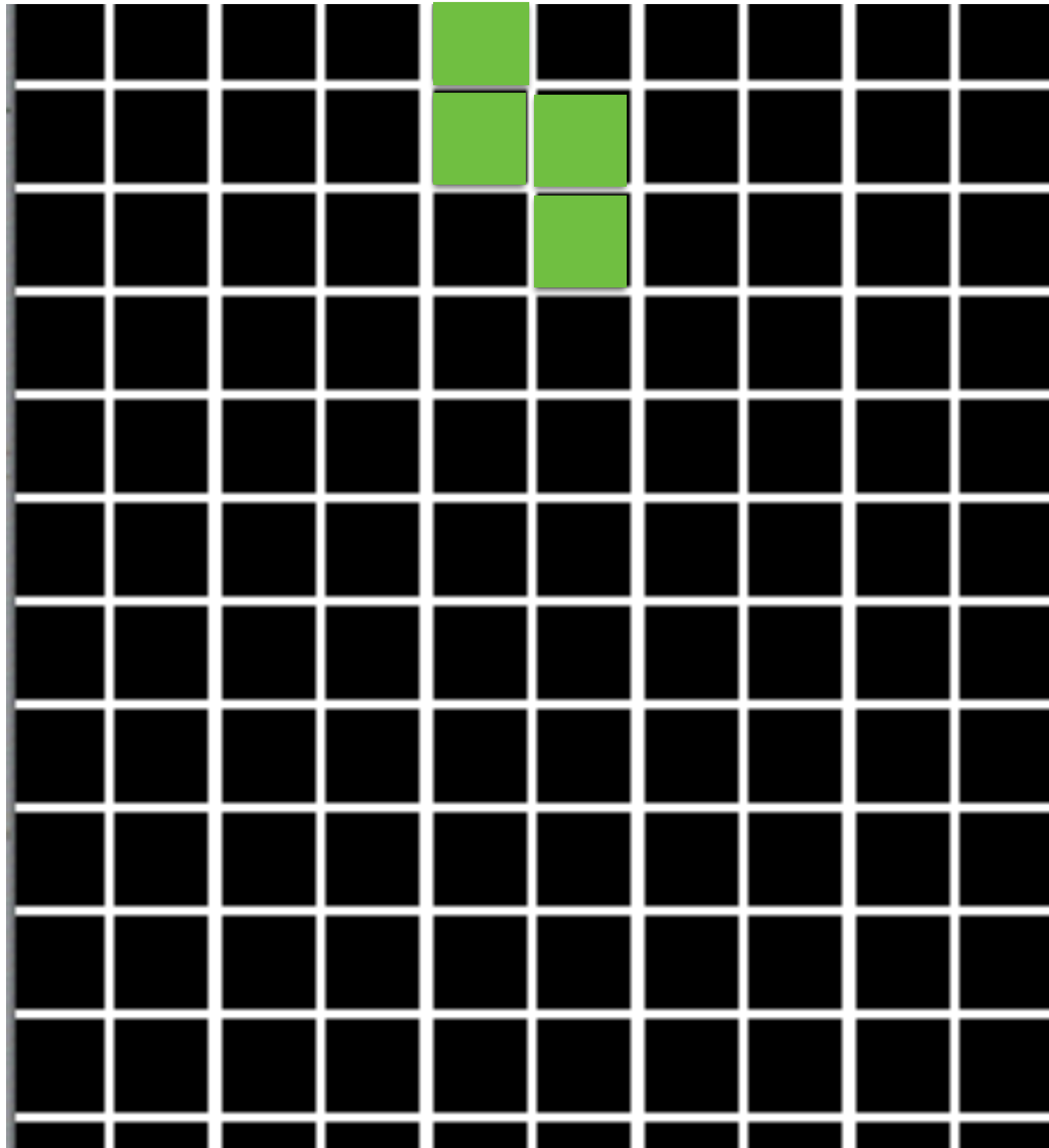
Adding a Piece



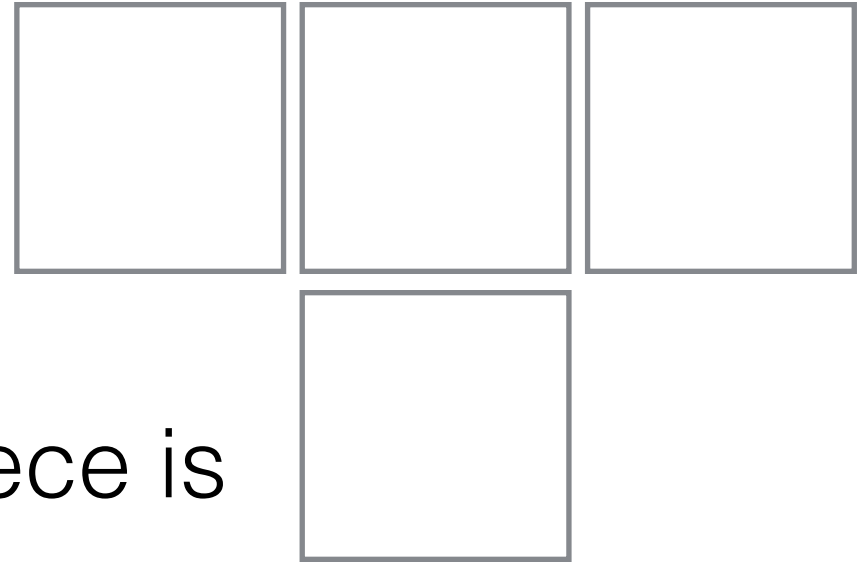
Adding a Piece



Adding a Piece



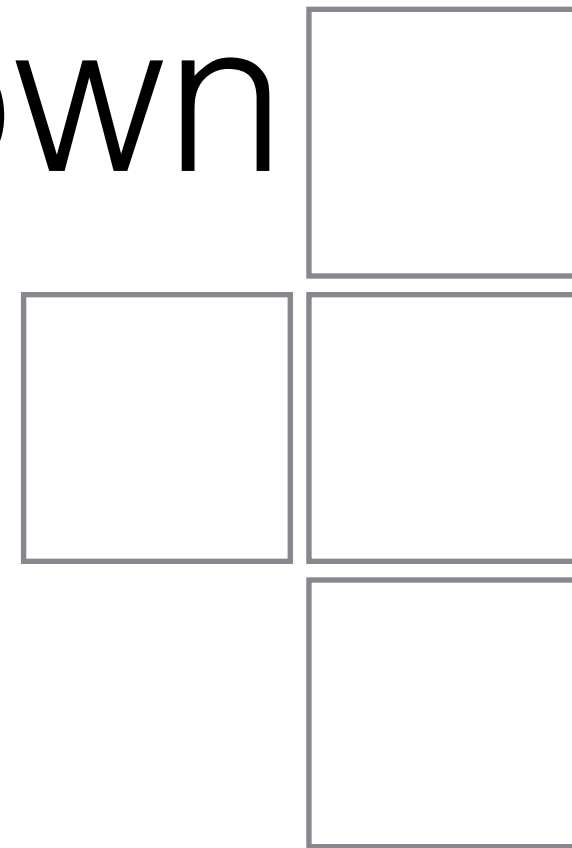
Removing a Piece



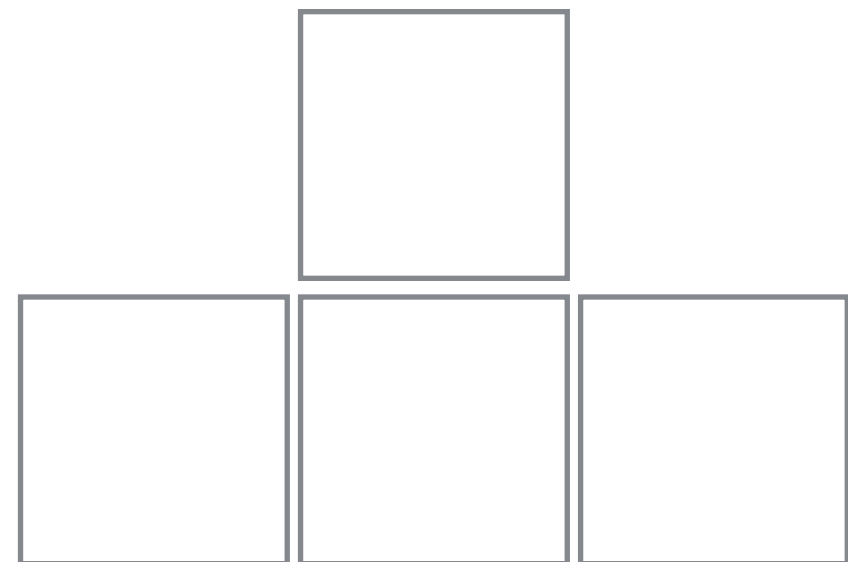
