lambdor.net Devblog

Gerold Meisinger

Oct 20, 2021

CONTENTS:

This was a series of blog post articles back in 2010-2012 written about Haskell, Yampa and game development. The original Wordpress blog lambdor.net (archived) eventually got corrupted, so some of the articles are revived here on GitLab and Read The Docs. I won't restore any other articles because I think they are no longer relevant. If you like to restore them I'm happy to include a pull request.

If you need more up to date tutorials on FRP and Yampa please check out my new book.

CHAPTER

ONE

LEARNING YAMPA AND FUNCTIONAL REACTIVE PROGRAMMING

This blog post was originally written back in 2010-06-13

If you are just starting to learn Yampa, please share your experiences! This post just covers my personal opinion on how to tackle Yampa, maybe you have additional or different recommendations.

1.1 Recommendations for learning

I recommend reading the following papers/presentations to learn Yampa and FRP in general:

- A Brief Introduction to Functional Reactive Programming and Yampa (slides)
- Arrows, FRP, and Functional Reactive Programming (PPT)
- Arrows, Robots and Functional Programming: covers Yampa basics in detail (Section 3 is very domain specific and may be omitted)
- Functional Reactive Programming, Continued: more Yampa basics
- The Yampa Arcade: standard paper for games
- Dynamic, Interactive Virtual Environments: read chapter 3 – Time and appendix A – Functional Reactive Programming
- Functional Programming and 3D Games: Yampa basics in games, not very detailed though
- Functional Reactive Programming from First Principles: Yampa implementation details
- Dynamic Optimization for Functional Reactive Programming: Yampa optimization details

1.2 Understanding FRP

I think to learn FRP (for games) you have to especially understand the following aspects:

- Signals make time omnipresent
- Systems are built with **Signal Functions** (`SF a b`)
- FRP is implemented in standard Haskell
- **Arrow notation** makes using FRP convenient and more readable
- Signal functions **diagrams** look just mirrored to the actual arrow notation code :)
- The signal function systems need to be updated somehow – usually via `reactimate`
- `reactimate` divides the programm into input IO (sense), the signal function (`SF`) and output IO (actuate)

- Switches allow dynamic changes of the reactive system. Note that in Yampa signal functions are continuation-based so they "switch into" a new signal function.
- To handle dynamic game object collections use the **delayed parallel switch** (`dpSwitch`) signal function
- **Input events** are propagated to the objects via `route`
- `route` also reasons about the whole object collection to produce **logical events** (f.e. hit detection)
- `killOrSpawn` collects all kill and spawn events into **one big function composition** (insertion/deletion) which is applied to the object collection
- In the Space Invaders example `gameCore :: IL Object -> SF (GameInput, IL ObjOutput) (IL ObjOutput)` is actually embedded in the function `core :: ... -> SF GameInput (IL ObjOutput)` which acts as the intermediate between sense and actuate (this is not mentioned in the Yampa Arcade paper)

1.3 Complete list of recommended papers

Covering FRP in general and FRP in games:

- A Brief Introduction to Functional Reactive Programming and Yampa (slides)
- Arrows, FRP, and Functional Reactive Programming (PPT)
- Arrows, Robots and Functional Programming
- Directions in Functional Programming for RealTime Applications
- Dynamic, Interactive Virtual Environments
- Dynamic Optimization for Functional Reactive Programming
- Functional Programming and 3D Games
- Functional Reactive Programming from First Principles
- Functional Reactive Programming, Continued
- Plugging A Space Leak With An Arrow
- Push-Pull Functional Reactive Programming
- Push-Pull Functional Reactive Programming (video)
- The Yampa Arcade

1.4 List of discarded papers

The reason for discarding was mostly because they are too old, too theoretical or off-topic from games:

- A Functional Reactive Animation Of A Lift Using Fran
- A Language for Declarative Robotic Programming
- Crafting Game-Models Using Reactive System Design
- Event-Driven FRP
- FrTime – A Language for Reactive Programs
- Functional Reactive Animation
- Functional Reactive Programming for Real-Time Reactive Systems

- Genuinely Functional User Interfaces
- Interactive Functional Objects in Clean
- Modelling Reactive Multimedia – Events and Behaviours
- Modular Domain Specific Languages and Tools
- Prototyping Real-Time Vision Systems
- Reactive Multimedia Documents in a Functional Framework
- Real-Time FRP

CHAPTER

TWO

RUNNING PRIMITIVE SIGNAL FUNCTIONS IN YAMPA

This blog post was originally written back in 2010-06-12

To test signal functions in Yampa, use the `embed` function. Enter the following commands in the Haskell command-line to show the header definition:

```
>>> :type FRP.Yampa.embed
-- FRP.Yampa.embed :: FRP.Yampa.SF a b -> (a, [(FRP.Yampa.DTime, Maybe a)]) -> [b]
```

So the parameters are:

1. the signal function to run
2. a tuple of…
   1. the first input value at time=0
   2. and a list of… `(time, Nothing|Just nextValue)`

and return a list of values produced by the signal function.

