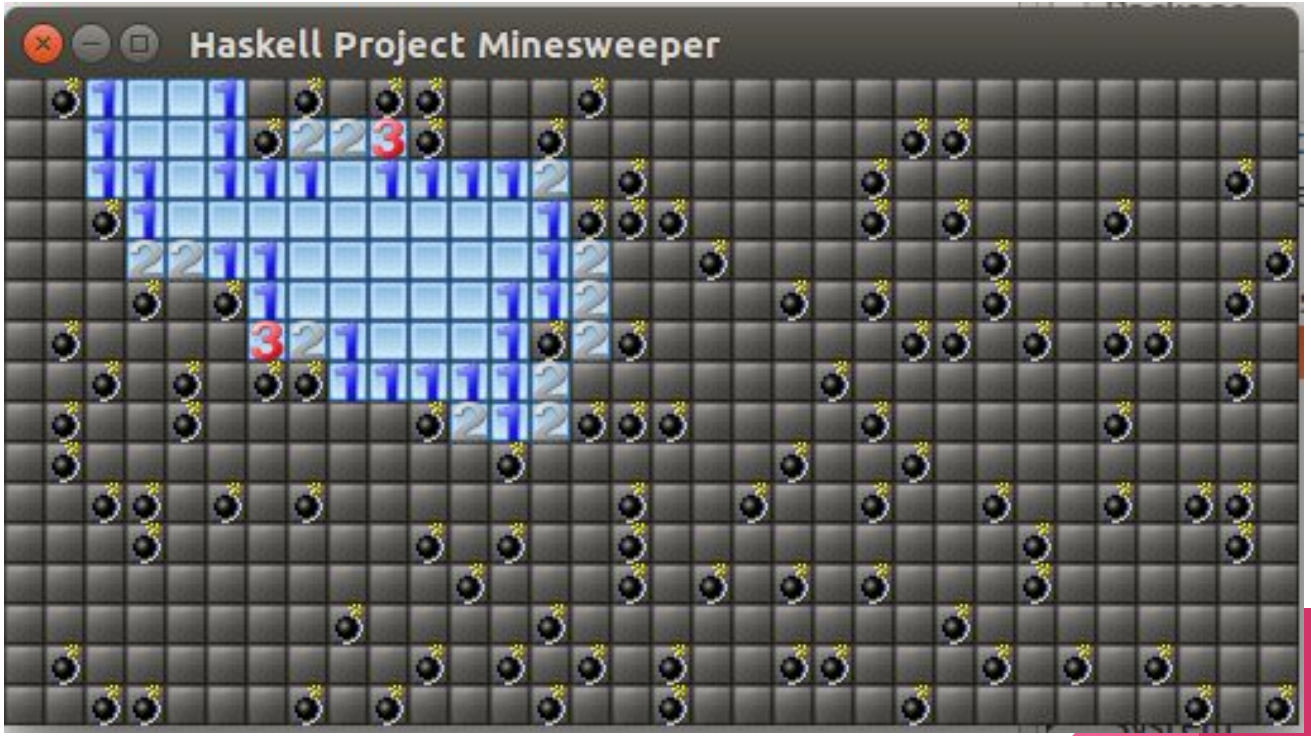
