Bomberman Game Discussion

requirement in the instruction.pdf

1.An overview of the project topic and of the group members, mentioning the main tasks that each of them primarily worked on during project development

2.A brief demonstration of what your project does (that is, you run the project to show how it's used: bring your laptop)

3.The main data types used in the project's implementation

4.How the main project functionalities of the project are implemented and by what functions

5.What kinds of tests you wrote and which functions you tested more thoroughly

6.Whether the project went through several revisions or was designed develop- ing the same idea from the start

part1:main tasks that each of them primarily worked on during project development

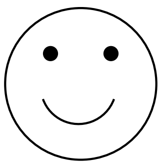
Elena Hu worked for the public,render,on-key,test2(render and key-handler test)

Chengyu Yu worked for the public,on-tick,stop-when,test(stop-when,public,on-tick test)

My friend designed the decorations part for us(If TAS ask this question)

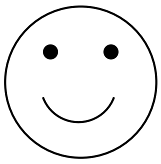
part2

Bring computer and play game for them



part4

See them in code and also prepare some questions for each other and talk about them tomorrow



part3 and part 6:Gamestate Field Modification

compare the gamestate structure

Previous gamestate:

```
(define-struct gamestate [layout bomb player1 player2 roundtimer maximum quit?])
layout is a vector of vector of symbol but just have 1-length legal symbol
bomb is a list of bombstate,and bombstate is:(define-struct bombstate [cor countdown])
each player: (define-struct player1 [posn dead? bombcount])
```

Milestone gamestate:

```
(define-struct gamestate [layout bomb player1 player2 roundtimer maximum quit? boom-cor])
Change compared to Previous gamestate:
Add a boom-cor field
Add three-length legal symbol in the layout
Modify each player:(define-struct player1[cor direction])
Bombstate:(define-struct bombstate [cor countdown owner])
```

Submit gamestate:

```
(define-struct gamestate [layout bomb player1 player2 roundtimer maximum quit?])
Change compared to Milestone gamestate:
Delete boom-cor field
Add two-length legal symbol 'E1 'E0 in the layout
```

Why we add direction field of each player?

Since we wanna to make sure player can show different image when they move.

Why we use cor field to replace the posn field of each player?

At first , we wanna use Posn(floating numbers) to represents the location of each player and use Cor(Integer) to represents each cell.

So we design a simple convert function:

```
(define (nearest-cor posn)
  (make-cor
   (floor (/ (posn-x posn) CELL-SIZE))
   (floor (/ (posn-y posn) CELL-SIZE))))
```

Obviously convert Posn to Cor we need use floor function to convert the floating Number to Integer, so it caused a lot of problems such as:

1.Move-predicator and put-bomb-predicator

2.Check-die functions

At that time we have two choice:

1. Giving up the first idea of Cor (it simplifies a lot of logic in our game) and use Posn to represents all the locations in this game

2. Use Cor to represent the location of player

At last, we choose the second choice.after that ,the keyhandler part and stop-when part are very easy to achieve.

The related video is named as Posn in backup.

Why we delete died? field of each player field

The reason for designing died? field of each player is serving for the stop-when part of this game before we change the Posn field to Cor of each player.

When the nearest-cor function's ouput Cor is in the range of 'E1 or 'E0, the player will die.

However, since the change of Posn to Cor, we update the symbol system.So that the died player will be represented directly in the layout such as:

–'E1R(E+who+its direction)

So that we just need to use get-symbol function to get the symbol of each player's Cor in the layout to know their state.

Why we Add a boom-cor field of the gamestate in the milestone but delete it in the final submit?