- It doesn't matter where the piece is
- Iterate over the entire board
- Remove any tile with a value of true



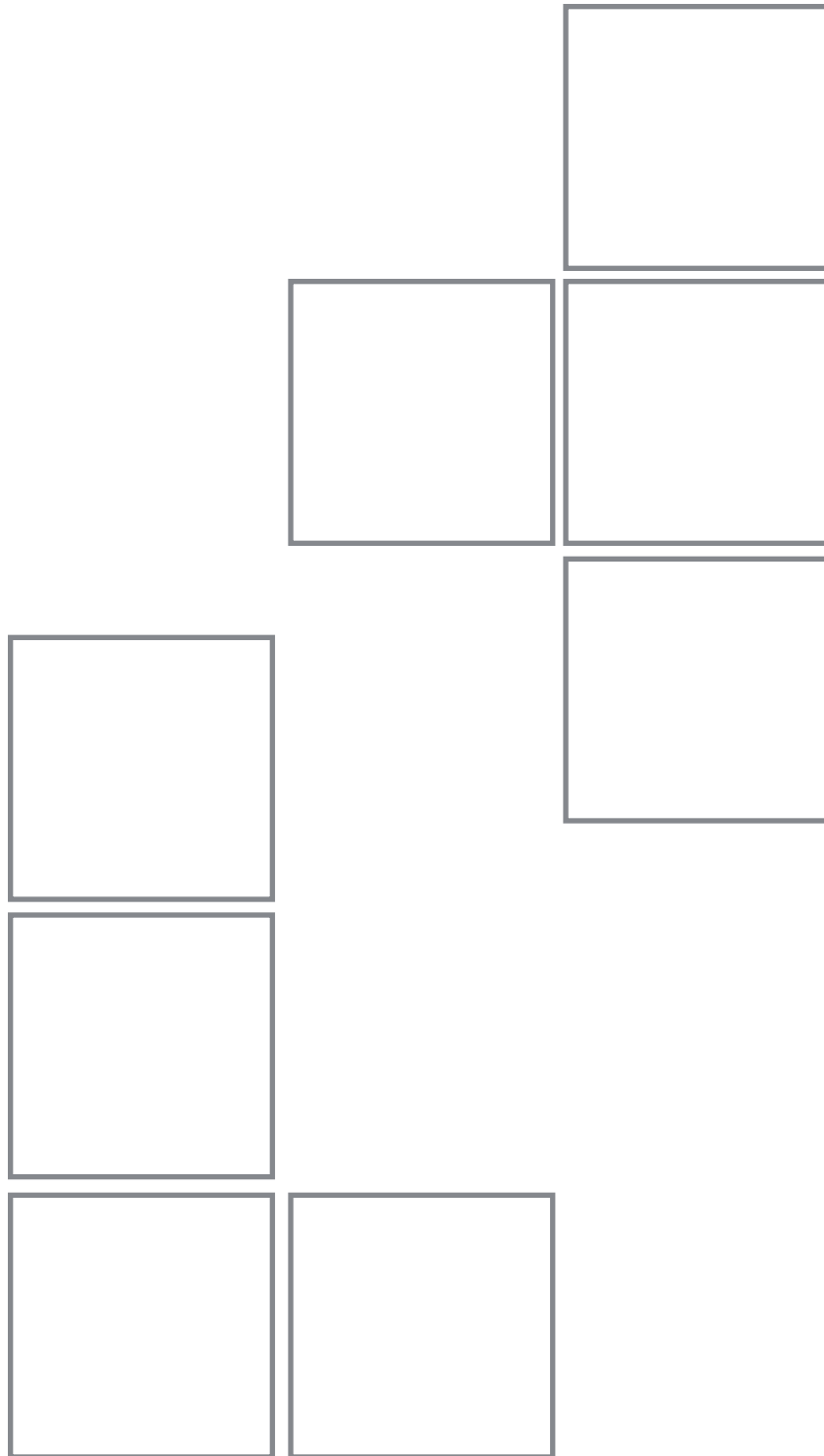
Moving a Piece Down



- Our piece knows its coordinates
- We just need to change the coordinates and re-render