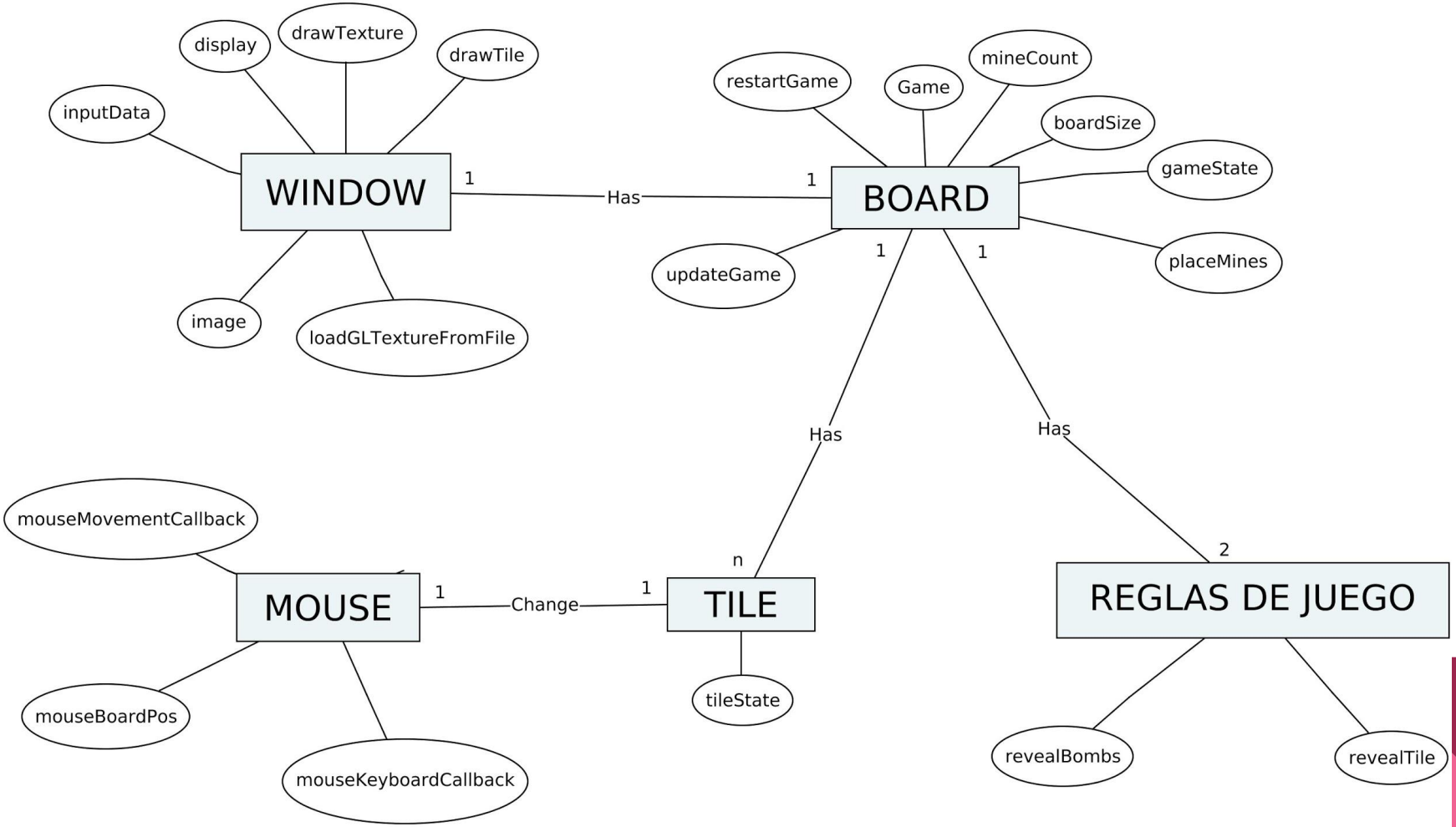
Minesweeper

