2D animations in Haskell using gloss, lens and state

Julien Dehos

Fosdem 2023

Who am I?

- assistant professor in Computer Science at ULCO, France
- using Haskell since 2015 (for teaching FP + small projects)

Animations with Haskell

- library bindings: sdl2...
- Entity-Component-System: apecs...
- Functional Reactive Programming: yampa, reactive-banana...
- some cool projects:
  - Defect Process (2d hack n' slash game)
  - reanimate (Haskell library for building declarative animations)
  - ...

In this talk

- implement several animations using functional programming
- improve the code using some Haskell features/libraries:
  - algebraic data types
  - lazy evaluation
  - lens
  - state

First example: draw a solid circle

- using the gloss library:
  - 2D vector graphics + animations
  - provides: functions (graphics, events...) + main-loops
  - we just have to write some handler functions

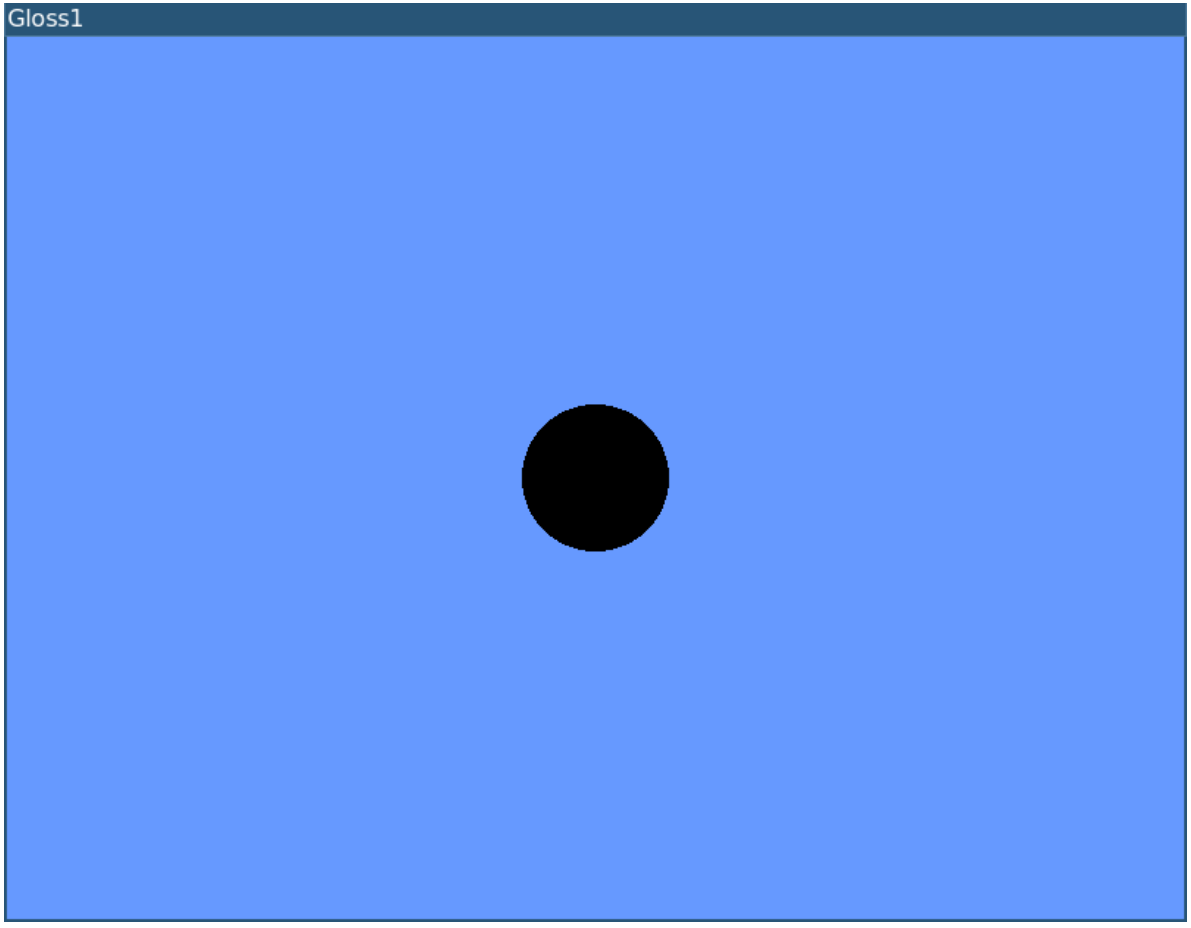
- handler functions:

```
type Model = ()

handleDisplay :: Model -> Picture
handleDisplay model = circleSolid 50

handleEvent :: Event -> Model -> Model
handleEvent event model = model

handleTime :: Float -> Model -> Model
handleTime deltaTime model = model
```

- main function:

```
winWidth, winHeight :: Int
winWidth = 800
winHeight = 600

main :: IO ()
main = do
    let model = ()
        window = InWindow "Gloss0" (winWidth, winHeight) (0, 0)
        bgcolor = makeColor 0.4 0.6 1.0 1.0
        fps = 30
    play window bgcolor fps model handleDisplay handleEvent handleTime
        -- run main loop
```