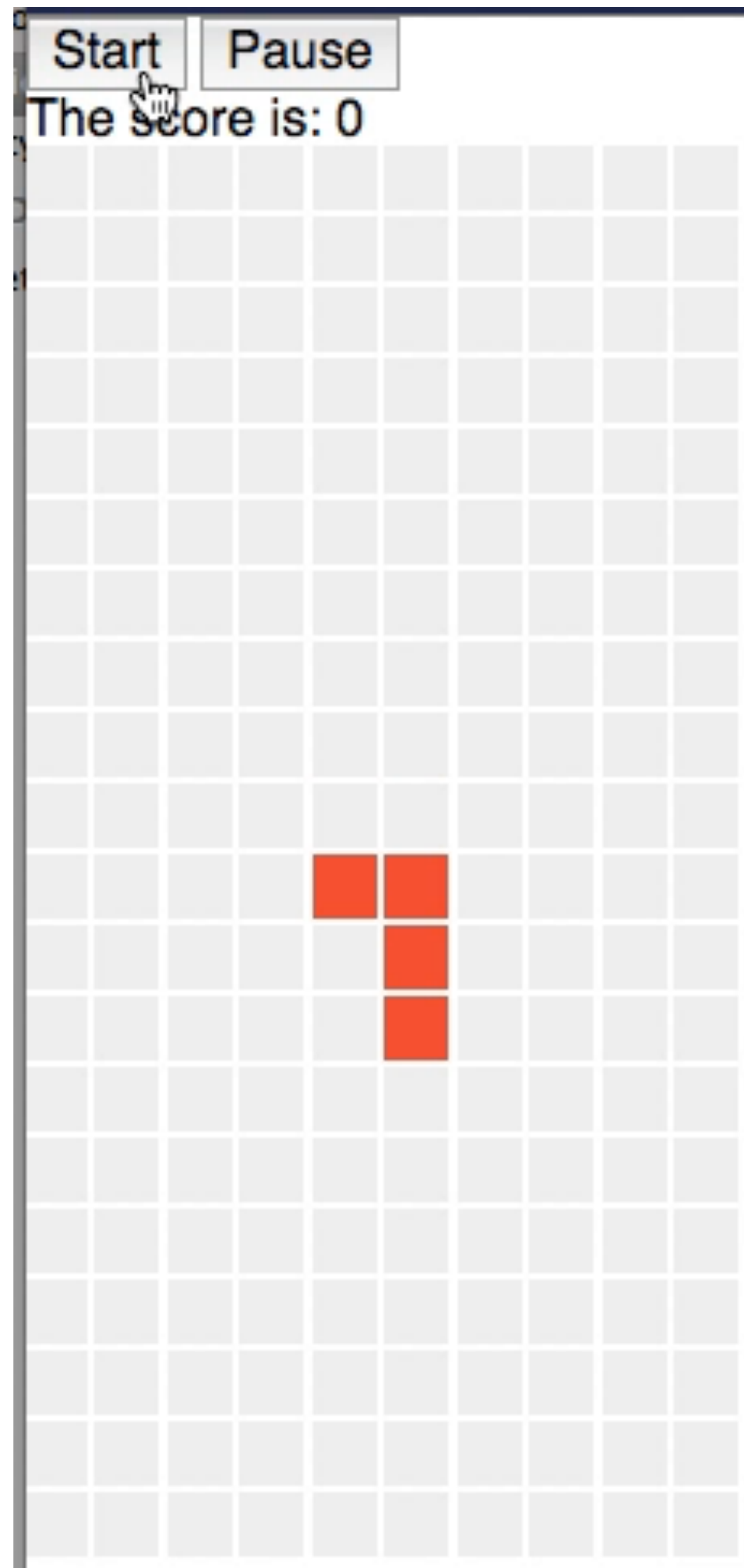
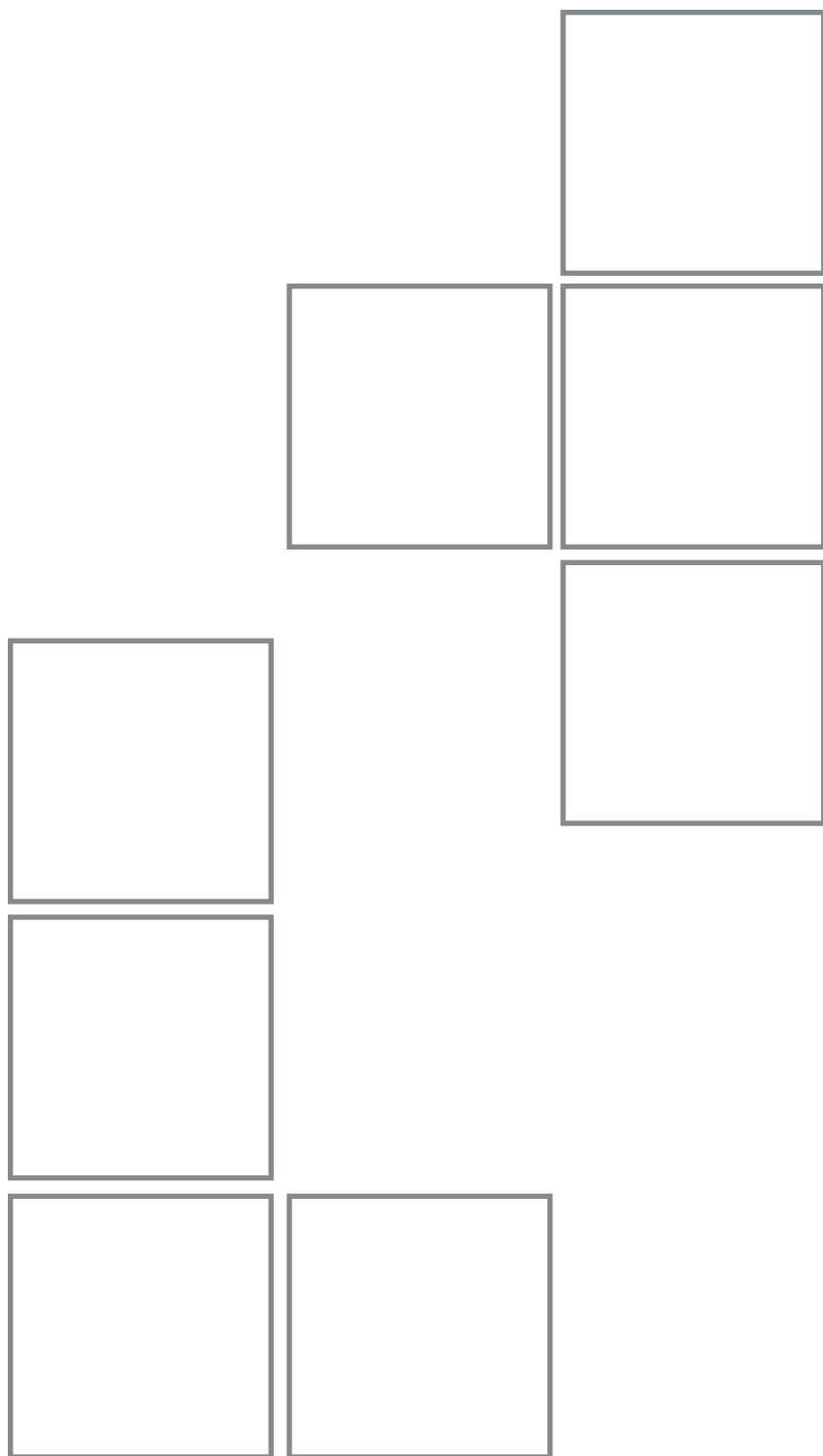


What does that leave us with?



