TLE ELIMINATION

Hui Zhi, Hansong Chen, Xingkai Zheng, Xuechun Li

PROJECT OVERVIEW

1. Game Formation : Basic Idea and Rules
2. Project Architecture : UI and Back-End Logic
3. Implementation: Libraries, Functions and Challenge
4. Test of Back-End Logic: quickCheck
5. Demo

Game Formation

Basic Idea:

a simple elimination game

Rules:

· three same blocks -> elimination

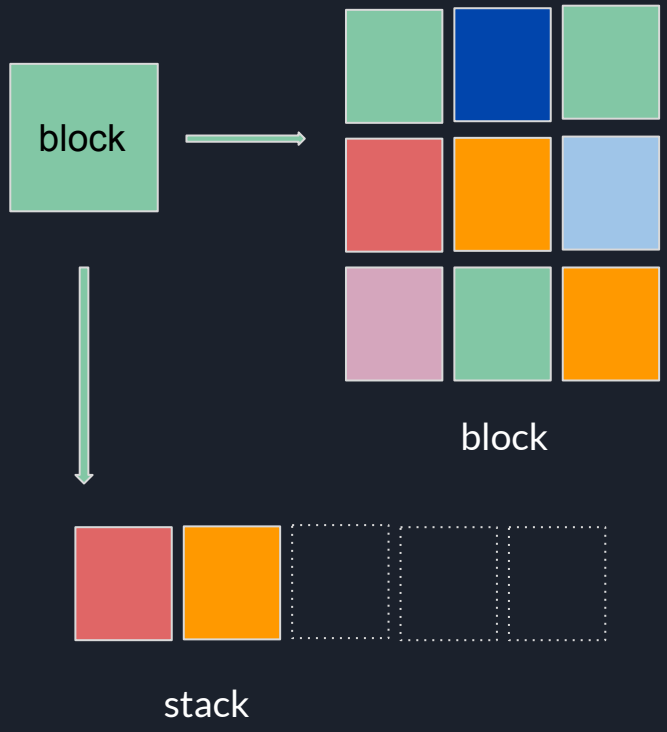
· exceed stack limit -> fail

· use helper tools: empty stack/remove/...