- result:



Random radius (1)

- using a pseudo-random number generator:

```
data Model = Model
    { _radius :: Float   -- current radius
    , _gen :: StdGen     -- current generator
    }

handleDisplay :: Model -> Picture
handleDisplay model = circleSolid (_radius model)

handleTime :: Float -> Model -> Model
handleTime deltaTime (Model _ gen) =
    let (radius', gen') = randomR (20, 50) gen
        -- generate a new radius
    in Model radius' gen'
```

```haskell
main :: IO ()
main = do
    (radius, gen) <- randomR (20, 50) <$> getStdGen
        -- get a generator and generate the first radius
    let model = Model radius gen
        window = InWindow "Gloss1" (winWidth, winHeight) (0, 0)
        bgcolor = makeColor 0.4 0.6 1.0 1.0
        fps = 1
    play window bgcolor fps model handleDisplay handleEvent handleTime
```

Random radius (2)

- using infinite lists (non-strict evaluation):

```
data Model = Model
    { _radius :: Float
    , _nextRadii :: [Float]
          -- to be initialised with an infinite list
    }

handleTime :: Float -> Model -> Model
handleTime deltaTime (Model _ rs) = Model (head rs) (tail rs)
    -- update current radius by consuming the infinite list
```

```haskell
main :: IO ()
main = do
    (r : rs) <- randomRs (20, 50) <$> getStdGen
        -- get an infinite list of pseudo-random radiuses
    let model = Model r rs
        window = InWindow "Gloss2" (winWidth, winHeight) (0, 0)
        bgcolor = makeColor 0.4 0.6 1.0 1.0
        fps = 1
    play window bgcolor fps model handleDisplay handleEvent handleTime
```

Second example: a bouncing ball

- let's define some types representing our data:

```
data Ball = Ball
    { _pos :: V2 Float
    , _vel :: V2 Float
    }

data Model = Model
    { _ball :: Ball
    , _nextBalls :: [Ball]
        -- infinite list of pseudo-random balls
    }
```

- access/update types using pattern matching or record syntax:

```
handleTime :: Float -> Model -> Model
handleTime deltaTime model =
    let ball1 = updateMotion deltaTime $ _ball model
        ball2 = updateBounces ball1
    in model { _ball = ball2 }
        -- update model using record syntax

updateMotion :: Float -> Ball -> Ball
...
```

```haskell
updateBounces :: Ball -> Ball
updateBounces ball0 = ball4
    where
        (V2 px py) = _pos ball0
            -- pattern match _pos
        (V2 vx vy) = _vel ball0
        ball1 = if xMin >= px
                    then Ball (V2 (2*xMin - px) py) (V2 (-vx) vy)
                        -- construct an updated Ball
                    else ball0
        ball2 = ...
```

Lens

- accessing/updating nested types can be cumbersome
- lenses can simplify that (or not):
  - construct lenses
  - use some functions/operators to access/update data

```haskell
makeLenses ''Ball
makeLenses ''Model
  -- construct lenses for our types

handleTime :: Float -> Model -> Model
handleTime deltaTime model =
    model & ball %~ updateMotion deltaTime
          & ball %~ updateBounces
      -- update model after applying two functions on _ball
```

```haskell
updateBounces :: Ball -> Ball
updateBounces ball0 = ball4
    where
        (V2 x y) = ball0 ^. pos
                        -- access to _pos
        ball1 = if xMin >= x
                    then ball0 & pos . _x .~ 2*xMin - x
                                            -- set a value
                               & vel . _x %~ negate
                                            -- apply a function
                    else ball0
        ball2 = ...
```

State

- a well-known monad in Haskell
- enables us to implement actions which access/modify a state
- stateful functions/operators in the lens library

```haskell
handleTime :: Float -> Model -> Model
handleTime deltaTime model =
    let updateActions = do updateMotion deltaTime
                           updateBounces
            -- define a state action
    in model & ball %~ execState updateActions
                    -- execute the action on _ball

updateMotion :: Float -> State Ball ()
...
    -- action that has a state Ball and produces a result ()
```

```haskell
updateBounces :: State Ball ()
updateBounces = do
    (V2 x y) <- use pos
                -- access to the field _pos of the current state
    when (xMin >= x) $ do
        pos . _x .= 2*xMin - x
                -- set a value for the current state
        vel . _x %= negate
                -- apply a function on the current state
    when ...
```

Conclusion

- using FP/Haskell, we can easily:
  - implement animations (gloss)
  - use infinite lists (lazy evaluation)
  - access/modify nested types (lens)
  - simulate a mutable state (State monad)
- based on pure functions + static typing:
  - easy to read
  - less error-prone

References

- slides & code:
  https://gitlab.com/juliendehos/talk-2023-fosdem
- gloss: https://hackage.haskell.org/package/gloss
- lens: https://hackage.haskell.org/package/lens
- state: https://wiki.haskell.org/State_Monad

Thank you! Questions/discussion?