FUNCTIONAL REACTIVE PROGRAMMING IN GAMES

Elise Huard - CodeMesh 2015
--# LANGUAGE RecursiveDo #--
{-# LANGUAGE PackageImports #-}
{-# OPTIONS_GHC -fno-warn-type-defaults #-}
module Hunted.Game (  
  hunted  
) where

import Hunted.GameTypes
import Hunted.Sound
import Hunted/Graphics

import FRP.Elerea.Simple as Elerea
import Control.Applicative ((<<$>, <*>), liftA2, pure)
import Data.Maybe (mapMaybe)
import Data.Foldable (foldl')
import System.Random (random, RandomGen(..), randomR)

initialPlayer :: Player
initialPlayer = Player (0, 0) Nothing Nothing

initialMonster :: (Float, Float) -> Monster
initialMonster pos = Monster pos (Wander WalkUp wanderDist) 4

initialViewport :: ViewPort
initialViewport = ViewPort { viewportTranslate = (0, 0), viewportRotate = (), viewportScale = viewportScale }

worldWidth :: Float
worldWidth = 2560

worldHeight :: Float
worldHeight = 1920
FRP
data Signal a
    Monad, Applicative, Functor

data SignalGen a
    Monad, Applicative, Functor, MonadFix
The Salespitch

initialPlayer :: Player
initialPlayer = Player (0, 0) Nothing Nothing

initialMonster :: (Float, Float) -> Monster
initialMonster pos = Monster pos (Wander WalkUp wander 0.4)

initialViewport :: ViewPort
initialViewport = ViewPort { viewportTranslate = (0, 0), viewportRotate = 0, viewportScale = viewportScale }

worldWidth :: Float
worldWidth = 2560

worldHeight :: Float
worldHeight = 1920

AT 3 o'clock

Picking flowers (100, 20)
game :: RandomGen t
    => Signal (Bool, Bool, Bool, Bool)
    => t
    => SignalGen (IO ())

game directionKey randomGenerator = mdo
    randomNumber <- stateful (undefined, randomGenerator) nextRandom
    player <- transfer2 initialPlayer (movePlayer 10) directionKey gameOver'
    monster <- transfer3 initialMonster wanderOrHunt player randomNumber gameOver'
    gameOver <- memo (playerEaten <$> player <*> monster)
    gameOver' <- delay False gameOver
    return $ renderFrame win glossState <$> player <*> monster <*> gameOver
start :: SignalGen (Signal a)
      -> IO (IO a)

network <- start $ game directionKey randomGenerator
fix $ \loop -> do
  readInput win directionKeySink
  join network
  threadDelay 20000
  esc <- exitKeyPressed win
  unless esc loop
(directionKey, directionKeySink) <-
   external (False, False, False, False)

(l,r,u,d) <- (,,,) <$>
keyIsPressed window Key'Left
<*> keyIsPressed window Key'Right
<*> keyIsPressed window Key'Up
<*> keyIsPressed window Key'Down
directionKeySink (l, r, u, d)
simpleSignal <- stateful 2 (+3)

randomNumber <- stateful (undefined, randomGenerator) nextRandom
player <-
  transfer2 initialPlayer
  movePlayer
directionKey
gameOver'

monster <-
  transfer3 initialMonster
  wanderOrHunt
player
randomNumber
gameOver'
gameState = GameState $> renderState <*> soundState
ARROWS -> PLAYER

