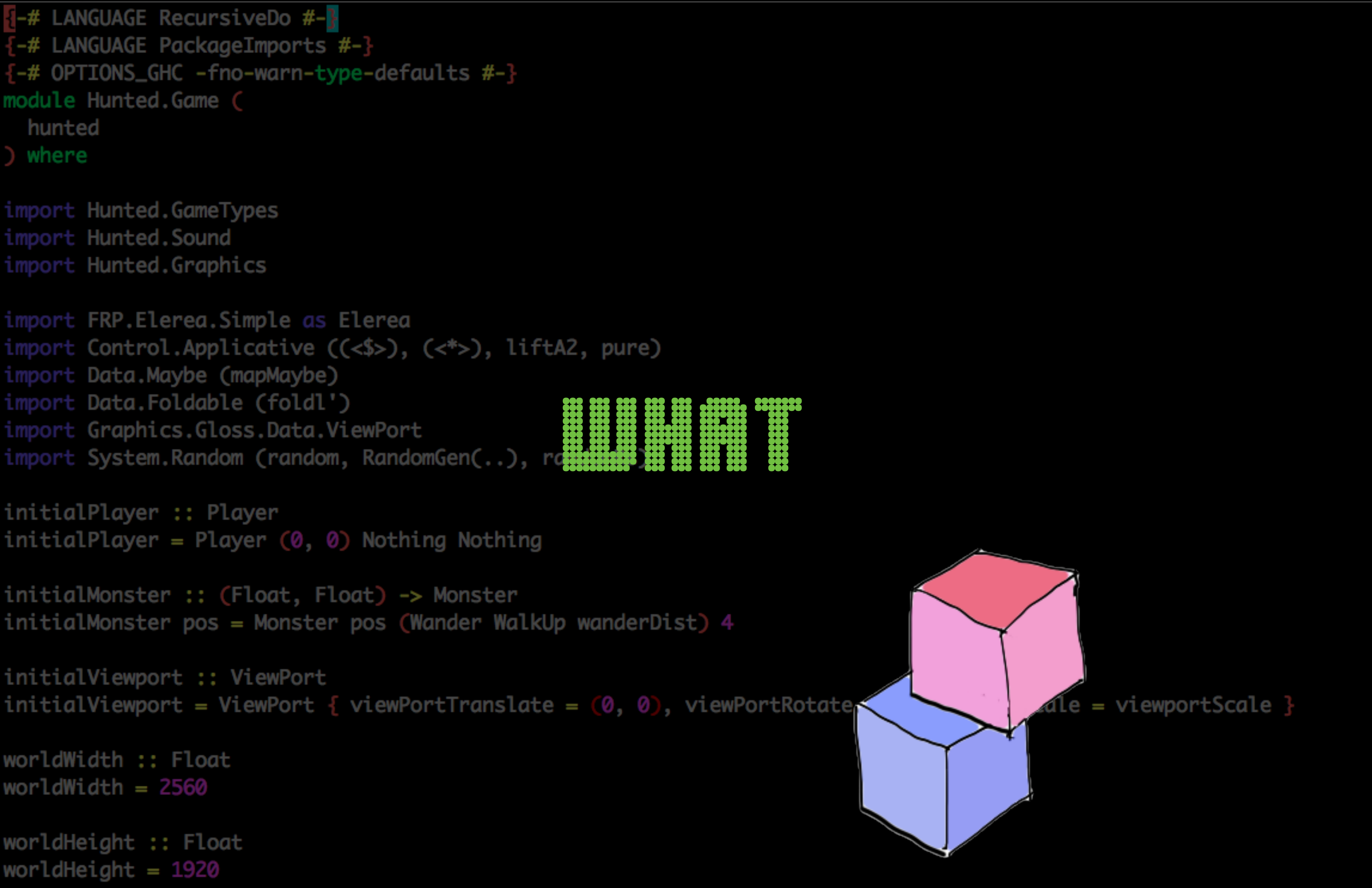
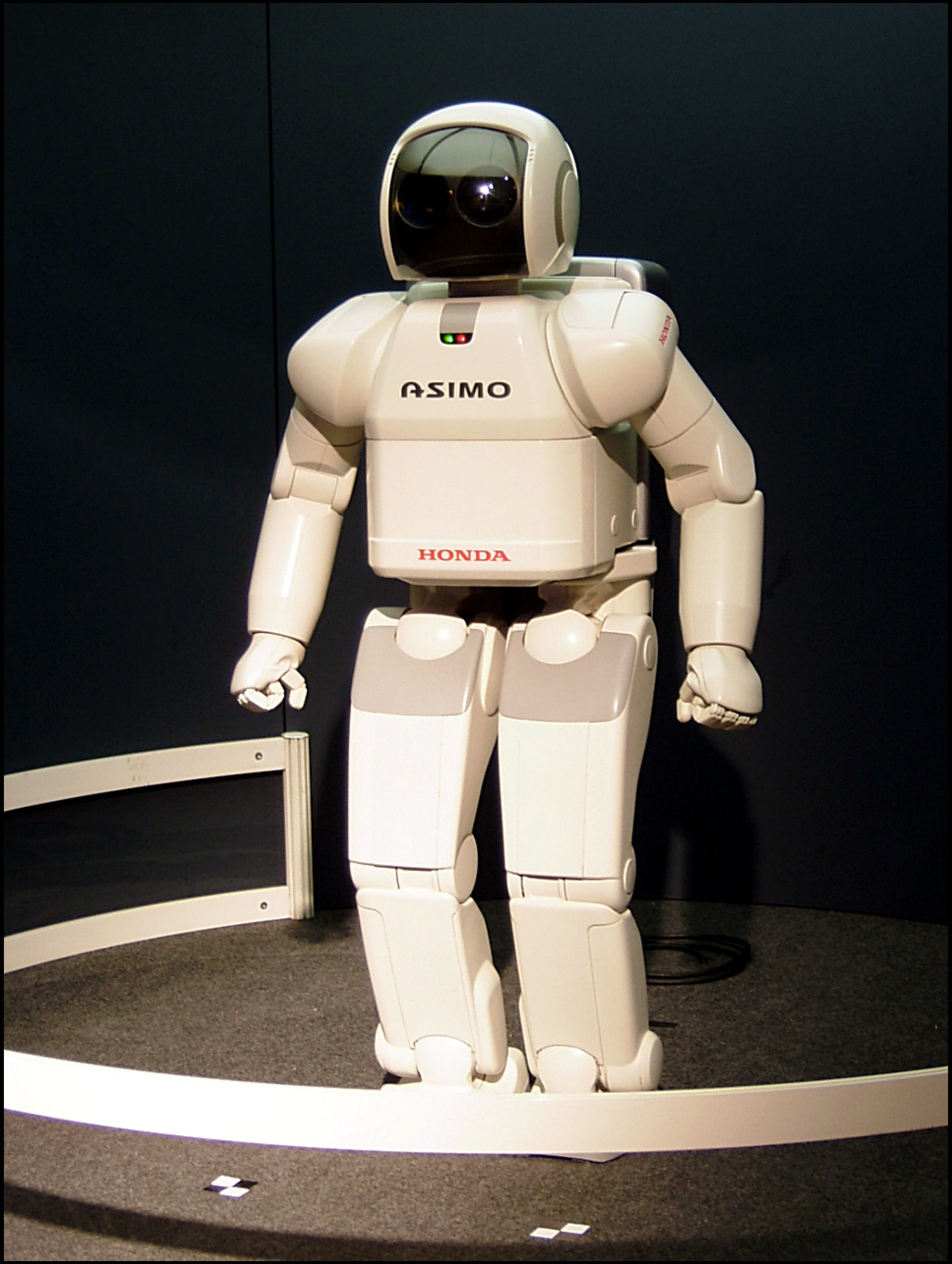
FUNCTIONAL REACTIVE PROGRAMMING IN GAMES