M. Fernanda Molina - Emil Vega

Yachay Tech

Minesweeper




display
drawTexture
drawTile
inputData
WINDOW
image
loadGLTextureFromFile
1
Has
1
restartGame
Game
mineCount
boardSize
gameState
BOARD
updateGame
placeMines
1
1
Has
Has
n
2
mouseMovementCallback
MOUSE
mouseBoardPos
mouseKeyboardCallback
1
Change
1
TILE
tileState
REGLAS DE JUEGO
revealBombs
revealTile

Libraries

Data.List

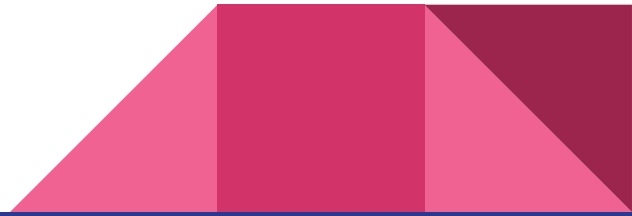
Has a bunch of useful functions /operation on lists

Data.IORef

Works with mutable references in the IO monad.

Graphics.UI.GLUT

The OpenGL Utility Toolkit (GLUT) is a programming interface for writing window system independent OpenGL programs.

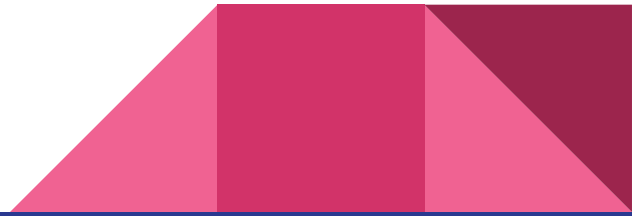


Graphics.GLUtil

Helpers for working with shaders, buffer objects, and textures in OpenGL.

Contol.Aplicative

This module describes a structure intermediate between a functor and a monad.
<?> or <**>



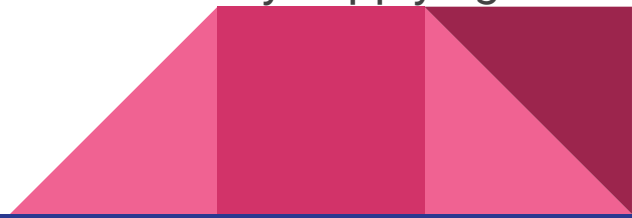
Libraries

Data.Map.Strict

- An efficient implementation of ordered maps from keys to values (dictionaries).

System.Random

- This library deals with the common task of pseudo-random number generation. The library makes it possible to generate repeatable results, by starting with a specified initial random number generator, or to get different results on each run by using the system-initialised generator or by supplying a seed from some other source.



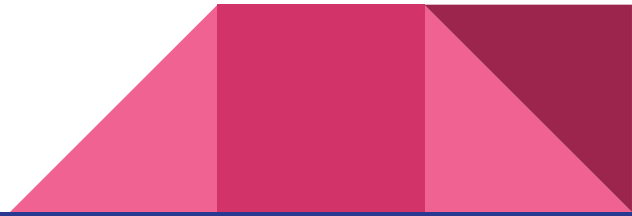
Libraries

Data.Maybe

- The Maybe type encapsulates an optional value. A value of type Maybe a either contains a value of type a (represented as Just a), or it is empty (represented as Nothing).

Control.Monad

- The Functor, Monad and MonadPlus classes, with some useful operations on monads.



Module Input.hs

```haskell
data InputData = InputData { onKeys :: [Key]
                           , mousePos :: Position
                           , previousMousePos :: Position
                           }
```

- data Key

A generalized view of keys

- data Position

A 2-dimensional position, measured in pixels

- Data IORef a

IORef allows to modify states of a.

```haskell
mouseKeyboardCallback :: IORef InputData -> Key -> KeyState -> Modifiers -> Position -> IO ()
mouseKeyboardCallback inputsIORef key keyState _ newMousePos =
    if keyState == Down
      then inputsIORef $~ \x -> x { onKeys = if elem key (onKeys x)
                                             then onKeys x
                                             else key : (onKeys x)
                                  , mousePos = newMousePos }
      else inputsIORef $~ \x -> x { onKeys = L.delete key (onKeys x)
                                  , mousePos = newMousePos }
```

Module Input.hs

```
mouseMovementCallback :: IORef InputData -> Position -> IO ()
mouseMovementCallback inputsIORef newMousePos = inputsIORef $~ \x -> x { previousMousePos = (mousePos x)
                                                                       , mousePos = newMousePos }

cancelOnKeys :: InputData -> InputData
cancelOnKeys input = input { onKeys = [] }-
```

Module Game.hs

```haskell
type Flagged = Bool
type MineCount = Int
type ClickedMinePos = (Int, Int)
data GameState = InitialisedGS | PlayingGS | GameOverGS ClickedMinePos deriving (Eq, Ord, Rea
data TileState = CoveredTS Flagged | MineFreeTS MineCount | MineTS deriving (Eq, Ord, Read, S
```

```haskell
data Game = Game { rndGen :: StdGen
                 , gameState :: GameState
                 , boardSize :: (Int, Int)
                 , board :: SM.Map (Int, Int) TileState
                 , minePositions :: SM.Map (Int, Int) Bool
                 , viewportRes :: (GLsizei, GLsizei)
                 , bitmapAtlas :: TextureObject
                 }
```