PLAYER -> MONSTER

MONSTER -> RANDOM

GAME OVER!
game :: RandomGen t
  => Signal (Bool, Bool, Bool, Bool)
  => t
  => SignalGen (IO ())

game directionKey randomGenerator = mdo
  player <- transfer2 initialPlayer (movePlayer 10) directionKey gameOver' 
  randomNumber <- stateful (undefined, randomGenerator) nextRandom
  monster <- transfer3 initialMonster wanderOrHunt player randomNumber gameOver'
  gameOver <- memo (playerEaten <$> player <*> monster)
  gameOver' <- delay False gameOver
  return $ renderFrame win glossState <$> player <*> monster <*> gameOver
module Hunted.Game (  
  hunted  
) where

import Hunted.GameTypes
import Hunted.Sound
import Hunted.Graphics

import FRP.Elerea.Simple as Elerea
import Control.Applicative (((<>)), (<*>), liftA2, pure)
import Data.Maybe (mapMaybe)
import Data.Foldable (foldl')
import System.Random (random, RandomGen, randomR)

initialPlayer :: Player
initialPlayer = Player (0, 0) Nothing Nothing

initialMonster :: (Float, Float) -> Monster
initialMonster pos = Monster pos (Wander WalkUp wanderDist) 4

initialViewport :: ViewPort
initialViewport = ViewPort { viewportTranslate = (0, 0), viewportRotate = 0, viewportScale = viewportScale }

worldWidth :: Float
worldWidth = 2560

worldHeight :: Float
worldHeight = 1920
generator :: Signal (SignalGen a)
    -> SignalGen (Signal a)

playLevel :: Signal (Bool, Bool, Bool, Bool) -- event signals
    -> LevelNumber -- pattern match on level number
    -> Score
    -> Health
    -> SignalGen (Signal GameState, Signal Bool)

-- in playGame main function
(gameState, levelTrigger) <-
    switcher $ playLevel directionKey <$> levelCount' <*> score' <*> lives'
Hi Score

Network

Start Game

Game

Level

"Your Game is Over"

"Here Be Dragons (150, 50)"

Levels

"Level 2 Plz"
module Hunted.Game (
  hunted
) where

import Hunted.GameTypes
import Hunted.Sound
import Hunted.Graphics

import FRP.Elerea.Simple as Elerea
import Control.Applicative ((<$>, (<>)), liftA2, pure)
import Data.Maybe (mapMaybe)
import Data.Foldable (foldl)
import Graphics.Gloss
import System.Random (randomRs, randomR)

initialPlayer :: Player
initialPlayer = Player (0, 0) Nothing Nothing

initialMonster :: (Float, Float) -> Monster
initialMonster pos = Monster pos (Wander WalkUp wanderDist) 4

initialViewport :: ViewPort
initialViewport = ViewPort { viewPortTranslate = (0, 0), viewPortRotate = 0, viewPortClip = (viewportClip), viewportScale }

worldWidth :: Float
worldWidth = 2560

worldHeight :: Float
worldHeight = 1920
bolts <- transfer2 []
manageBolts
shootKey
player
let bolt direction range startPosition =
    stateful (Bolt startPosition direction range False) moveBolt
mkShot shot currentPlayer = if hasAny shot
    then (:[]) <$> bolt (dirFrom shot) boltRange (position currentPlayer)
    else return []
newBolts <- generator (mkShot <$> shoot <*> player)
bolts <- collection newBolts (boltIsAlive worldDimensions <$> monsters)
collection :: (Signal [Signal Bolt])
    -> Signal (Bolt -> Bool)
    -> SignalGen (Signal [Bolt])

collection source isAlive = mdo
    boltSignals <- delay [] (map snd <$> boltsAndSignals')
    -- add new bolt signals
    bolts <- memo (liftA2 (+) source boltSignals)
    let boltsAndSignals = zip <$> (sequence <<< bolts) <*> bolts
        -- filter out dead ones
    boltsAndSignals' <- memo (filter <$> ((.fst) <$> isAlive) <*> boltsAndSignals)
    return $ map fst <$> boltsAndSignals'
module Hunted.Game (hunted)

import Hunted.GameTypes
import Hunted.Sound
import Hunted.Graphics

import FRP.Elerea.Simple as Elerea
import Control.Applicative ((<<$>), (<*>), liftA2, pure)
import Data.Maybe (mapMaybe)
import Data.Foldable (foldl’)
import System.Random (random, RandomGen)