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 Graphics.Gloss.Data.ViewPort
import System.Random (random, RandomGen(..), ra

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          ale = viewportScale }

worldWidth :: Float
worldWidth = 2560

worldHeight :: Float
worldHeight = 1920
```

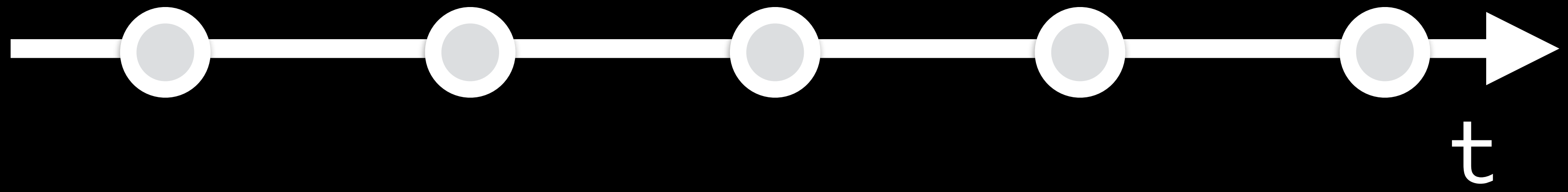
WHAT


ASIMO
HONDA

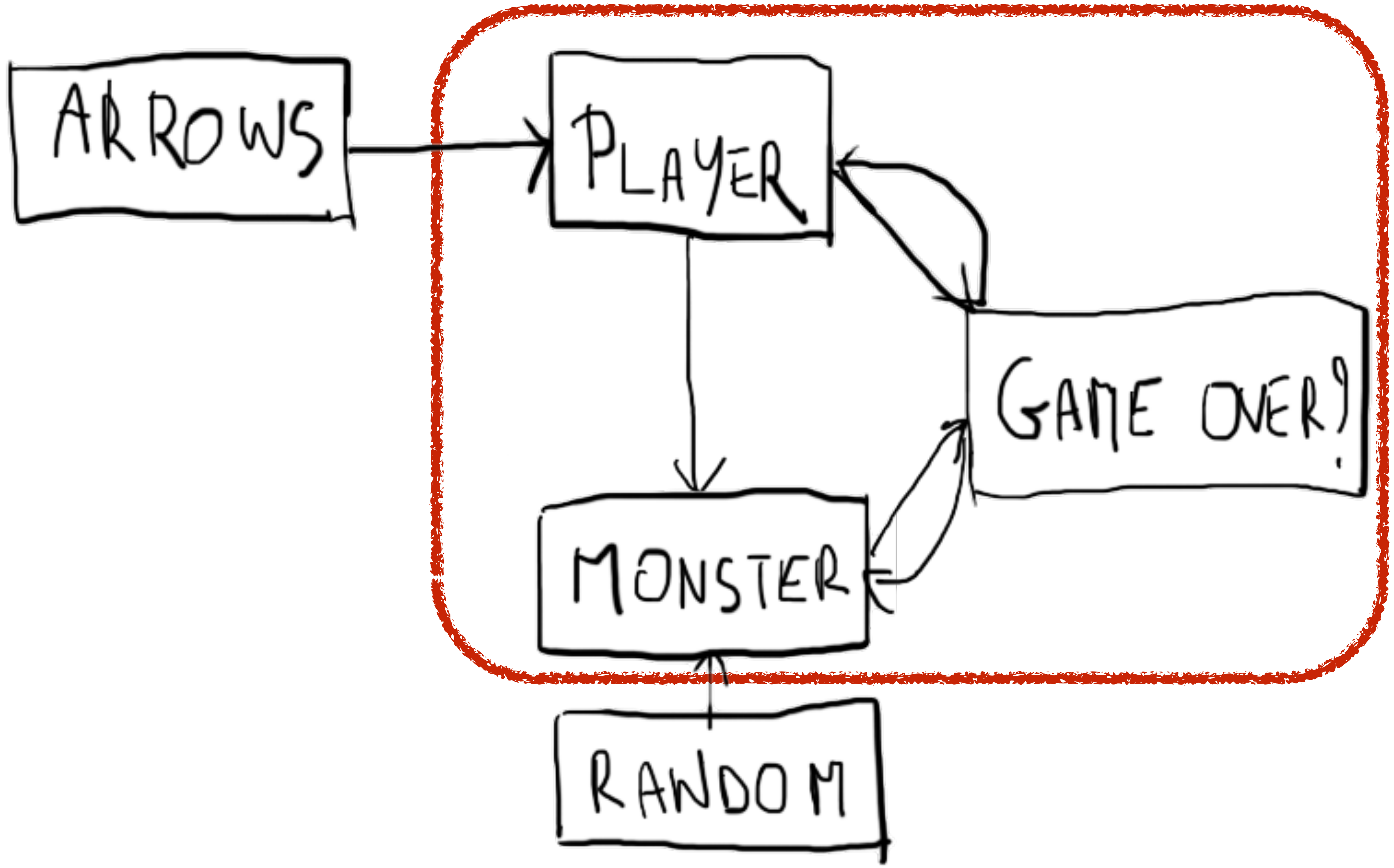
FRP

Elerea https://github.com/cobbpg/elerea