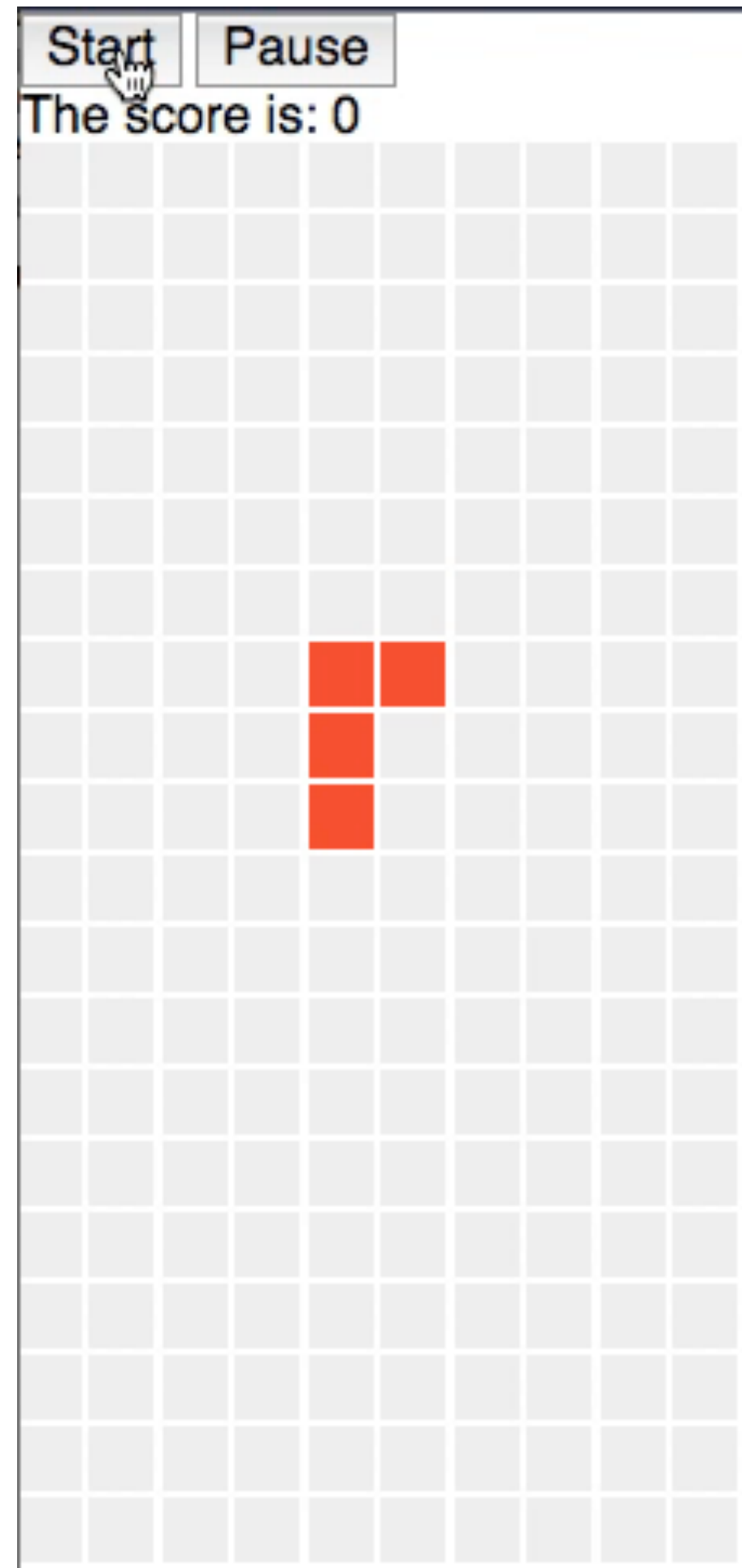
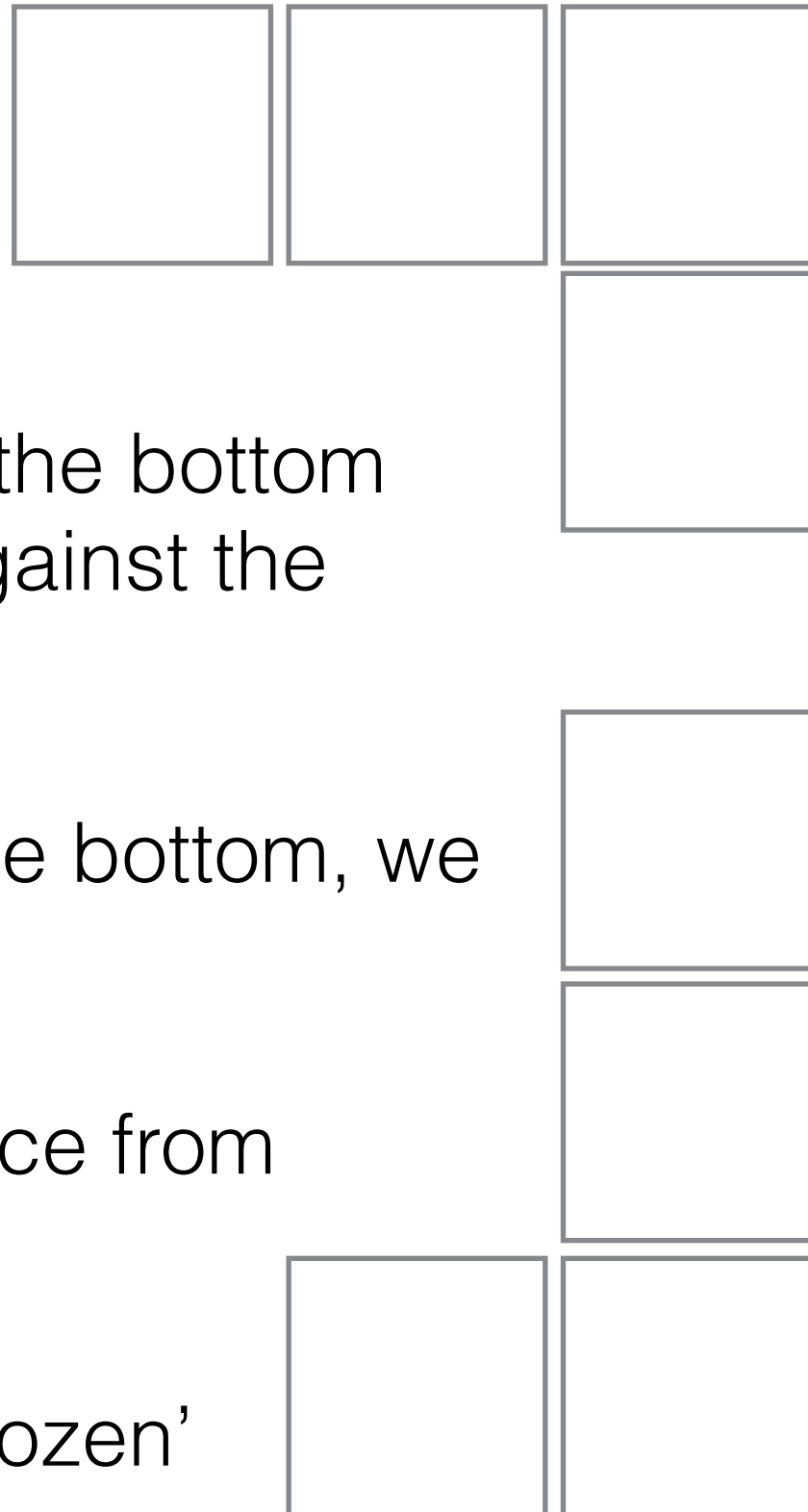
- Each time the board 'ticks':
 - It can add an active piece
 - It tells the active piece to tick
 - It re-renders the board
- Each time the piece 'ticks':
 - It moves coordinates down by one
 - It removes itself
 - It adds itself at the new coordinates

What does that leave us with?



