Othello(Reversi)

A single-player and multiplayer game in Haskell using the libraries gloss for graphics and
cabal.

Harshit Patel
CS16BTECH11016

Gorikapudi Prudhvi
CS16BTECH11017

Bhrat Gupta
CS16BTECH11007

CONTENTS -

Rules of the Game -

Reversi is a strategy board game for two players, played on an 8×8 uncheckered board. There are sixty-four identical game pieces called *disks* which are light on one side and dark on the other. Players take turns placing disks on the board with their assigned color facing up. During a play, any disks of the opponent's color that are in a straight line and bounded by the disk just placed and another disk of the current player's color are turned over to the current player's color.

Every move of a player is considered valid if it flips atleast one of the discs of the opponent player.

A disc or row of discs is outflanked when it is surrounded at the ends by discs of the opposite color.

A disc may outflank any number of discs in one or more rows in any direction (horizontal, vertical, diagonal).

The objective of the game is to have the majority of disks turned to display the players color when the last playable empty square is filled.

Libraries Used -

- ## Cabal

  Cabal is a system for building and packaging Haskell libraries and programs. It defines a common interface for package authors and distributors to easily build their applications in a portable way. Cabal is part of a larger infrastructure for distributing, organizing, and cataloging Haskell libraries and programs.

  Specifically, the Cabal describes what a Haskell package is, how these packages interact with the language, and what Haskell implementations must to do to support packages. The Cabal also specifies some infrastructure (code) that makes it easy for tool authors to build and distribute conforming packages.

- ## Gloss

  Gloss is a graphics library used for drawing simple vector based graphics. It uses OpenGL under the hood.

  - Graphics.Gloss.Data.Color
  It helps us to build custom colors

  - Graphics.Gloss.Interface.Pure.Game
  It helps us to manage our own input and checks for any events.

- **Data.Array**
  This module provides a general interface to immutable arrays.

Main.hs

This is our main module which gets called when cabal is run using the command -

cabal run

It imports all our game, rendering and logic files

To create a window -

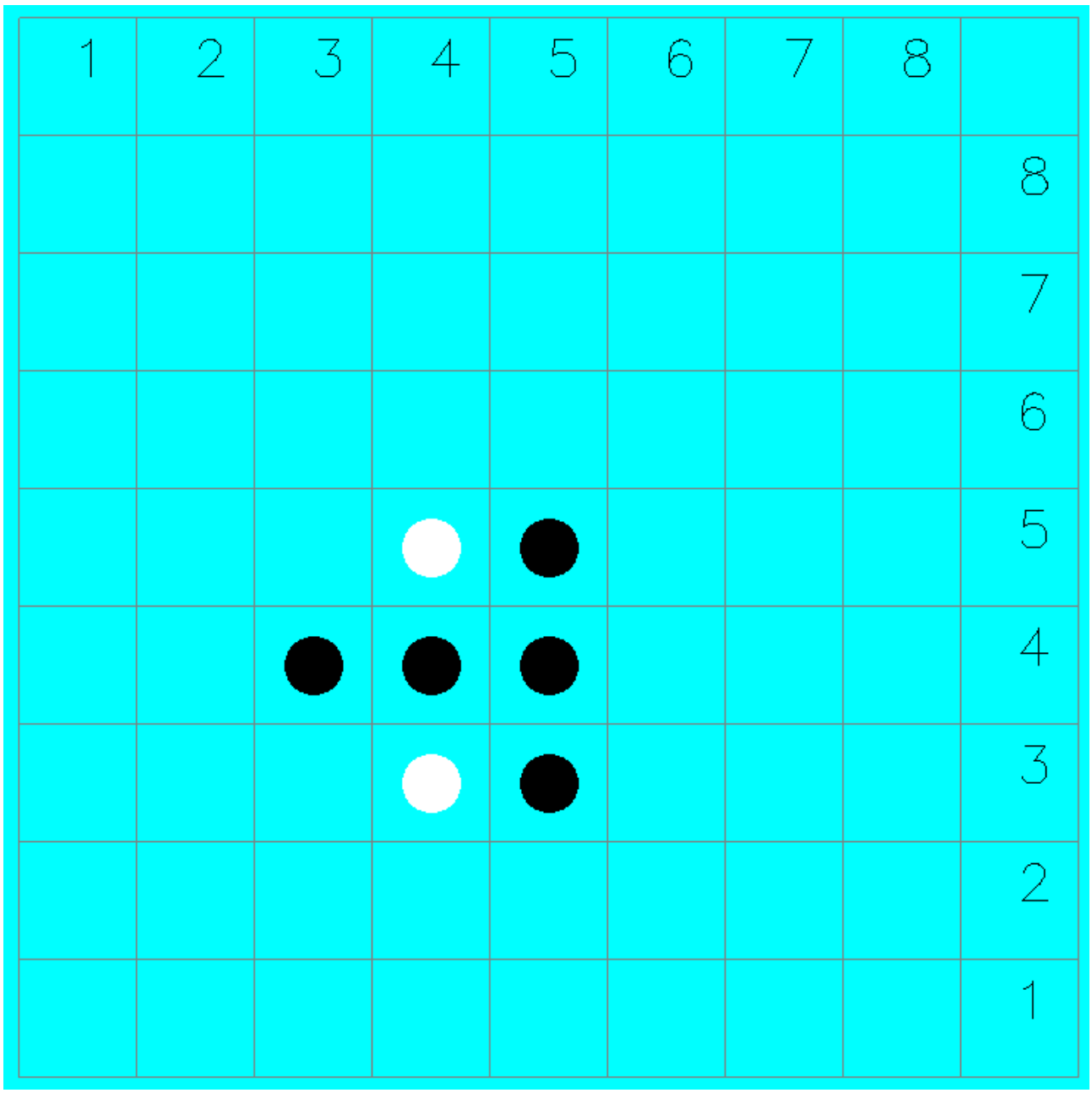
window :: Display
window = InWindow "Othello/Reversi" (screenWidth, screenHeight) (100, 100)