Module Game.hs

```haskell
defaultTileSize :: Int
defaultTileSize = 16

defaultMineCount :: Int
defaultMineCount = 100

defaultBoardSize :: (Int, Int)
defaultBoardSize = (32, 16)

-- Reiniciar el juego.
restartGame :: Game -> Game
restartGame g = g { board = board'
                  , gameState = InitialisedGS
                  , minePositions = SM.empty
                  }
    where (boardW, boardH) = boardSize g
          positions = [(x, y) | x <- [0..boardW-1], y <- [0..boardH-1]]
          board' = SM.fromList $ zip positions (repeat $ CoveredTS False)
```

Module Game.hs

```
-- Actualizar el juego según los inputs.
updateGame :: InputData -> Game -> Game
updateGame input game = game'
    where
        -- ***************************** Obtención entradas ******************************
        mousePosition = mouseBoardPos (mousePos input) game
        mouseLeftPressed = elem (MouseButton LeftButton) (pressedKeys input)
        mouseRightPressed = elem (MouseButton RightButton) (pressedKeys input)

        -- ***************************** Actualizaciones del juego ******************************
        game' | mouseLeftPressed = case gameState game of
                                    InitialisedGS -> (revealTile mousePosition) . (placeMines mousePosition defaultMineCount) $ game
{gameState = PlayingGS}
                                    PlayingGS -> case board game SM.! mousePosition of
                                                    CoveredTS _ -> case SM.lookup mousePosition $ minePositions game of
                                                                    Nothing -> revealTile mousePosition game
                                                                    Just _  -> revealBombs $ game {gameState = GameOverGS
mousePosition}
                                                    _           -> game
                                    GameOverGS _ -> restartGame game

              | mouseRightPressed = case gameState game of
                                     PlayingGS -> case board game SM.! mousePosition of
                                                    CoveredTS flagged -> let board' = SM.insert mousePosition (CoveredTS (not flagged)) $
board game
                                                                         in game {board = board'}
                                                  _                   -> game
                                     _         -> game

              | otherwise = game
```

Module Game.hs

```haskell
-- Ubicar todas las minas después de un clic inicial
placeMines :: (Int, Int) -> Int -> Game -> Game
placeMines (clickX, clickY) mineCount game =
    let (boardW, boardH) = boardSize game
        maxMineCount = floor $ (fromIntegral boardW) * (fromIntegral boardH) * 0.75
        placeMines' 0         rndGen' placedMines = game {rndGen = rndGen', minePositions = placedMines}
        placeMines' mineCount rndGen' placedMines = let (mineX, rndGen'') = randomR (0, boardW-1) rndGen'
                                                        (mineY, rndGen''') = randomR (0, boardH-1) rndGen''
                                                        mineTooCloseToFirstClick = abs (clickX - mineX) <= 1 &&
                                                                                   abs (clickY - mineY) <= 1
                                                        alreadyAMine = SM.member (mineX, mineY) placedMines
                                                        placedMines' = SM.insert (mineX, mineY) False placedMines
                                                    in if alreadyAMine || mineTooCloseToFirstClick
                                                         then placeMines' mineCount rndGen''' placedMines
                                                         else placeMines' (mineCount-1) rndGen''' placedMines'
    in placeMines' (max 0 $ min maxMineCount mineCount) (rndGen game) SM.empty
```

Module Game.hs

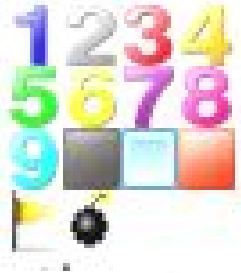
```haskell
-- Revela la celda en la posición dada y actualiza el juego consequentemente.
revealTile :: (Int, Int) -> Game -> Game
revealTile clickedPos game =
    case SM.lookup clickedPos $ minePositions game of
        Nothing -> let (clickX, clickY) = clickedPos
                       (boardW, boardH) = boardSize game
                       surrounding = [(x,y) | x <- [clickX-1 .. clickX+1], x >= 0, x < boardW
                                            , y <- [clickY-1 .. clickY+1], y >= 0, y < boardH
                                            , (x,y) /= clickedPos]
                       surroundingMineCount = length $ filter (\pos -> SM.member pos $ minePositions game) $ surrounding
                       isCovered tileState = case tileState of
                                                 CoveredTS _ -> True
                                                 _           -> False
                       surroundingToVisit = filter (\pos -> isCovered $ board game SM.! pos) surrounding
                       board' = SM.insert clickedPos (MineFreeTS surroundingMineCount) $ board game
                       game' = game {board = board'}
                   in case surroundingMineCount of
                          0 -> foldr revealTile game' surroundingToVisit
                          n -> game'
        Just _  -> let board' = SM.insert clickedPos MineTS $ board game
                   in game {board = board'}
```