cr:Internet

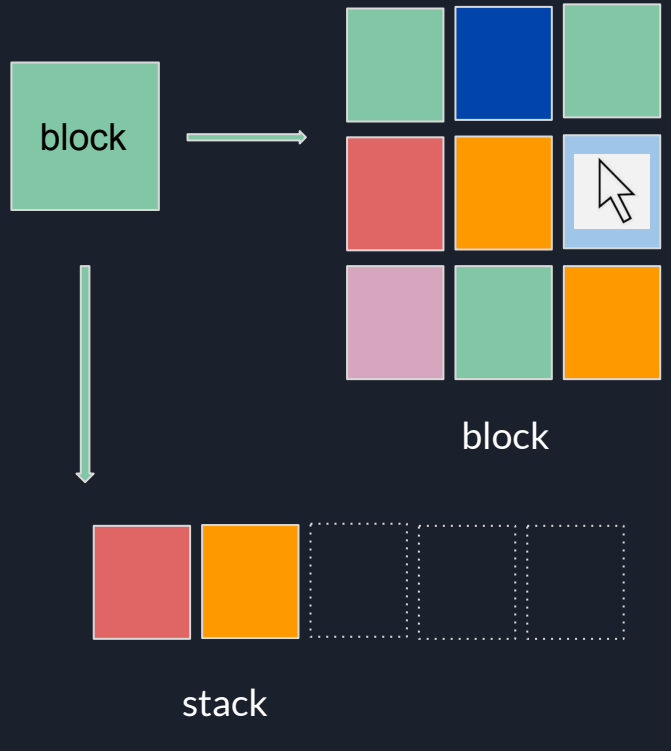
Project Architecture : UI

(1) Display of block and stack

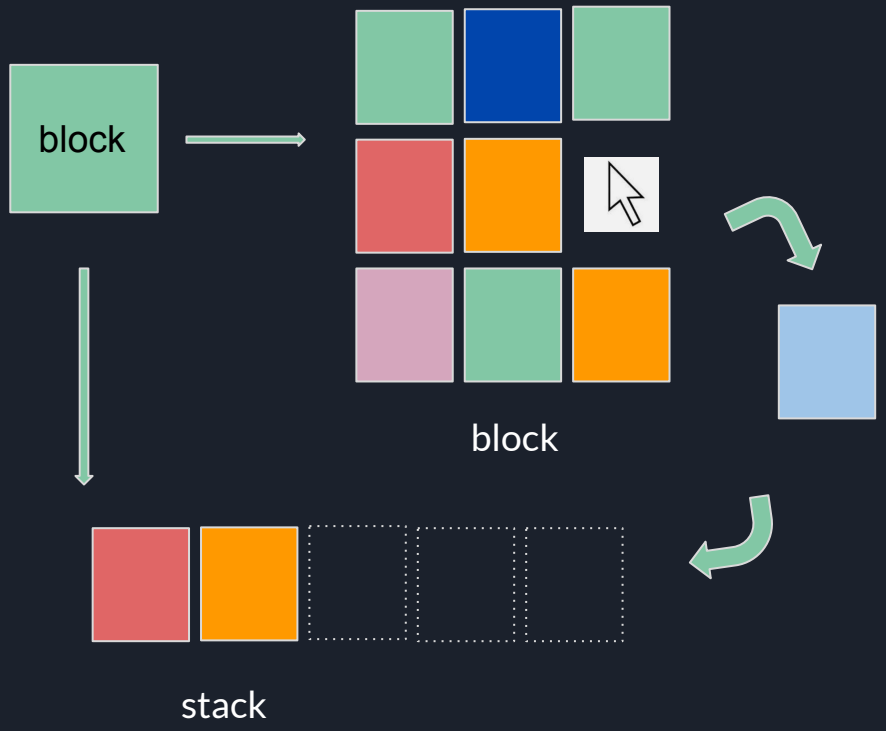


Project Architecture : UI

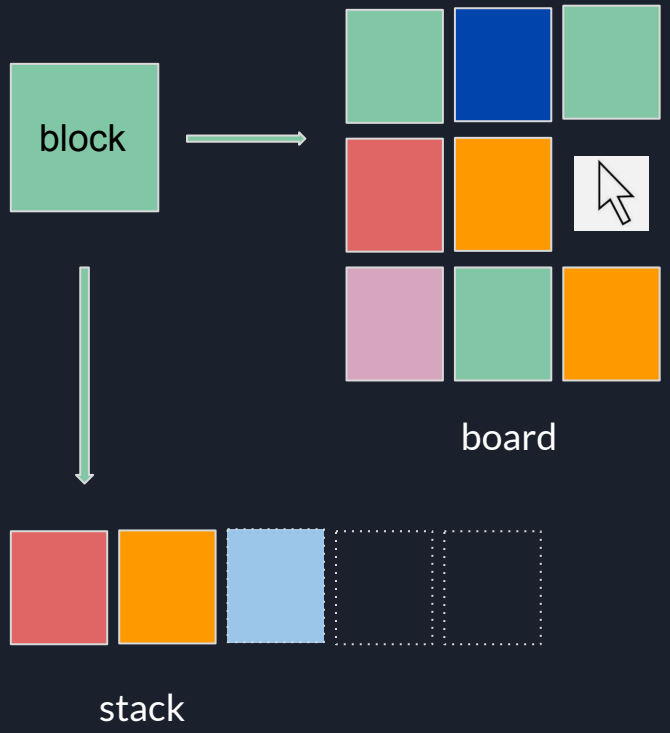
(2) Interaction with mouse click



Project Architecture : UI



Project Architecture : UI



Project Architecutre : Back-End Logic

the higher level object : world (containing stack, board, mousePos, status, score…)

user input	update world based on user input
click one block	move the block from board to stack, check stack status, update world status and score
revoke last move	add the last block back to board and delete it from the stack
shuffle the board	rearrange the position of blocks in the board’s first layer