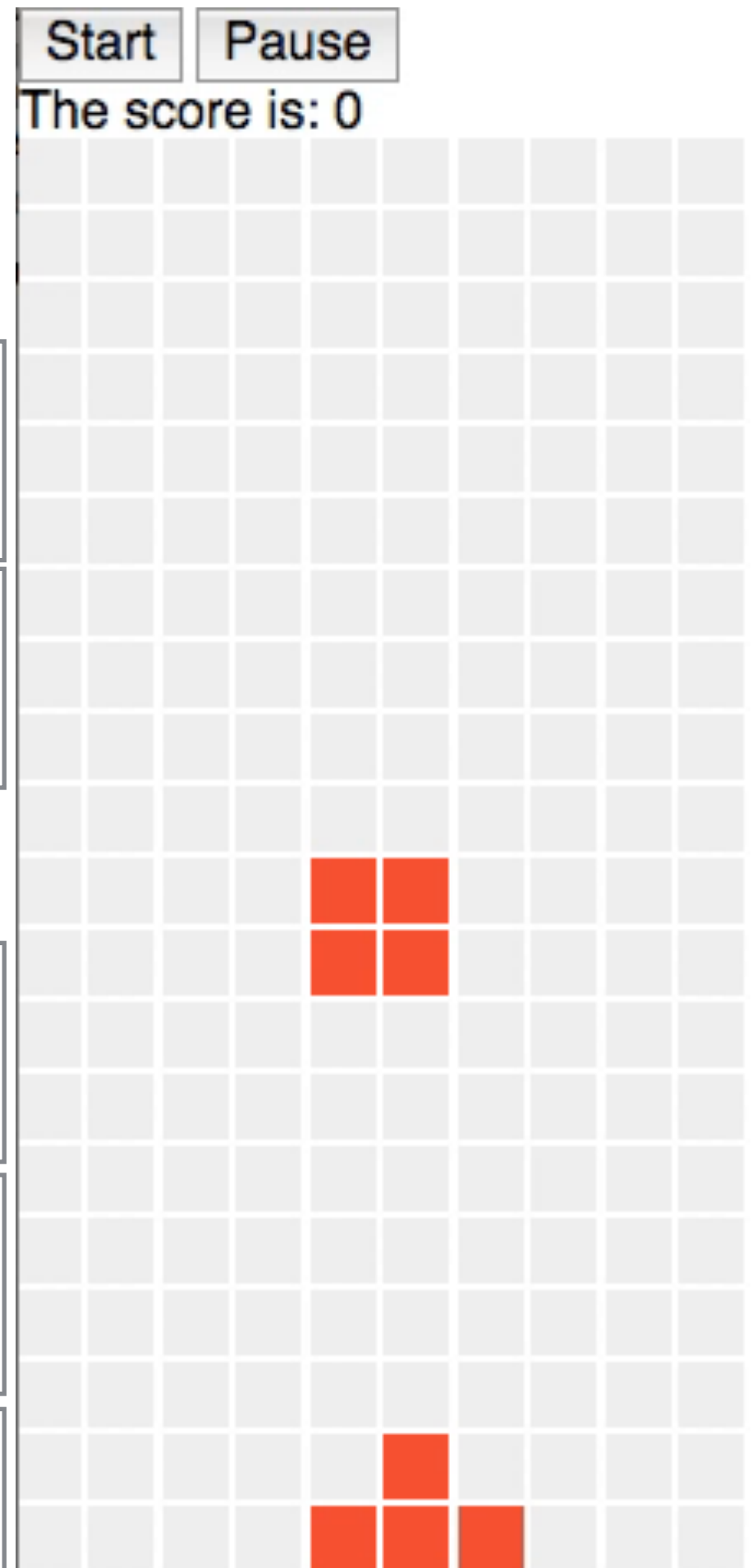
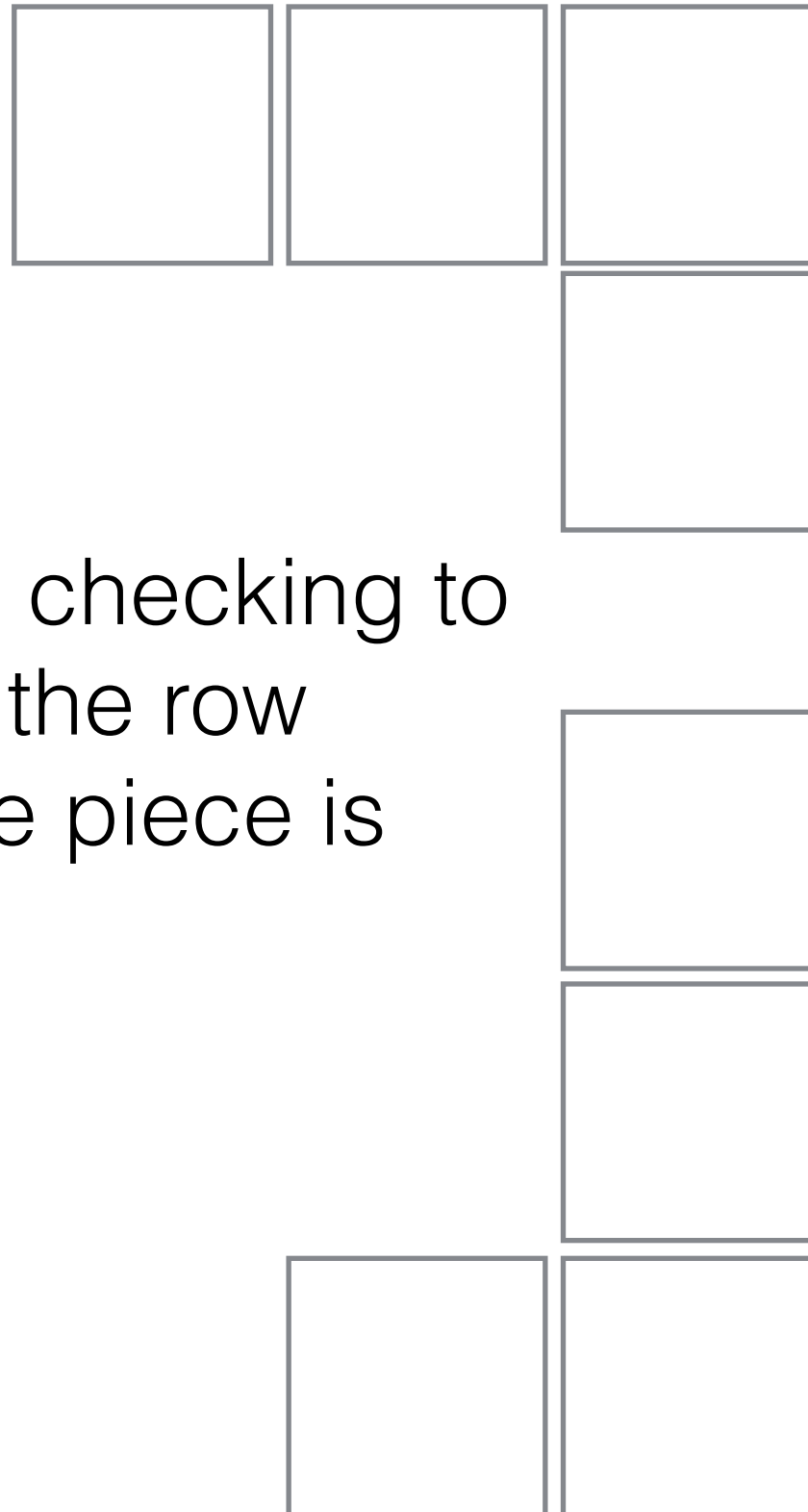
Stacking Pieces

- We need to check and see if a piece can move down
- Only need to check the bottom row and compare against the height of the board
- If a piece reaches the bottom, we 'freeze' it.
 - Remove activePiece from board
 - Set the value to 'frozen'



Stacking Pieces

- Evaluate piece, checking to see if the tile in the row below the active piece is frozen