```
data Signal a
  Monad, Applicative, Functor
```



```
data SignalGen a
  Monad, Applicative, Functor, MonadFix
```

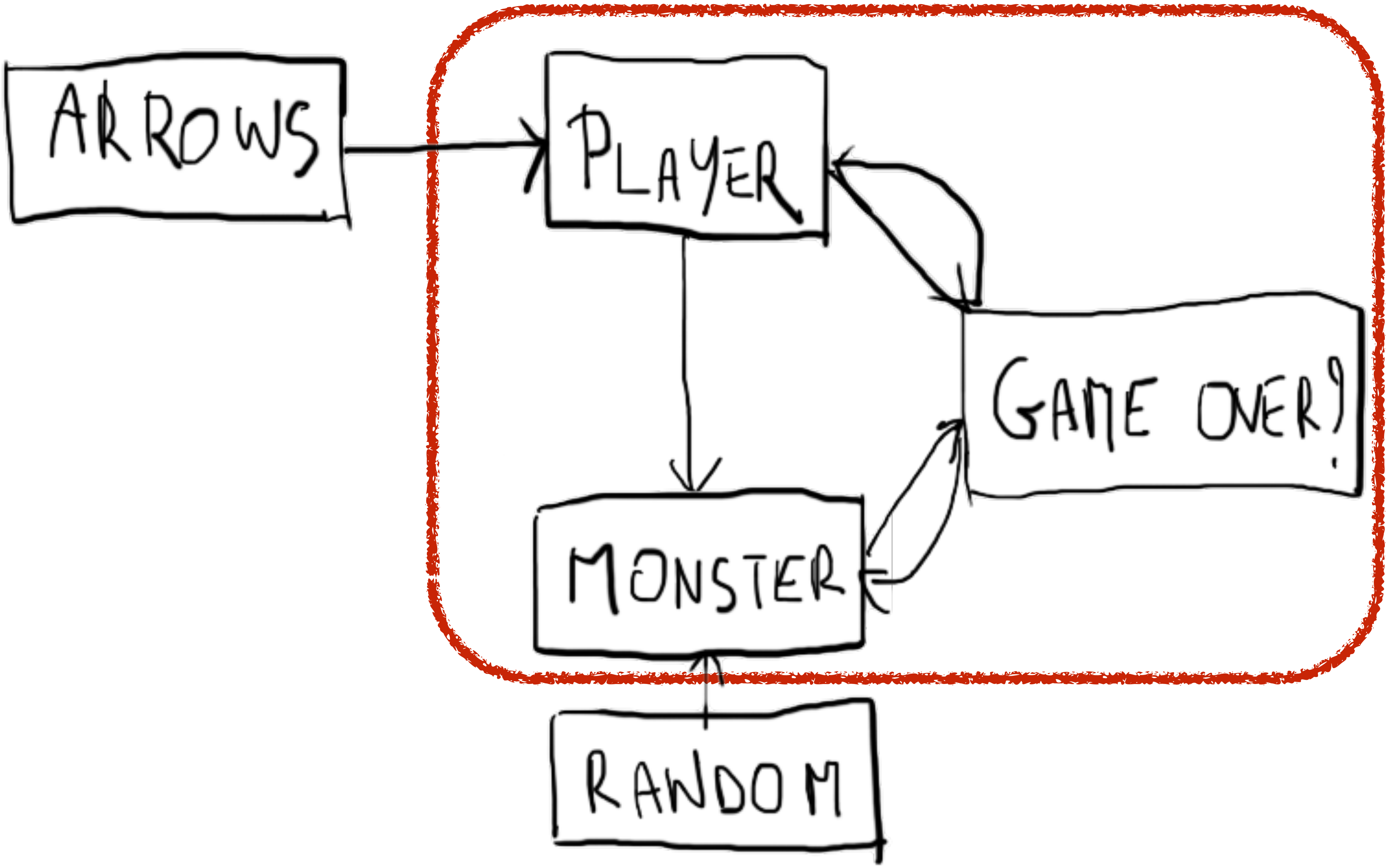

ARROWS
PLAYER
GAME OVER?
MONSTER
RANDOM

THE SALESPITCH

```
-# LANGUAGE RecursiveDo #-}
-# LANGUAGE PackageImports #-}
-# OPTIONS_GHC -fno-warn-type-defaults #-}
odule Hunted.Game (
 hunted
 where

mport Hunted.GameTypes
mport Hunted.Sound
mport Hunted.Graphics

mport FRP.Elerea.Simple as Elerea
mport Control.Applicative ((<$>), (<*>), liftA2, pure)
mport Data.Maybe (mapMaybe)
mport Data.Foldable (foldl')
mport Graphics.Gloss.Data.ViewPort
mport System.Random (random, [illegible]), randoms

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

nitialMonster :: (Float, Float) -> Monster
nitialMonster pos = Monster pos (Wander WalkUp wande[illegible]) 4

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

orldWidth :: Float
orldWidth = 2560

orldHeight :: Float
orldHeight = 1920
```

AT 9 o' CLOCK

PICKING FLOWERS (100, 20)


ARROWS
PLAYER
GAME OVER?
MONSTER
RANDOM

```haskell
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
```

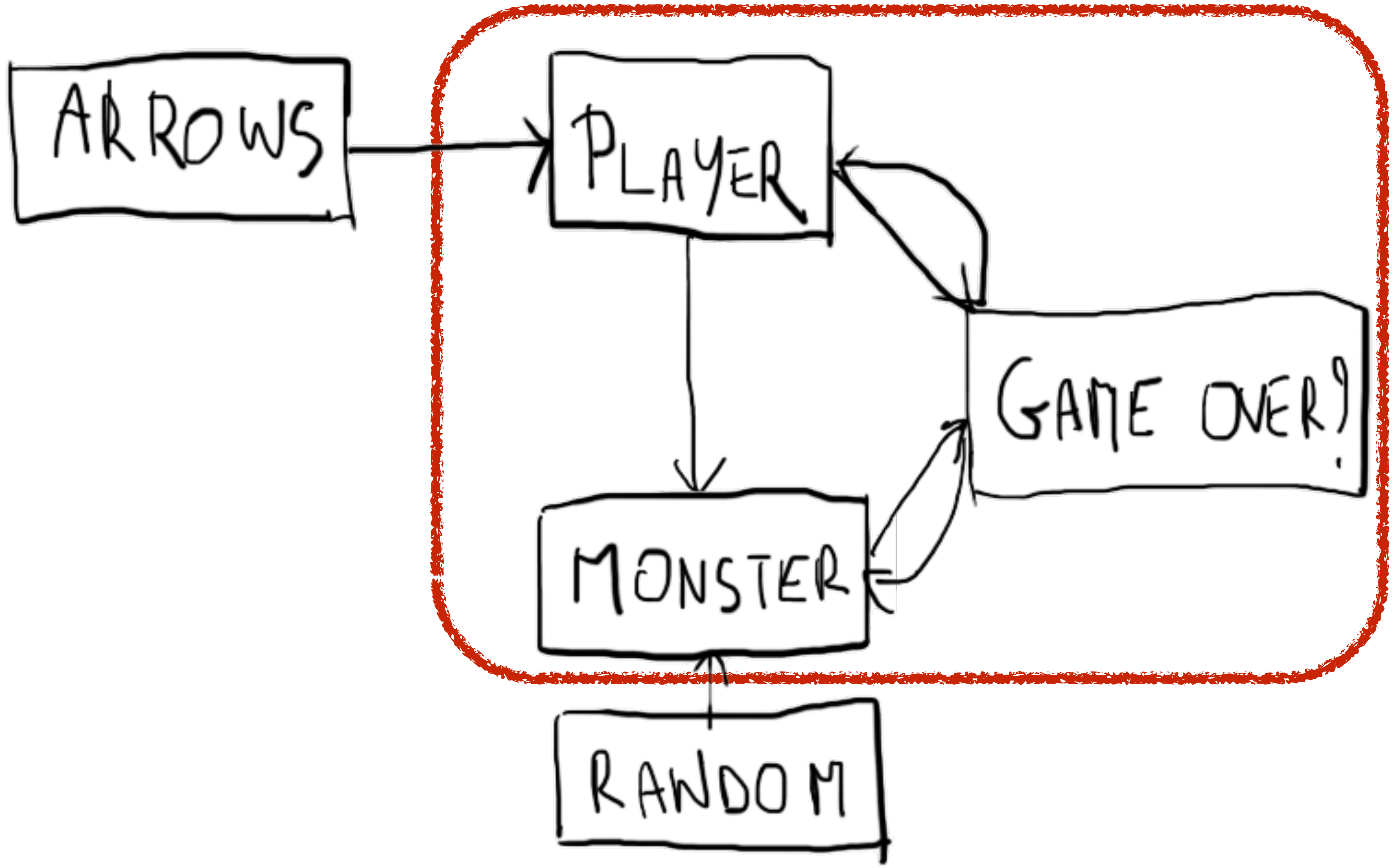
```
start :: SignalGen (Signal a)
      -> IO (IO a)