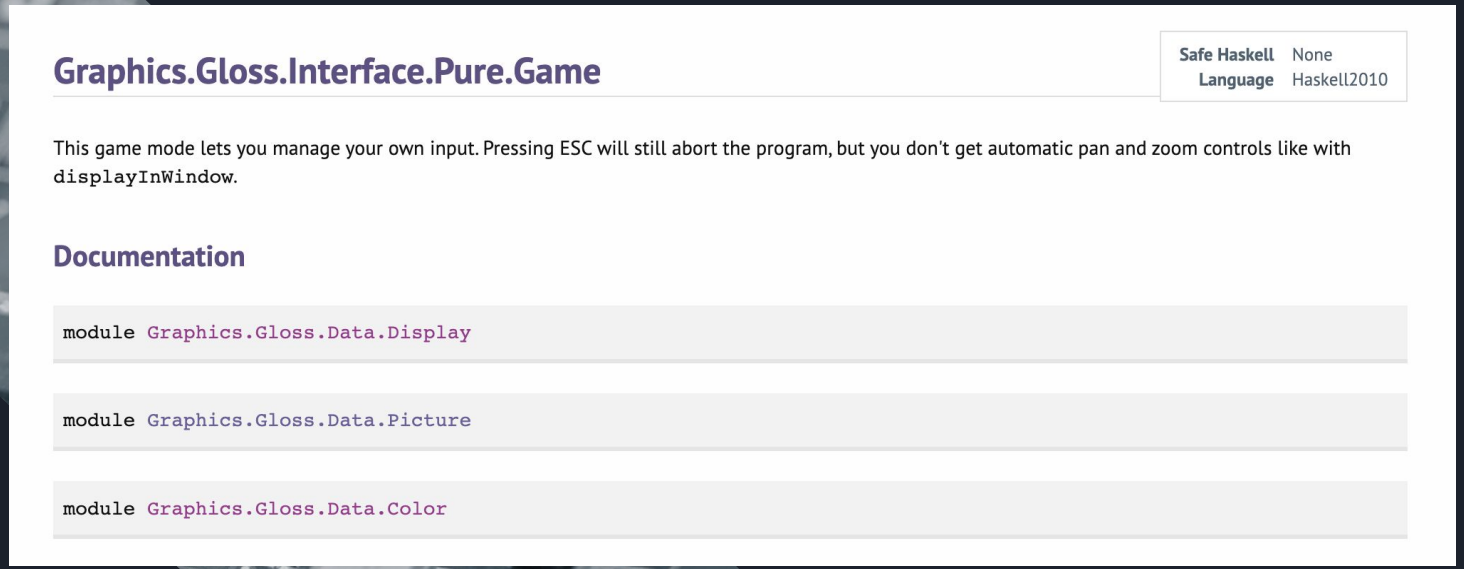
……(later display in the demo)

```haskell
data World = World {
                    wMousePos:: Point
                  , mouseGridPos:: (Int, Int)
                  , stack :: Stack
                  , board :: MBoard
                  , score :: Int
                  , random :: Random.StdGen
                  , status :: Int
                  , showtext :: String
                  , mouseEnter ::Bool
                  , canUndoLast :: Bool
                  , pics :: [Picture]
                  } deriving (Show)
```

```haskell
main :: IO ()
main = do
  ran <- Random.getStdGen
  loadedpics <- loadpictures
  play --FullScreen
   (InWindow "TLE" (1400, 1000) (10, 10))
    backgroundColor
    9
    (World (0,0) (0,0) initstack initboard 0 ran 0 "" False False loadedpics)
    (drawWorld 0)
    handleEvent  -- update world
    (updateWorld 0)
  where
    initstack = Stack False [(0,0,0)] False
    pureGen = mkStdGen 137
    initboard = initBoard pureGen blockNumber
```

Implementation: Libraries

libraries: Data.List, System.Random, Data.Set, Graphics.Gloss.Interface.Pure.Game



Graphics.Gloss.Interface.Pure.Game

Safe Haskell	None
Language	Haskell2010

This game mode lets you manage your own input. Pressing ESC will still abort the program, but you don't get automatic pan and zoom controls like with `displayInWindow`.

Documentation

```
module Graphics.Gloss.Data.Display
```

```
module Graphics.Gloss.Data.Picture
```

```
module Graphics.Gloss.Data.Color
```

Implementation: Functions (examples)

(1) Initialize the Board

```
initBoard :: Random.StdGen -> Int -> MBoard
initBoard pureGen blockNum = formboard coloredboard
    where initboard0 = [(i,j) | i <- [1.. blockNum], j <- [1.. blockNum]]
          rolls n =  take n . unfoldr (Just . uniformR (1, 12))
          initcolors0 = rolls (blockNum * blockNum) pureGen
          initcolors1 = initcolors0 ++ initcolors0 ++ initcolors0
          initcolors = shuffle' initcolors1 (blockNum * blockNum * layers) pureGen
          initboard = tricopy initboard0
          coloredboard = putToghther initboard initcolors
```