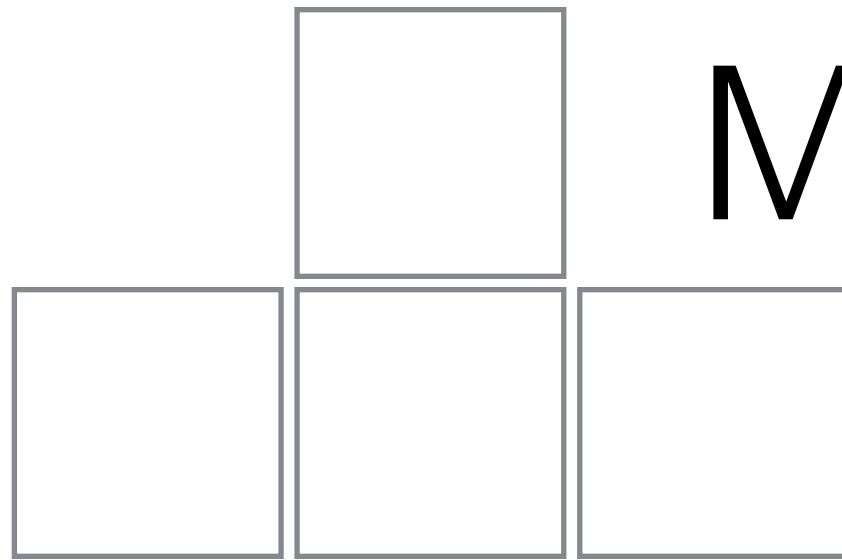




Keyboard Input

- Move left or right or down really fast or rotate on keydown
- But, instead of moving right away, we're going to use the key press to set the game state
- ```
board.input = {
 right: false,
 left: false,
 down: false,
 up: false
};
```





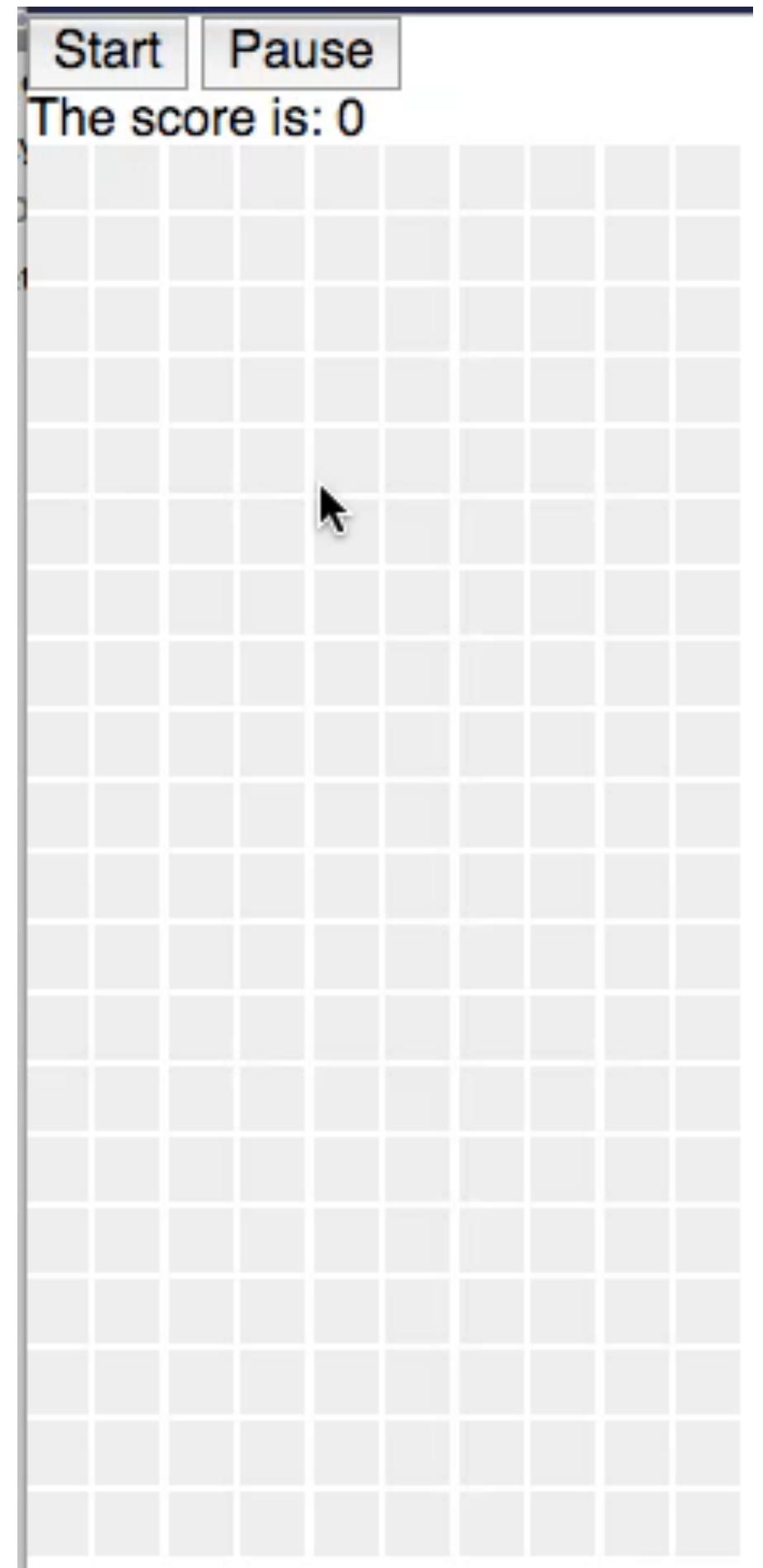
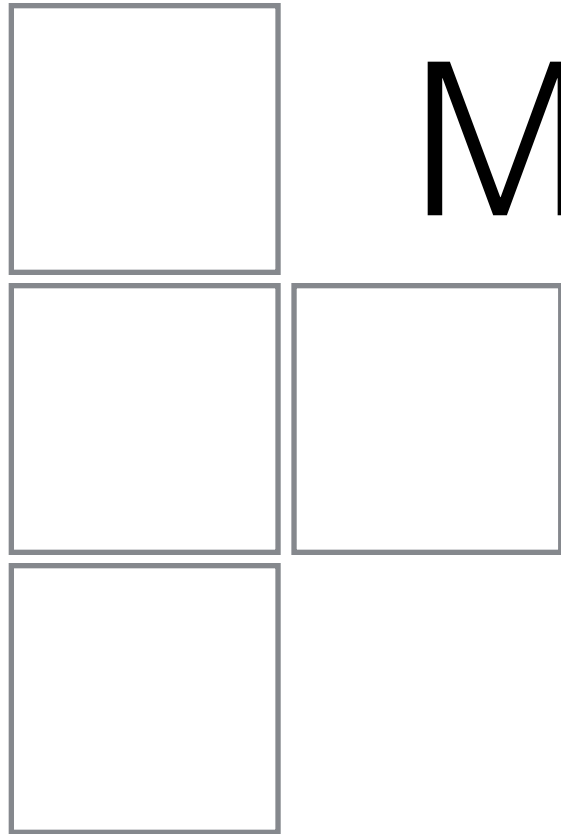
# Moving

- Moving from keyboard input is based on the speed of the 'tick', not the speed at which the piece moves down
- Calculating moving right and left are similar to moving down
- Move down on every tick while `board.input.down` is true