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
```

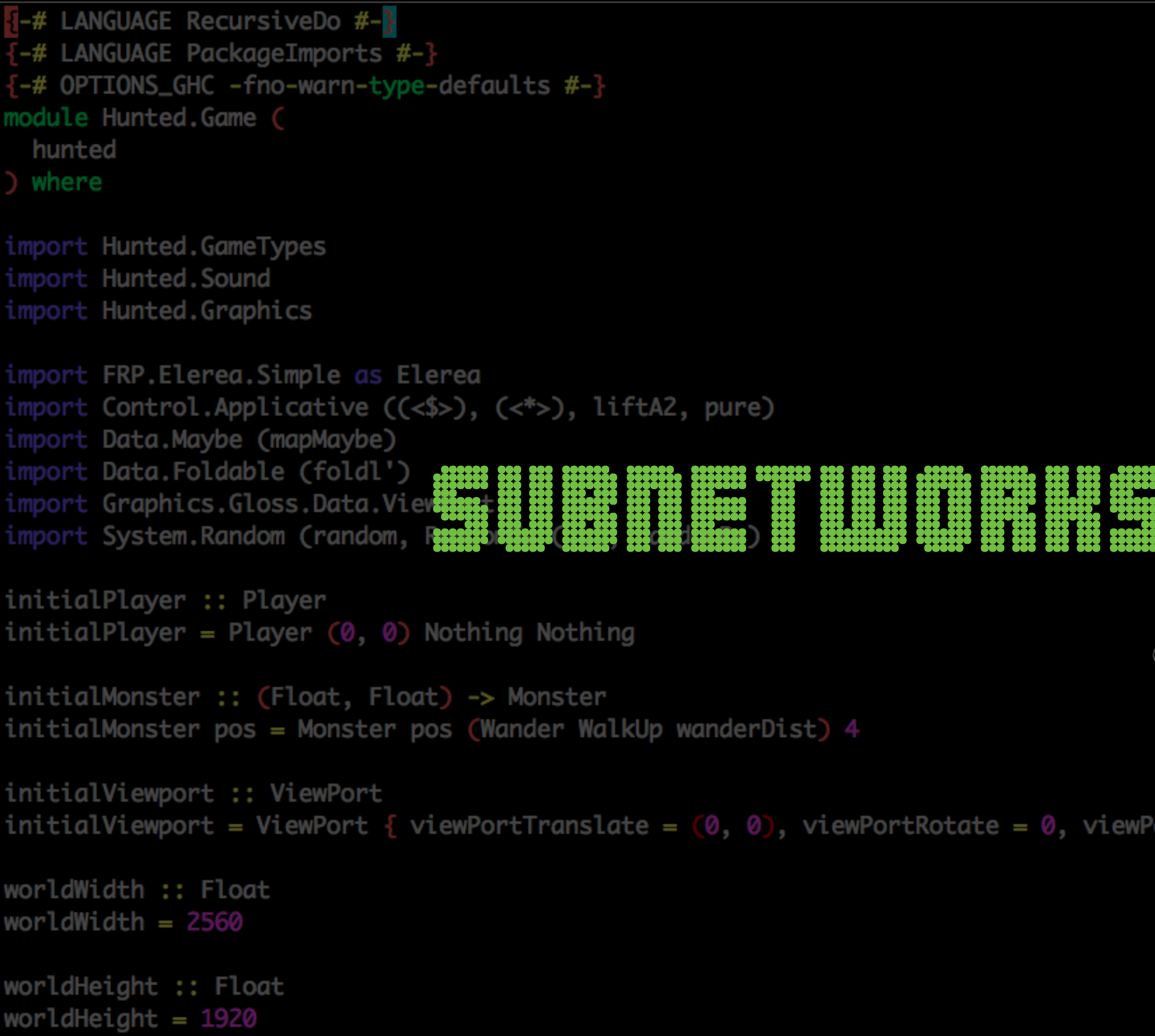
```
player <-
  transfer2 initialPlayer
            movePlayer
            directionKey
            gameOver’

monster <-
  transfer3 initialMonster
            wanderOrHunt
            player
            randomNumber
            gameOver’
```

```
gameState = GameState <$> renderState <*> soundState
```

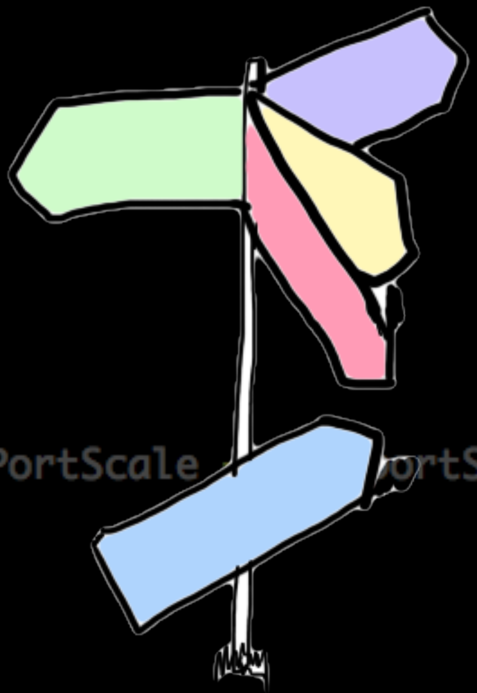

ARROWS
PLAYER
GAME OVER?
MONSTER
RANDOM

