2a. I created my project using the Racket programming language and the DrRacket integrated development environment. My project is a visual memory game. The user is presented with certain randomly selected tiles that are white and others that are blue. Within a second all the white tiles will turn blue. The players job is to remember the white tiles and click the correct ones that are then flipped over. The game starts out as a 3x3 grid but gets bigger as the player successfully completes a level. To complete a level on the 3x3 grid the player must successfully complete 2 iterations. On the 4x4, 3 iterations and so on. The game levels go up to a 6x6 grid. First, only 3 tiles will be white but every time you complete a level there will be another white tile added to the grid. The greatest number of white tiles is 16 with a 6x6 grid. If you guess incorrectly 3 times within one of the grids, you lose a life. If you lose a life, you get demoted a level. You have 3 lives and once you run out the game is over. The high score and what score you got is shown.

2b. The entire development for this project was my own independent work. One part of the process was the implementation of the rules of the game and the second part was the graphical representation of the game state in response to the user's actions. There was an error when I was trying to get the tiles to appear beyond a 3x3. I wasn't sure why only the 3x3 was working because if that was working realistically the others should have worked. I reviewed my code and discovered that when I was defining the 4x4 etc. I forgot to add an ending part to the define which is why the 3x3 worked but the others didn't. After that I had the opportunity to develop the clicker function that lets the user click the tiles to make the 3x3 go to a 4x4 and so on.

2c. The tock function is called from the racket interpreter for every on-tick event. The function implements the rules of the game. It first checks to see if the game is in the playing state. If so, it determines whether the user has remaining lives to keep going. If they do, the game board with the white tiles is displayed. The function ready-to-close-open-tiles determines whether when the white tiles should be blued. The advance-to-next function and subsequent implements an algorithm to determine the correct number and position of tiles selected by the user. The next-playing function updates the PlayingStruct to go one level higher by checking the currently opened tiles and grid size. The reverse-to-previous function determines whether the incorrect number or position of tiles were selected. The previous-playing function also updates the PlayingStruct but to go back one level lower by checking the currently opened tiles and grid size. The user has only 30 seconds to make their selection. This is implemented in the idle-3-seconds function.

```
; tock : WorldState -> WorldState
; g : either ready/playing/end struct
; updates the playing struct if there are lives remaining,
; otherwise, update the end struct
(define (tock g)
       (if (playing? g)
       ; lives is greater than 0
       (if (> (playing-lives g) 0)
                    ; time has expired, close the opened tiles
              (cond [(ready-to-close-opened-tiles? (playing-state g)(playing-time g))
                                                   (close-opened-playing g)]
                    ; advance to the next level
                    [(advance-to-next? (playing-grid g) (playing-tiles-to-open g))
                                       (next-playing g (playing-high-score g))]
                    ; go back down to the previous level
                    [(reverse-to-previous? (playing-grid g)) (previous-playing g)]
                    ; no activities for 30 seconds end the game
                    [(idle-30-seconds? (playing-state g) (playing-time g))
                    (make-end (playing-score g) (playing-high-score g))] ; end game
                    [else g])
       ; no more lives, end the game
       (make-end (playing-score g) (playing-high-score g)))
       g))
```

2d. The PlayingStruct data structure is used by a number of functions to maintain the state of the game. The functions that implement the rules of the games write into the structure and the drawing functions use it to determine which tiles need to be drawn. This abstraction of game state allows the rule checking/setting function to work in coordination with the drawing functions.

```
(define-struct playing [lives score high-score state side-length rows columns tiles-
to-open time grid])
```