This reason is related to a issue in the timehandler part, and it is also the reason of changing the range of countdown of each Bombstate Structure.

```
;countdown:
;represents the countdown of the each bombstate
;5 is the time for bomb just added to game
;2 is the time for bomb start to explode
;0 is the time for bomb end explode

(define (timehandler gamestate)
  (let*
      (
      [updated-gamestate
       (make-gamestate (gamestate-layout gamestate)
                       (updated-bomb (gamestate-bomb gamestate))
                       (gamestate-player1 gamestate)
                       (gamestate-player2 gamestate)
                       (updated-roundtimer (gamestate-roundtimer
                          gamestate))
                       (if (check-roundtimer? (gamestate-roundtimer
                          gamestate))
                                           (add-maximum
                                              (gamestate-maximum
                                              gamestate))
                                           (gamestate-maximum
                                              gamestate)))]

               ;;;;;;;This part is responsible for updating all the
                  countdown(roundtimer,whether add maximum,countdown of
                  each Bombstate);;;;;;;;;;;

      [boom-end-state
       (if (check-zero? (gamestate-bomb updated-gamestate))
           (boom-end updated-gamestate)
           updated-gamestate)]

             ;;;;;;;;;;This part is responsible for handling the
                countdown=0 bomb (update layout from 'E->'W,remove
                exploded bomb)

      [bomb-boom-state
       (if (check-two? (gamestate-bomb boom-end-state))
           (boom boom-end-state)
           boom-end-state)]

           ;;;;;;;;;;This part is responsible for handling the
              countdown=2 bomb (convert the cor in boom-range in layout
              to 'E, kill player,similar to submit version)

      )
    bomb-boom-state))
```

In this version: We need to calculate the total boom-range in the boom-function, and it is the real boom-range. But boom-end function cannot get it,so we have to calculate it again in boom-end function to make sure which range needed to be converted to 'W.

However, this method make the code so long and complex.and it also cause a bug:

Since one of the boom-range rule is: the explosion of the bomb could dismental the tree('D) but it should stop extending in this direction.However,since the 'D has been converted in the boom function, So in the boom-end function when we recalculate the boom-range,it will regard it as 'E,so in fact ,it will still extend in this direction to make the next 'D to 'W.

(This can be seen in the video named as "Double calculating boom-range")

To solve this issue, my first idea is to use some trick so that boom-function can pass the boom-range to the boom-end-function, so we add a boom-cor field (a list of Cor) the gamestate so that boom-end-function can reuse the boom-range that boom-function calculated before.

But this bring another issue:

–1.Some 'E cannot disappear(This can be seen in the vedeo named as "Passing boom-range").

–2.In fact, in almost all the bomberman games, we wanna see that after bomb boom, the player's bombcount can be removed after the bomb-boom but not boom-end.

And inspired by the issue 2. (I guess the boom and boom-end is divided to handle in bomberman games in the markets).We decided to divide the logic of boom and boom-end(in fact its just the convert of E to W),so we rewrite the countdown of Bombstate setting and delete the boom-cor field.and add two functions to handle the convert of E to W in the layout.

```
;countdown:
;represents the countdown of the each bombstate
;3 is the time for bomb just added to game
;0 is the time for bomb start to boom

(define (update-E-symbols symbol)
  (cond
    [(equal? symbol 'E1) 'E0]
    [(equal? symbol 'E0) 'W]
    [else symbol]))

(define (updated-layout layout)
  (vector-map
  (lambda (row)
    (vector-map update-E-symbols row))
  layout))
```

So that, we deleted the boom-end function, and handle all the issue about boom in the boom-function.and after boom, we remove the countdown=0 Bombstate from the list.

Why we delete the bombcount field of each player and add owner field of bombstate?

This is not a very big issue. But remove-bomb function will remove the countdown=0 Bombstate from the list,so that if we use owner field in Bombstate,we don't need to access player field in the remove-bomb function.Its better than using bombcount field of each player.

Why we update symbol system(the symbol that is legal in layout)?

For adding three length symbols('E1R 'E2L) it is becasue of the change in Posn convert to Cor part.

For adding two length symbols('E1 'E0) it is because of the change in Why we Add a boom-cor field of the gamestate in the milestone but delete it in the final submit? part.

Why we choose Vector of Vector of Symbol not List of list of Symbol?