```haskell
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
```

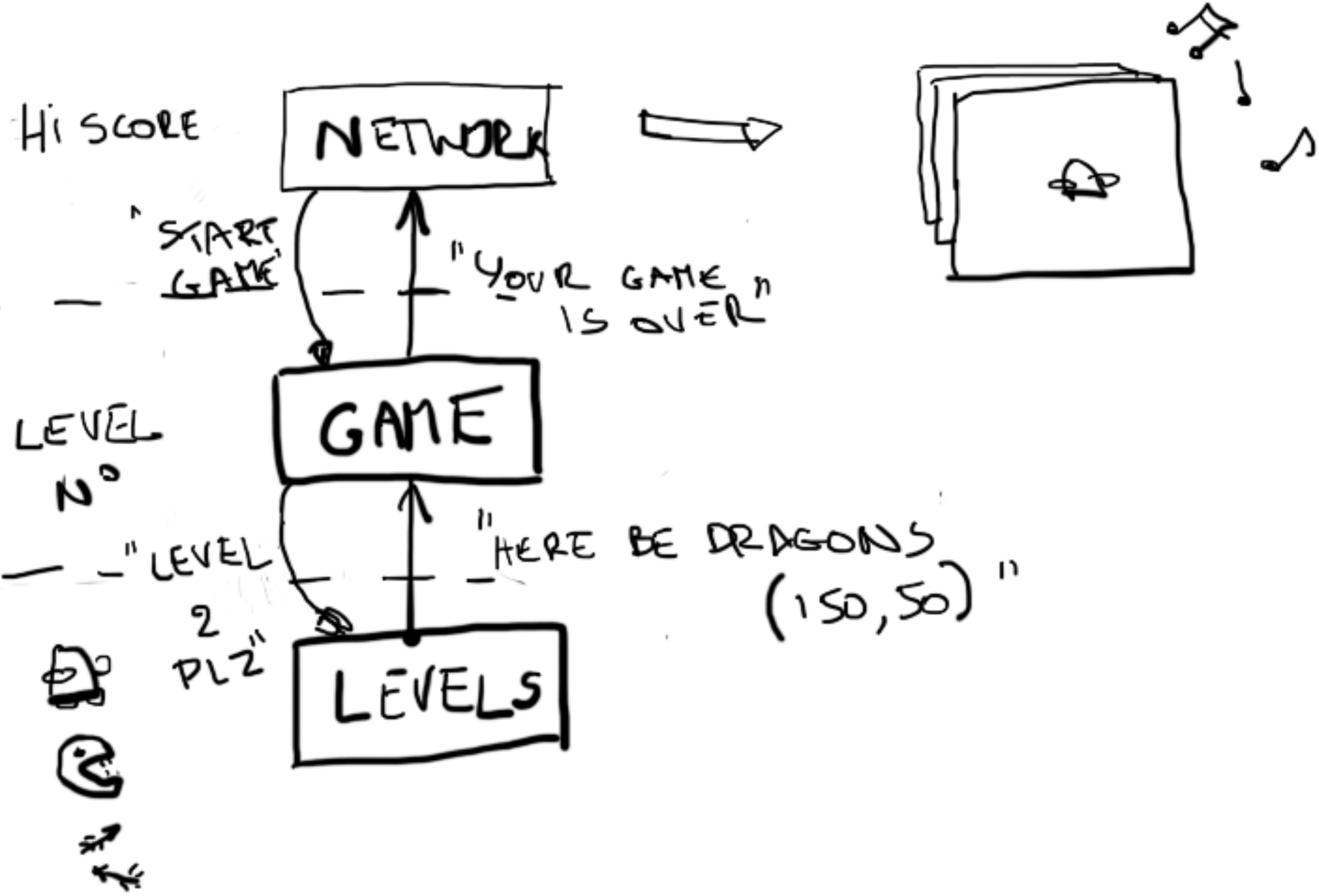
SUBNETWORKS



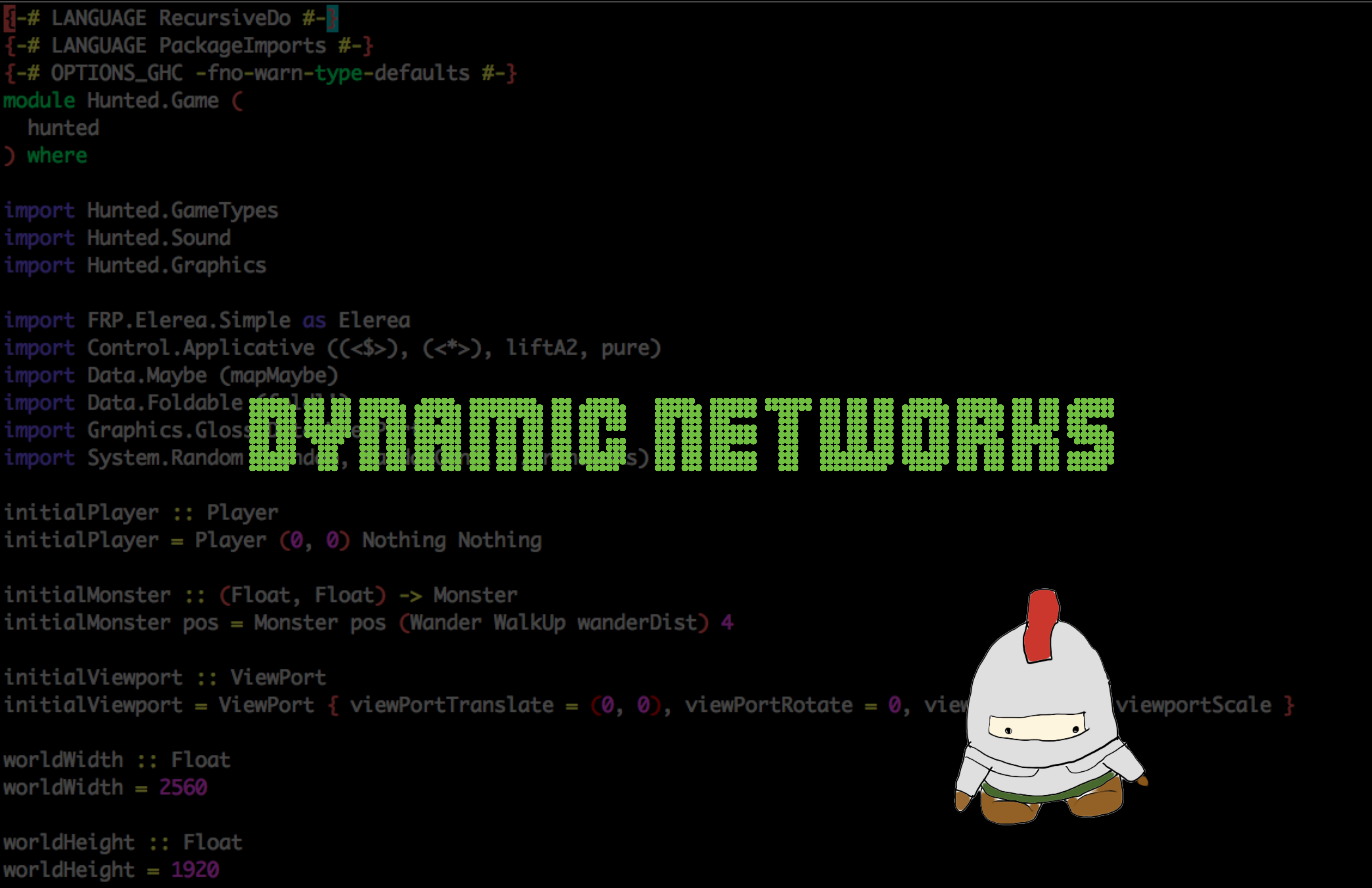
```
{-# LANGUAGE RecursiveDo #-}
{-# LANGUAGE PackageImports #-}
{-# OPTIONS_GHC -fno-warn-type-defaults #-}
module Hunted.Game (
  hunted
) where

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 Graphics.Gloss.Data.View
import System.Random (random, R

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      ortScale }

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"
"YOUR GAME IS OVER"
GAME
LEVEL N°
"LEVEL 2 PLZ"
"HERE BE DRAGONS (150,50)"
LEVELS

DYNAMIC NETWORKS



```haskell
{-# 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 [illegible]
import Graphics.Gloss[illegible]
import System.Random [illegible]s)

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, view[illegible] viewportScale }

worldWidth :: Float
worldWidth = 2560

worldHeight :: Float
worldHeight = 1920
```