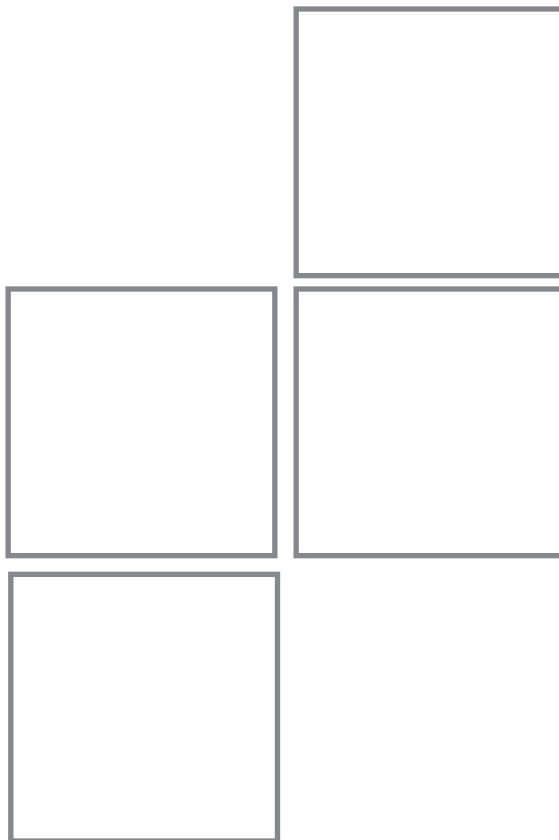


# Moving



# Rotating

- Get input same as other directional keys - state stored in `board.input`
- Each piece has an array of all orientations



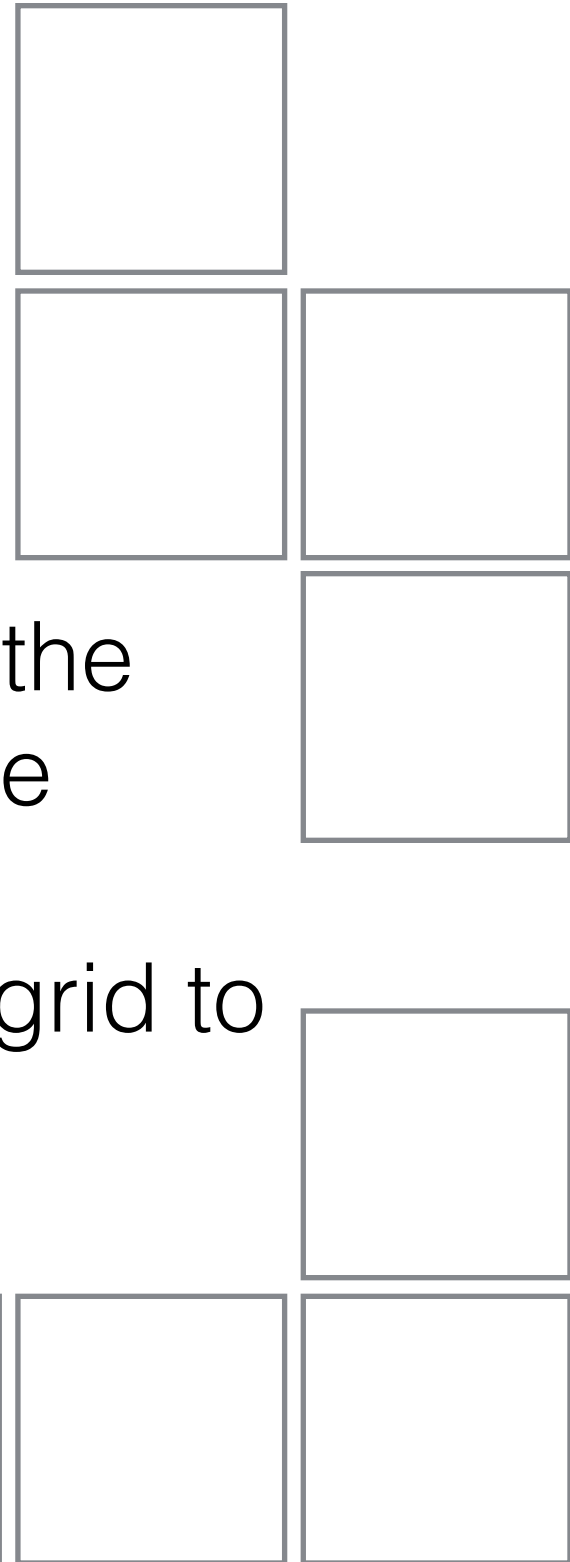
# Rotating

- Get input same as other directional keys - state stored in `board.input`
- Each piece has an array of all orientations
- We just need to change the current orientation to the next in the array, and loop back around



# Can you even rotate?

- Need to validate your rotations
- Check for a 'wall jump' - if piece will go off the board, shift the coordinates to keep it inside
- Compare the next orientation to the board grid to see if a frozen piece is already there

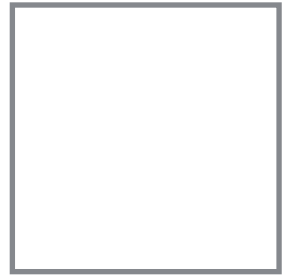


# Can you even rotate?



- Need to validate your rotations
- Check for a 'wall jump' - if piece will go off the board, shift the coordinates to keep it inside
- Compare the next orientation to the board grid to see if a frozen piece is already there

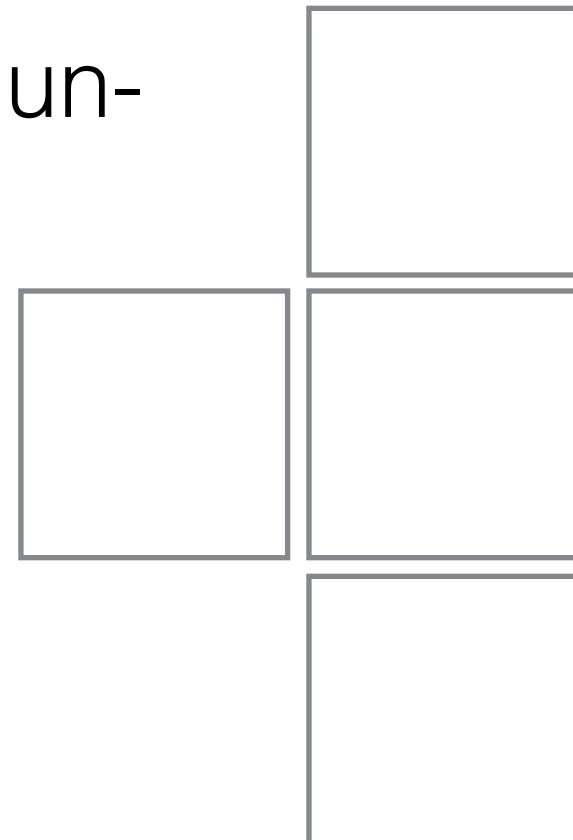




# Winning!



- Delegate this back to the board
- Iterate over all the rows and columns
- If you get through a row without finding an unfrozen tile, that row is removed
- A new row is added to the top of the grid