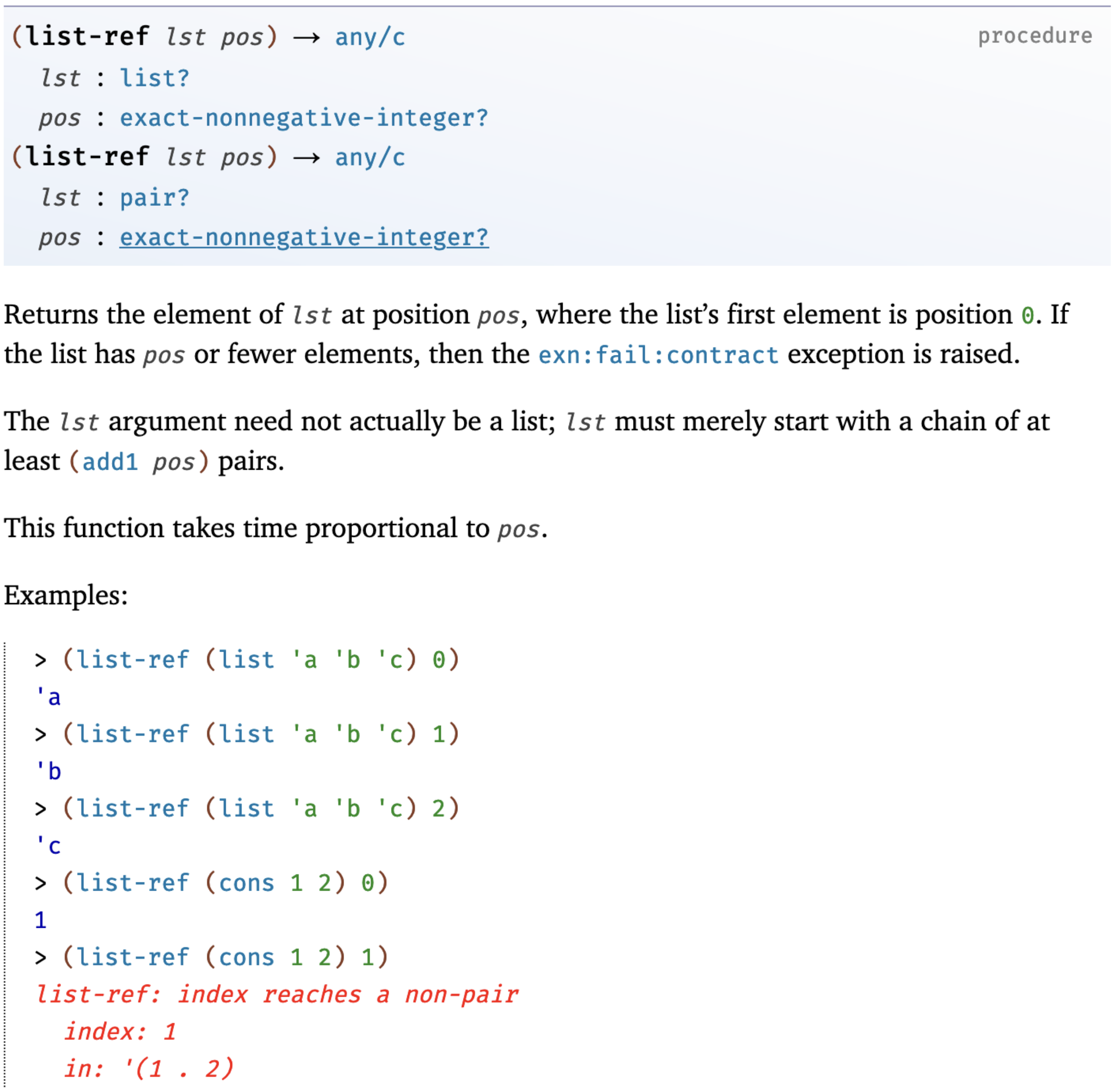
It is based on the two functions in the public.rkt

```
(define (get-symbol layout cor)
  (if (in-bound? cor layout)
  (vector-ref
   (vector-ref layout (cor-row cor))
   (cor-column cor))
  'ILEGAL))
(define (convert layout cor symbol)
  (cond
    [(empty? cor) layout]
    [(list? cor)
    (convert
     (convert layout (first cor) symbol)
     (rest cor)
    symbol)]
    [else
     (begin
      (vector-set!
       (vector-ref layout
                  (cor-row cor))
       (cor-column cor)
       symbol)
      layout)]))
```

reason1:time complexity

See the document of list-ref, we can know that: the time complexity of list-ref is O(n) since list is a linked list

so list-ref accesses an element by traversing the entire list from the beginning to the specified position



```
(list-ref lst pos) → any/c                                procedure
  lst : list?
  pos : exact-nonnegative-integer?
(list-ref lst pos) → any/c
  lst : pair?
  pos : exact-nonnegative-integer?
```

Returns the element of *lst* at position *pos*, where the list's first element is position `0`. If the list has *pos* or fewer elements, then the `exn:fail:contract` exception is raised.