Signal [Bolt]

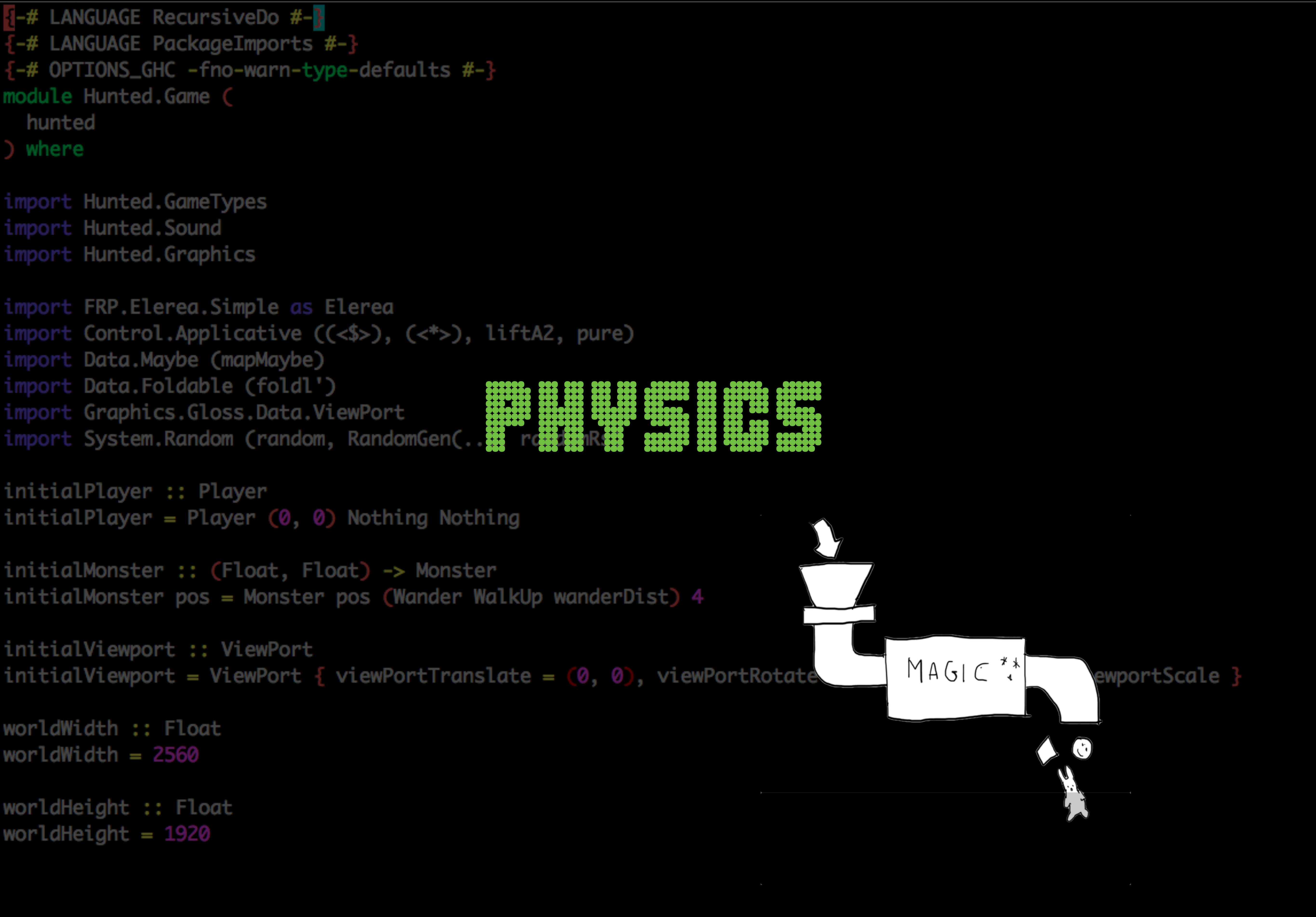
```
bolts <- transfer2 []
                   manageBolts
                   shootKey
                   player
```

SignalGen [Signal Bolt]

```
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)
```

```haskell
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'
```

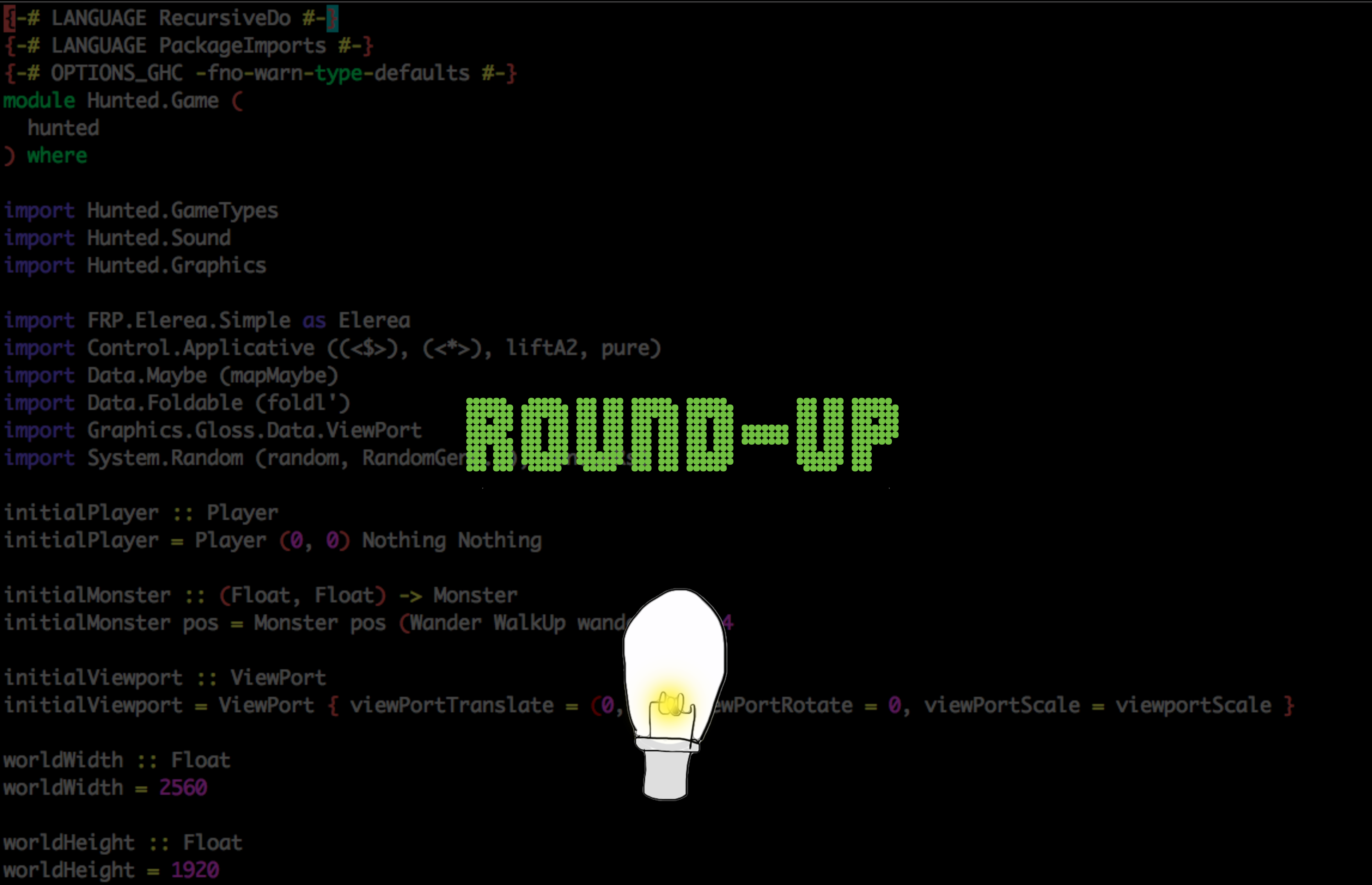
PHYSICS



```
{-# 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 Graphics.Gloss.Data.ViewPort
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   ewportScale }

worldWidth :: Float
worldWidth = 2560

worldHeight :: Float
worldHeight = 1920
```