Module Game.hs

```haskell
-- Revela todas las bombas.
revealBombs :: Game -> Game
revealBombs game = foldr (\pos g -> revealTile pos g) game $ SM.keys $ minePositions game

-- Convierte las coordenadas de la ventana en las coordenadas del tablero de juego.
mouseBoardPos :: Position -> Game -> (Int, Int)
mouseBoardPos (Position mX mY) game = let (resW, resH) = viewportRes game
                                          (boardW, boardH) = boardSize game
                                      in (floor $ (fromIntegral mX / fromIntegral resW) * fromIntegral boardW,
                                          floor $ (fromIntegral mY / fromIntegral resH) * fromIntegral boardH)
```

Module OpenGLUtils.hs



```
loadGLTextureFromFile :: FilePath -> IO TextureObject
loadGLTextureFromFile f = do t <- either error id <$> readTexture f
                             textureFilter Texture2D $= ((Linear', Nothing), Linear')
                             texture2DWrap $= (Mirrored, ClampToEdge)
                             return t
..............
drawTexture :: (GLfloat,GLfloat) -> (GLfloat,GLfloat) -> (GLfloat,GLfloat) -> (GLfloat,GLfloat) -> IO()
drawTexture (u,v) (uw,vh) (x,y) (w,h) = do
    renderPrimitive Quads $ do
        tex (u)      (v - vh) >> ver (x)     (y - h) -- Top left coor: (-1, 1)
        tex (u + uw) (v - vh) >> ver (x + w) (y - h)
        tex (u + uw) (v)      >> ver (x + w) (y)....
        tex (u)      (v)      >> ver (x)     (y)....
        where ver x y = vertex (Vertex2 x y :: Vertex2 GLfloat)
              tex u v = texCoord (TexCoord2 u v :: TexCoord2 GLfloat)
```

Module Render.hs

```haskell
data AtlasUV = CoveredUV | UncoveredUV | FlagUV | MineUV | DigitUV Int deriving (Eq, Ord, Read, Show)
```

```haskell
display :: IORef Game -> DisplayCallback
display gameIORef = do
  game <- readIORef gameIORef

  clear [ColorBuffer, DepthBuffer]
  mapM_ (drawTile game) $ SM.assocs $ board game
  color $ Color3 1 1 (1 :: GLfloat)
  flush
```

Module Render.hs

```
drawTile :: Game -> ((Int, Int), TileState) -> IO()
drawTile game ((x,y), tileState) = do
    let (boardW, boardH) = boardSize game
        (tileW, tileH) = (2 / fromIntegral boardW,
                          2 / fromIntegral boardH)
        x' =  ((fromIntegral x / fromIntegral boardW) - 0.5) * 2
        y' = -((fromIntegral y / fromIntegral boardH) - 0.5) * 2
        drawTexture' uv = drawTexture uv (1/4, 1/4) (x',y') (tileW, tileH)
    case tileState of
         CoveredTS True -> do drawTexture' $ imageUV CoveredUV
                              drawTexture' $ imageUV FlagUV

         CoveredTS _    -> do drawTexture' $ imageUV CoveredUV

         MineFreeTS mc  -> do drawTexture' $ imageUV UncoveredUV
                              when (mc > 0) $ do
                                drawTexture' $ imageUV (DigitUV mc)

         MineTS         -> do drawTexture' $ imageUV CoveredUV
                              drawTexture' $ imageUV MineUV
```

Module Render.hs

```
imageUV :: AtlasUV -> (GLfloat, GLfloat)
imageUV CoveredUV         = (1/4, -2/4)
imageUV UncoveredUV       = (2/4, -2/4)
imageUV FlagUV            = (0/4, -3/4)
imageUV MineUV            = (1/4, -3/4)
imageUV (DigitUV n)       = let u = fromIntegral (mod (n-1) 4) / 4
                                v = -(fromIntegral $ div (n-1) 4)
                            in (u, v)
```