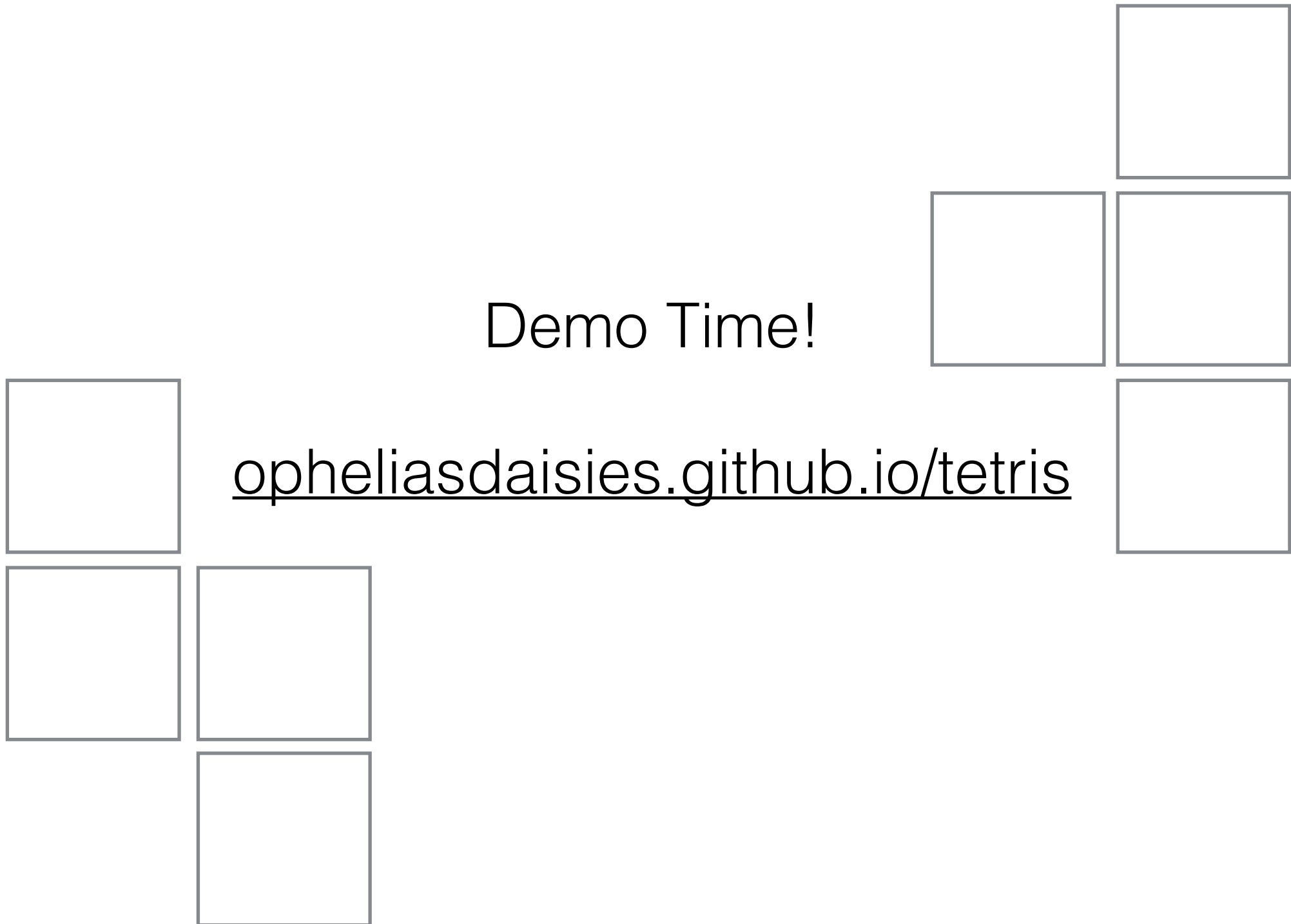




# Let's Play!

Demo Time!

[opheliasdaisies.github.io/tetris](https://opheliasdaisies.github.io/tetris)



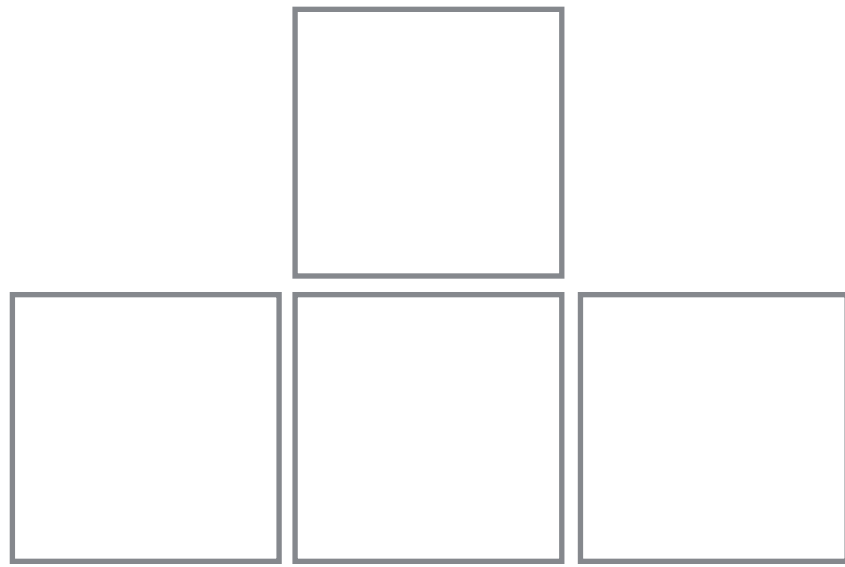
# Next Steps

- Add color
- Add losing



- Add score and row count
- Add levels with different speeds
- Add piece preview

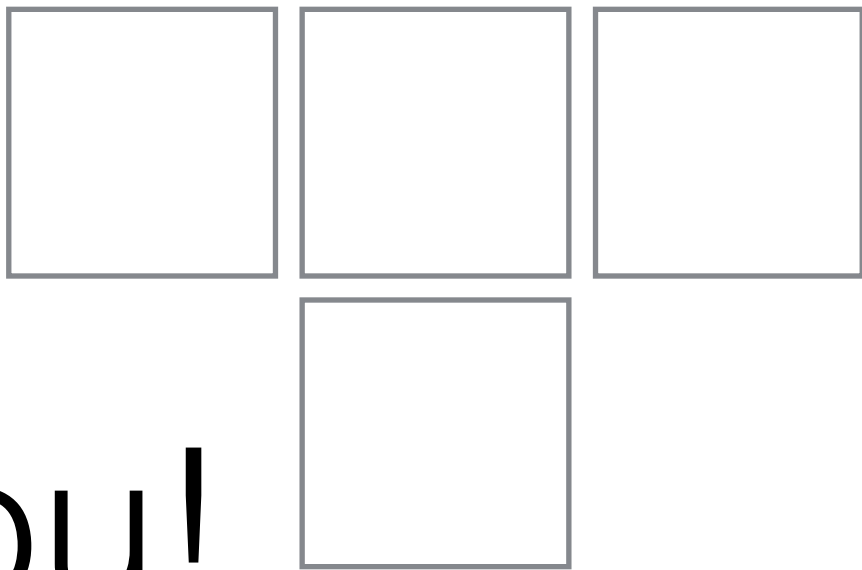




Go make games!







Thank You!

Sara Gorecki  
@opheliasdaisies

[spider-solitaire.herokuapp.com](https://spider-solitaire.herokuapp.com)  
[opheliasdaisies.github.io/tetris](https://opheliasdaisies.github.io/tetris)