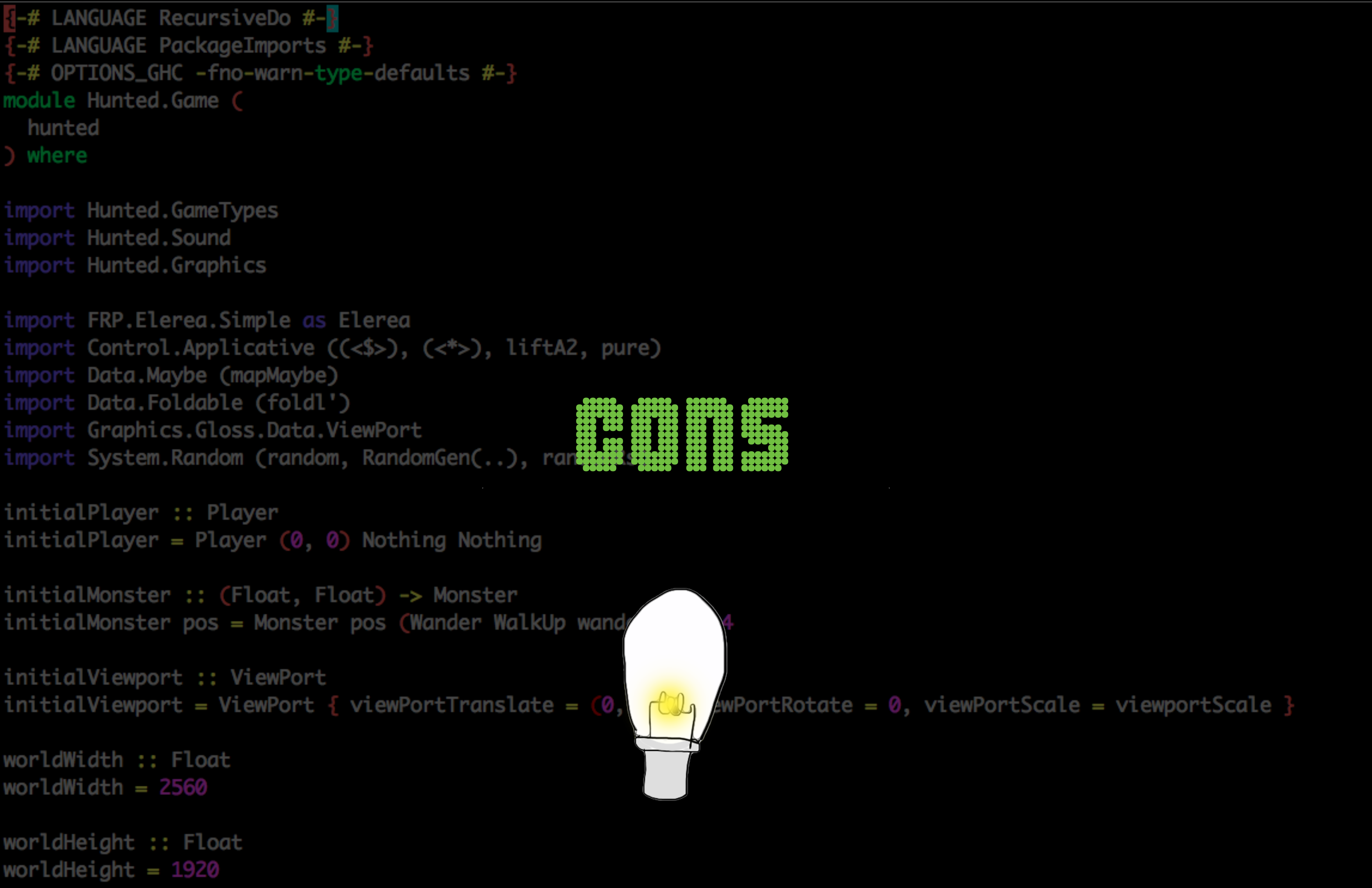
```
execute :: IO a
          -> SignalGen a

effectful :: IO a
          -> SignalGen (Signal a)
```

ROUND-UP



```
{-# 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 Graphics.Gloss.Data.ViewPort
import System.Random (random, RandomGen(..), randoms)

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

initialMonster :: (Float, Float) -> Monster
initialMonster pos = Monster pos (Wander WalkUp wande[illegible] 4

initialViewport :: ViewPort
initialViewport = ViewPort { viewPortTranslate = (0, [illegible]ewPortRotate = 0, viewPortScale = viewportScale }

worldWidth :: Float
worldWidth = 2560

worldHeight :: Float
worldHeight = 1920
```

