Game programming in Haskell

I HAVE A DREAM

FRP?

OpenGL?

Processing/Quil?

Gloss!

Kinds of applications

- **display**
- **animate**
- **simulate**
- **play**

```
display :: Display -> Color -> Picture -> IO ()
```

```
animate :: Display -> Color -> (Float -> Picture) -> IO ()
```

```
simulate :: Display ->
            Color ->
            Int ->
            model ->
            (model -> Picture) ->
            (ViewPort -> Float -> model -> model) ->
            IO ()
```

```
play :: Display ->
        Color ->
        Int ->
        world ->
        (world -> Picture) ->
        (Event -> world -> world) ->
        (Float -> world -> world) ->
        IO ()
```

Game 1 – The Blob

```
blob :: IO ()
blob = play (InWindow "The Blob" (400, 400) (10, 10))
       white 100 startWorld draw onEvent tick

startWorld :: Float
startWorld = 50.0

draw :: Float -> Picture
draw world = color rose $ circleSolid world

onEvent :: Event -> Float -> Float
onEvent _ world = world - 10.0

tick :: Float -> Float -> Float
tick dt world = world + dt * 50.0
```

Game 2 - Armada

```
type Pos = (Float,Float)

startWorld :: [Pos]
startWorld = [(x,y) | x <- [400, 420.. 500], y <- [-250, -230..200]]

drawShip :: Pos -> Picture
drawShip (x,y) = translate x y $ color white $ circleSolid radius
    where radius = 10 * (sin (x * 0.01) + 2)

draw :: [Pos] -> Picture
draw world = pictures $ map drawShip world
```

Composing an update function

```
moveLeft :: Float -> Ship -> Ship
moveLeft dt (x,y) = (x', y)
    where x' = x - dt * speed
          speed = 400 - y

wrap :: Ship -> Ship
wrap (x,y) = (x', y)
    where x' = if x < -510 then 510 else x

curve :: Ship -> Ship
curve (x,y) = (x, y')
    where y' = y + 1.0 * sin (x / 100)

updateShip :: Float -> Ship -> Ship
updateShip dt = wrap . curve . moveLeft dt

tick :: Float -> [Ship] -> [Ship]
tick dt world = map (updateShip dt) world
```

Comparison to Clojure

```clojure
(defn move-left [dt [x y]]
  (let [speed (- 400 y)]
    [(- x (* dt speed)) y]))

(defn wrap [[x y]]
  (if (< x -510)
    [510 y]))

(defn curve [[x y]]
  [x (+ y (Math/sin (* x 0.01)))])

(defn update-ship [ship dt]
  (->> ship
       (move-left dt)
       wrap
       curve))
```

Will crash - bugfix!

```
(defn wrap [[x y]]
  (if (< x -510)
    [510 y]
    [x y]))
```

Game 3 - Shnake

```haskell
type Pos = (Int, Int)

data Dir = N | E | S | W deriving (Show, Eq)

data World = World {
    randg :: StdGen,
    snake :: Snake,
    cherry :: Maybe Pos,
    time :: Float
} deriving Show

data Snake = Snake {
    body :: [Pos],
    dir :: Dir,
    moveTimer :: Float,
    alive :: Bool
} deriving Show
```

Setting up the world

```
makeWorld :: StdGen -> World
makeWorld g = World {
    randg = g,
    snake = makeSnake,
    cherry = Nothing,
    time = 0.0
}

makeSnake :: Snake
makeSnake = Snake {
    body = [(x,10) | x <- [15..17]],
    dir = W,
    moveTimer = 0.0,
    alive = True
}
```

Rendering (excerpt)

```
drawSnake :: Snake -> Picture
drawSnake s = color c $ pictures $ map drawSegment $ body s
    where c = if alive s then black else makeColor 0.8 0.8 0.7 1.0

drawSegment :: Pos -> Picture
drawSegment pos = drawAtPos pos $ rectangleSolid tileSize tileSize
```

Update loop

```
tick :: Float -> World -> World
tick dt = maybeCreateCherry . maybeEatCherry . tickSnakePart dt . increaseWorldTime dt

tickSnakePart :: Float -> World -> World
tickSnakePart dt w = w { snake = updateSnake dt (snake w) }

updateSnake :: Float -> Snake -> Snake
updateSnake dt = checkSelfCollision . checkBounds . maybeMove . increaseMoveTimer dt
```

Randomness?

```haskell
maybeCreateCherry :: World -> World
maybeCreateCherry world =
    case cherry world of
        (Just _) -> world -- cherry exists
        Nothing -> createCherryAtRandomPosition world
```

Randomness!

```
createCherryAtRandomPosition :: World -> World
createCherryAtRandomPosition = execState place
    where place = do randPos <- getRandomPosition
                     modify (addCherry randPos)

addCherry :: Pos -> World -> World
addCherry pos world = world { cherry = Just pos }

getRandomPosition :: State World Pos
getRandomPosition = do rx <- getRand (0, boardSize)
                       ry <- getRand (0, boardSize)
                       return (rx,ry)
```

Input

```
onEvent :: Event -> World -> World
onEvent (EventKey (SpecialKey KeySpace) Down _ _) world = restart world
onEvent (EventKey (SpecialKey key) Down _ _) world = controlSnake world key
onEvent _ world = world

...

controlDir :: SpecialKey -> Snake -> Dir
controlDir key s =
    let currentDir = dir s
        (Just opposite) = lookup currentDir opposites
        desiredDir = case key of
                        KeyUp -> N
                        KeyRight -> E
                        KeyDown -> S
                        KeyLeft -> W
    in if desiredDir == opposite then currentDir else desiredDir
```

Restarting

```
restart :: World -> World
restart world = makeWorld (randg world)
```

Game 3 - Lentals

Lots of boilerplate...

```
data Dot = Dot {
      _pos :: Point,
      _col :: Color,
      _rad :: Float
}

flipColor :: Dot -> Dot
flipColor dot = dot { _col = invertColor (_col dot) }
```

Lens!

```
makeLenses ''Dot

flipColor :: Dot -> Dot
flipColor = over col invertColor
```

True happiness

```
onEvent (EventKey (SpecialKey KeySpace) Down _ _) world =
    over dots (map flipColor) world

drawDot :: Dot -> Picture
drawDot dot = translateUsingPoint (dot^.pos)
            $ color (dot^.col)
            $ circleSolid (dot^.rad)
```

Quil vs Gloss

- bugs, correctness
- interactive development
- adapting to changing requirements
- docs, tutorial and support
- limitations

The Haskell School of Expression

by Paul Hudak

#HASKELL-GAME

John Carmack's keynote at Quakecon 2013 part 4

https://github.com/eriksvedang/Gamelecture