Our main has a play function provided by -
Graphics.Gloss.Interface.Pure.Game

main :: IO ()
main = play window backgroundColor 30 initialGame gameAsPicture (transformGame) (const id)

initialGame is defined in Game.hs
gameAsPicture is function which is called for rendering any graphics from Rendering.hs

transformGame is function which is called for the logic implementation from Logic.hs



Game.hs

This module basically defines the data constructors used, initializes our board and our game and sets the initial game condition.

For our game we are using a data-type Game to represent the state of game.

```
data Game = Game { gameBoard :: Board
                 , gamePlayer :: Player
                 , gameState :: State
               } deriving (Eq, Show)
```

initialGame sets our opening game -
gameState is set to running,
initialPlayer is PlayerB
gameBoard defined is as -

```
Game {
      gameBoard = (array indexRange $ zip (range
indexRange) (cycle [Empty])
      , gamePlayer =  PlayerB
      , gameState = Running
}
```

indexRange varies from (0,0) to (n-1,n-1)
range creates a list for indexRange, zip merges two list by taking the corresponding element from both list and array creates our board array.

Rendering.hs

This module renders our graphics .

Bcell – creates our black cells

wCell – creates our white cells

sCell – displays the text from 1...8

boardGrid – the grid for our board

This module checks the state of game , running or gameover and renders the graphics accordingly.

BoardAsRunningPicture returns picture which are combinations of bCells of the board, wCells of board, xCells of board and the board grid.

The board grid color changes according to the players turn and finally changes to black or white according to the winner of the game.

SnapPictureToCell takes a picture , index of row and column and translates it accordingly.

BoardGrid function uses a concat map to render the grids i.e the horizontal and vertical lines separated by cell width.

Similiarly we render the gameOver picture

Logic.hs

This module is responsible for any change in the state of the game, for the events recognised by any mouse clicks and implements the logic of our game.

```
transformGame (EventKey (MouseButton LeftButton) Up _
mousePos) game =
      case gameState game of
            Running -> playerTurn game $
mousePosAsCellCoord mousePos
```

This function takes a game and looks for an event and correspondingly changes the game state.

The mouse Position clicked by any player is mapped to our cell coords by mousePosAsCellCoord()

```
mousePosAsCellCoord :: (Float, Float) -> (Int, Int)
mousePosAsCellCoord (x, y) = ( floor ((y + (fromIntegral
screenHeight * 0.5)) / cellHeight)
      , floor ((x + (fromIntegral screenWidth * 0.5)) / cellWidth)
      )
```

For any movement of discs we first check the cell coords clicked are valid , they must be adjacent to discs of other player and they must flip atleast one discs of other player in any of the 8 directions.

To check whether the cell coords belongs to board we have isCoordCorrect() function

```
isCoordCorrect = inRange ((0, 0), (m - 1, m – 1))
```

isCoordAdjacent() checks if the discs placed by a player is adjacent to any discs of opposite color.

To check whether there would be any flipping of discs on placing a disc on any cell coord we have functions isUp() ,

isDown(), isLeft(), isRight(), isNE(), isNW(), isSE(), isSW() which checks respectively in each direction if there is a disc of opposite player which can be flipped.

```
isDown :: Game -> (Int, Int) -> Bool
isDown game (x,y) | x-1 >=0 && board!(x-1, y) == (Full $
player) && isendSymbolDown game (x-1,y) = True
| otherwise = False
where board = gameBoard game
player =
if gamePlayer game == PlayerB
then PlayerW
else PlayerB
```

If any of these conditions hold, we place the player disc at that coordinate , flip the opponents discs and switch the player.
If the conditions donot hold we just switch the player.

To flip the discs we traverse through all the 8 directions and maintain a list of discs we want to flip in each direction and finally merge the lists. Then we change the gameboard according to the merged list.

The BOT Logic -

In this othello game the corner positions are the safest as they can never be flipped. These positions provide a strong-hold in the game for that particular player.

The bot first checks if there is a corner piece available in which it is legal to move.It occupies the corner position.
If no corner position is available it follows a greedy approach and checks for all the legal positions which result in the maximum flipping.
This greedy approach though may not be optimal but it provides a sufficiently good move at that game state.

```
botMove :: Game -> (Int,Int)
botMove game | checkValidPositon game (0,0)==True = (0,0)
| checkValidPositon game (0,7)==True = (0,7)
| checkValidPositon game (7,0)==True = (7,0)
| checkValidPositon game (7,7)==True = (7,7)
| getPos game /= (-1,-1) = (getPos game)
```

getPos() returns the cellcoord obtained by following the greedy approach.