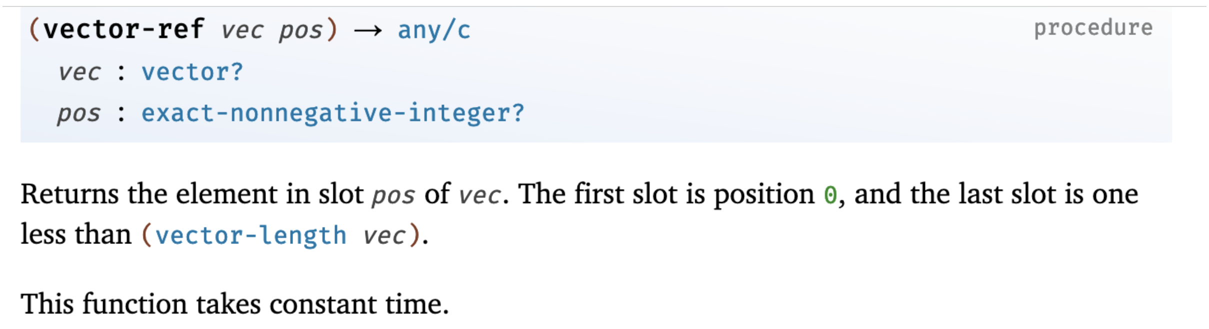
The *lst* argument need not actually be a list; *lst* must merely start with a chain of at least `(add1 pos)` pairs.

This function takes time proportional to *pos*.

Examples:

```
> (list-ref (list 'a 'b 'c) 0)
'a
> (list-ref (list 'a 'b 'c) 1)
'b
> (list-ref (list 'a 'b 'c) 2)
'c
> (list-ref (cons 1 2) 0)
1
> (list-ref (cons 1 2) 1)
list-ref: index reaches a non-pair
  index: 1
  in: '(1 . 2)
```

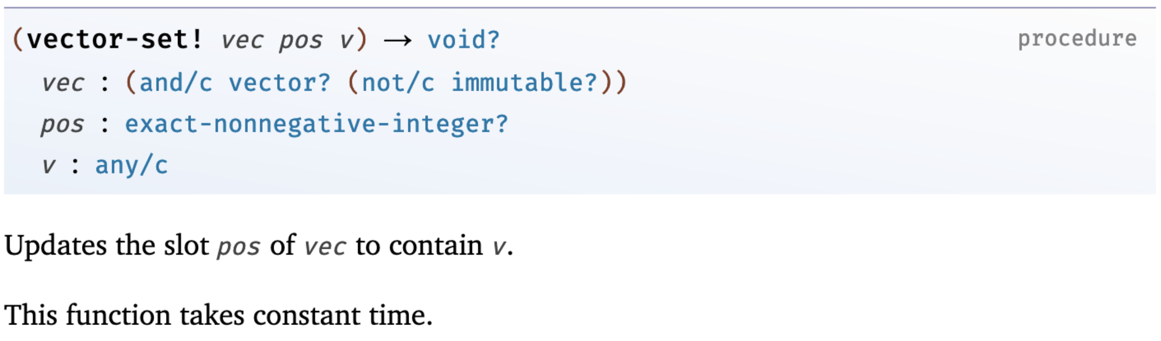
See the document of vector-ref and vector-set! we can know that : the time complexity of vector is O(1) since vector is contiguous storage



```
(vector-ref vec pos) → any/c                                    procedure
  vec : vector?
  pos : exact-nonnegative-integer?
```

Returns the element in slot *pos* of *vec*. The first slot is position `0`, and the last slot is one less than `(vector-length vec)`.

This function takes constant time.



```
(vector-set! vec pos v) → void?                                 procedure
  vec : (and/c vector? (not/c immutable?))
  pos : exact-nonnegative-integer?
  v : any/c
```

Updates the slot *pos* of *vec* to contain *v*.

This function takes constant time.

reason2: list is immutable data structure

Since a list is an immutable data structure, using convert to modify a list would require recursively traversing the entire list. At the specified position, the function would need to replace the value while reconstructing the list. Additionally, convert must handle both single coordinates (cor) and lists of coordinates (boom-range).

But if we use vector, its very easy. we just need to use vector-ref to achieve this functionality.

reason3: Vector is a fixed size data structure

For the layout,it has a fixed columns and rows.So we just need a fixed size data structure,we don't need to add/delete symbol in the Vector,but just change them.For list,it is more convenient to handle the un-fixed-size data because like the list of bombstate since we need to remove / add bombstate in the bomb-list.

part5:test part

I need to make dinner, talk about it tomoroow.