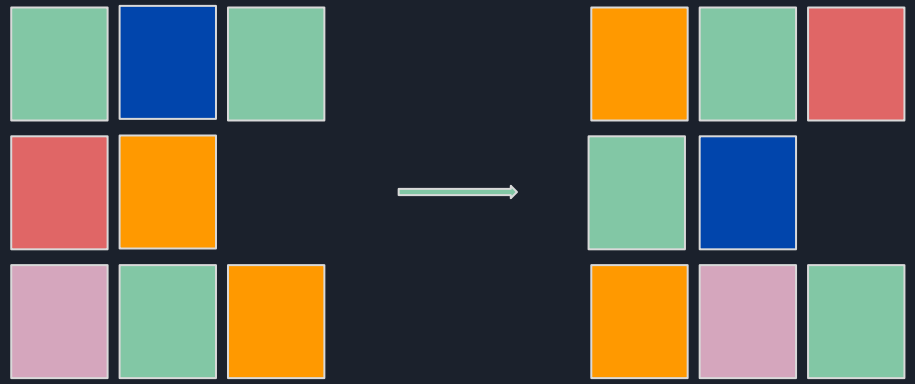
Implementation: Functions (examples)

(2) handle mouse click and keyboard interrupt

step1: validate the mouse position (on one block in the board)
step2: translate the mouse position(float, float) to the block position (int, int)
step3: move the selected block from board to stack

```
handleEvent :: Event -> World -> World
handleEvent e g = case e of
    (EventKey (MouseButton LeftButton) Down _ pos) -> onMouseDown (onMouseMove pos g)
    (EventKey (SpecialKey KeySpace) Down _ _) -> cleanStack g
    (EventKey (SpecialKey KeyLeft) Down _ _) -> undoLastMove g
    (EventKey (SpecialKey KeyRight) Down _ _) -> shuffleboard1 g
    _ -> g
```

Implementation: One challenge – the shuffle function



old board

shuffled board

Test of Back-End Logic: quickCheck

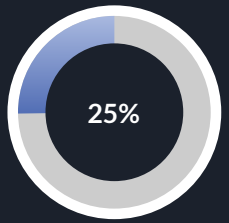
e.g. check proper deletion from the board

prop1: the number of blocks in the board should -1 if the board is not empty
prop2: the deleted block is not in the result any more
prop3:all the rest blocks remained in the result
prop4: not introduce new blocks

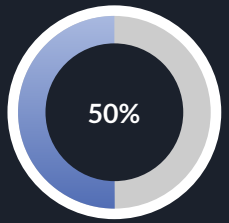
```
get first
  properly get the first item [✔]
    +++ OK, passed 100 tests.
shuffle board1
  properly shuffles board, have the same number of blocks [✔]
    +++ OK, passed 100 tests.
shuffle board2
  properly shuffles board, have the same colors [✔]
    +++ OK, passed 100 tests.
remove last1
  properly remove last, have n-1 blocks [✔]
    +++ OK, passed 100 tests.
remove last2
  properly shuffles board, does not change the content [✔]
    +++ OK, passed 100 tests.
proper putOnBoard
  properly put on board, if pos and color meets, should put on one elem, else do not put any
 elem [✔]
    +++ OK, passed 100 tests.
proper countS
  proper count the number of color [✔]
    +++ OK, passed 100 tests.
proper deleteColor
  do not add new contents( y and c) when delete color [✔]
    +++ OK, passed 100 tests.
proper deleteColor
  x start from 0 [✔]
    +++ OK, passed 100 tests.
delete color
  properly deleted the expected color [✔]
    +++ OK, passed 100 tests.
proper deleteItem
  the item number of deleted board should be less than the original board [✔]
    +++ OK, passed 100 tests.
proper deleteItem
  the required item is deleted [✔]
    +++ OK, passed 100 tests.
```

Demo

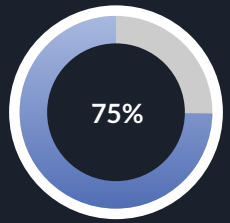
Summary



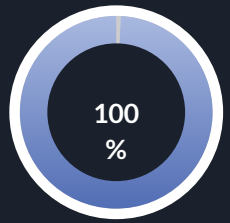
game formation



architecture



implementation



test and demo

Thanks for your attention !