initialPlayer :: Player
initialPlayer = Player (0, 0) Nothing Nothing

initialMonster :: (Float, Float) -> Monster
initialMonster pos = Monster pos (Wander WalkUp wanderDist) 4

initialViewport :: ViewPort
initialViewport = ViewPort { viewportTranslate = (0, 0), viewportRotate = rotateR, viewportScale }

worldWidth :: Float
worldWidth = 2560

worldHeight :: Float
worldHeight = 1920
execute :: IO a
    -> SignalGen a

effectful :: IO a
    -> SignalGen (Signal a)
module Hunted.Game (hunted) where

import Hunted.GameTypes
import Hunted.Sound
import Hunted.Graphics

import FRP.Elerea.Simple as Elerea
import Control.Applicative (((<>), (<*>), liftA2, pure)
import Data.Maybe (mapMaybe)
import Data.Foldable (foldl’)
import System.Random (random, RandomGen)

initialPlayer :: Player
initialPlayer = Player (0, 0) Nothing Nothing

initialMonster :: (Float, Float) -> Monster
initialMonster pos = Monster pos (Wander WalkUp wander 4)

initialViewport :: ViewPort
initialViewport = ViewPort { contentViewRect = (-8, -8) viewPortRotate = 0, viewPortScale = viewportScale }

worldWidth :: Float
worldWidth = 2560

worldHeight :: Float
worldHeight = 1920
module Hunted.Game (hunted)

import Hunted.GameTypes
import Hunted.Sound
import Hunted.Graphics

import FRP.Elerea.Simple as Elerea
import Control.Applicative (liftA2, pure)
import Data.Maybe (mapMaybe)
import Data.Foldable (foldl')
import System.Random (random, RandomGen(..), random)

initialPlayer :: Player
initialPlayer = Player (0, 0) Nothing Nothing

initialMonster :: (Float, Float) -> Monster
initialMonster pos = Monster pos (Wander WalkUp wander 4)

initialViewport :: ViewPort
initialViewport = ViewPort { viewPortTranslate = (0, 0), viewPortRotate = 0, viewPortScale = viewportScale }

worldWidth :: Float
worldWidth = 2560

worldHeight :: Float
worldHeight = 1920
SOME ADDED COMPLEXITY IN HANDLING INFRASTRUCTURE
module Hunted.Game

  hunted
) where

import Hunted.GameTypes
import Hunted.Sound
import Hunted.Graphics

import FRP.Elerea.Simple as Elerea
import Control.Applicative ((<$>), (<*>), liftA2, pure)
import Data.Maybe (mapMaybe)
import Data.Foldable (foldl’)
import System.Random (random, RandomGen(..), randomR)

initialPlayer :: Player
initialPlayer = Player (0, 0) Nothing Nothing

initialMonster :: (Float, Float) -> Monster
initialMonster pos = Monster pos (Wander WalkUp wander 4)

initialViewport :: ViewPort
initialViewport = ViewPort { viewportTranslate = (0, 0), viewportRotate = 0, viewportScale = viewportScale }

worldWidth :: Float
worldWidth = 2560

worldHeight :: Float
worldHeight = 1920
Conceptually simpler
(smaller units)
TESTABILITY
prop_insideLimits move player@(Player (x,y) _) =
  (x > ((-worldWidth) `quot` 2 + playerSize `quot` 2)) &&
  (x < (worldWidth `quot` 2 - playerSize `quot` 2)) &&
  (y > ((-worldHeight) `quot` 2 + playerSize `quot` 2)) &&
  (y < (worldHeight `quot` 2 - playerSize `quot` 2))
  ==> not $ (\p -> outsideOfLimits (worldWidth, worldHeight) p playerSize)
  $ position
  $ movePlayer playerSpeed (worldWidth, worldHeight) move Nothing (False, False, False, False) Nothing player