cons

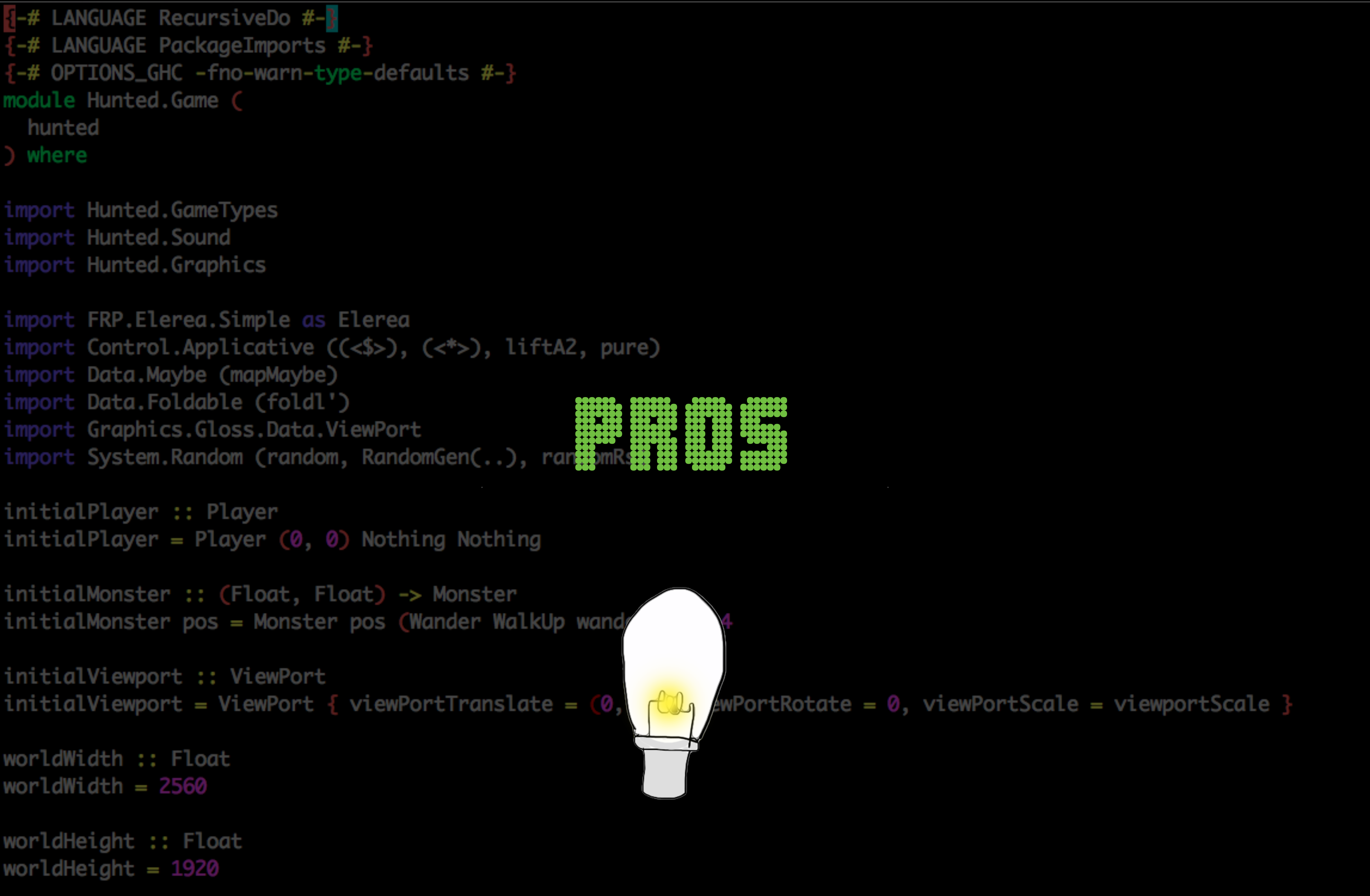


```
{-# LANGUAGE RecursiveDo #-}
{-# LANGUAGE PackageImports #-}
{-# OPTIONS_GHC -fno-warn-type-defaults #-}
module Hunted.Game (
  hunted
) where

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 Graphics.Gloss.Data.ViewPort
import System.Random (random, RandomGen(..), ran

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

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

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

worldWidth :: Float
worldWidth = 2560

worldHeight :: Float
worldHeight = 1920
```

SOME ADDED COMPLEXITY IN HANDLING INFRASTRUCTURE

PERFORMANCE?

PROS



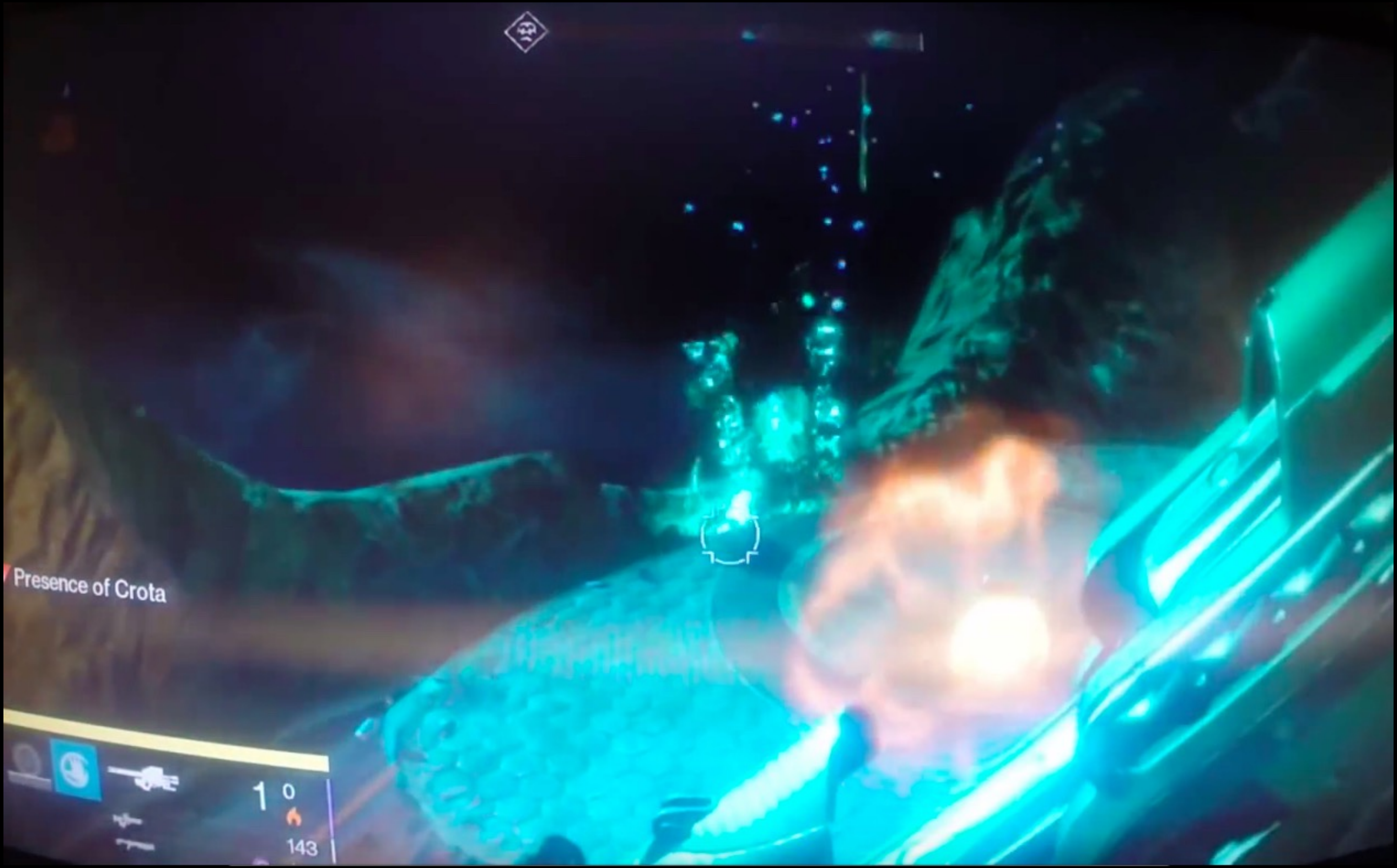
```
{-# LANGUAGE RecursiveDo #-}
{-# LANGUAGE PackageImports #-}
{-# OPTIONS_GHC -fno-warn-type-defaults #-}
module Hunted.Game (
  hunted
) where

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 Graphics.Gloss.Data.ViewPort
import System.Random (random, RandomGen(..), randomRs

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

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

initialViewport :: ViewPort
initialViewport = ViewPort { viewPortTranslate = (0,       ewPortRotate = 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
```


Presence of Crota
1
0
143