Primitive signal functions include: `time`, `identity` and `constant`

```
main :: IO ()
main = do
    putStrLn $ show $ embed time (Nothing, [(1.0, Nothing), (0.2, Nothing), (0.03,␣
↪Nothing)])
    putStrLn $ show $ embed time (123, [(1.0, Just 234), (0.2, Just 345), (0.03, Just␣
↪456)])
    -- [0.0,1.0,1.2,1.23]
    -- [0.0,1.0,1.2,1.23]

    putStrLn $ show $ embed identity (123, [(1.0, Just 234), (0.2, Just 345), (0.03,␣
↪Just 456)])
    putStrLn $ show $ embed identity (537, [(1.0, Nothing), (0.2, Nothing), (0.03, Just␣
↪123)])
    -- [123,234,345,456]
    -- [537,537,537,537]

    putStrLn $ show $ embed (constant 537) (Nothing, [(1.0, Nothing), (0.2, Nothing), (0.
↪03, Nothing)])
    putStrLn $ show $ embed (constant 537) (123, [(1.0, Just 234), (0.2, Just 345), (0.
↪03, Just 456)])
    -- putStrLn $ show (embed constant (123, [(1.0, Just 234), (0.2, Just 345), (0.03,␣
↪Just 456)])) -- ERROR
```

(continues on next page)

(continued from previous page)

```
-- [537,537,537,537]
-- [537,537,537,537]
```

CHAPTER

THREE

YAMPA/SDL PROGRAM STUB

This blog post was originally written back in 2010-06-14

I just completed my first Yampa/SDL program stub. This stub is meant to provide a quickstart for using Yampa with SDL and explains the basic Yampa functions needed for game development in the most minimalistic way I could think of. You can also download the `whole source file`. The "game" basically is a player object (black square) which can move around on a 3x3 field and an obstacle object (blue square) which gets killed on collision.

To get an overview of Yampa reactimate have a look at the diagrams of my 2 recent posts *Activity diagram of Yampa reactimate* and *Dataflow diagram of Yampa reactimate*.

3.1 Definitions

At first we are defining some types:

- `Input`: **non-deterministic** events from input devices which have to come from an IO task.
- `Logic`: **deterministic** events from object preprocessor in `route`.
- `ObjEvents`: `Input` and `Logic` bundled together
- `State`: the logical object states (**position**, velocity etc.) produced after each step which are used for **collision detection and rendering**.
- `ObjOutput`: the overall object output consisting of `State` and the produced **kill- and respawn requests**.
- `ObjOutput`: just an abstract **signal function** type which takes the events and produces an output.

```
module Main where

import IdentityList

import Maybe
import Control.Monad.Loops

import FRP.Yampa              as Yampa
import FRP.Yampa.Geometry
```

(continues on next page)

(continued from previous page)

```
import Graphics.UI.SDL        as SDL
import Graphics.UI.SDL.Events as SDL.Events
import Graphics.UI.SDL.Keysym as SDL.Keysym

type Position2 = Point2  Double
type Velocity2 = Vector2 Double

type Input = [SDL.Event]    -- non-deterministic events from input devices
type Logic = Yampa.Event () -- deterministic events from object processor

data ObjEvents = ObjEvents
    { oeInput :: Input
    , oeLogic :: Logic
    } deriving (Show)

data State = Rectangle Position2 SDL.Rect SDL.Pixel | Debug String
     deriving (Show)

data ObjOutput = ObjOutput
    { ooState         :: State
    , ooKillRequest   :: Yampa.Event ()       -- NoEvent|Event ()
    , ooSpawnRequests :: Yampa.Event [Object]
    }

defaultObjOutput = ObjOutput
    { ooState         = undefined
    , ooKillRequest   = Yampa.NoEvent
    , ooSpawnRequests = Yampa.NoEvent
    }

type Object = SF ObjEvents ObjOutput

instance (Show a) => Show (Yampa.Event a) where
    show (Yampa.Event a) = "LogicEvent: " ++ (show a)
    show Yampa.NoEvent   = "NoEvent"

instance Show (SF a b) where
    show sf = "SF"
```

`IdentityList` is taken from the Yampa SpaceInvaders example which you can get via:

```
>>> cabal unpack spaceinvaders
```

3.2 Main

Don't get scared by the long definition, it mostly consists of object bindings. I split `main` into 2 definitions which can be run seperately by uncommenting them (line 4-5). `mainLoop` runs the **complete game**: move via [Arrow] keys and quit with [Esc]. `mainSteps` runs each **step individually** and in **isolation** which should help to understand what is going on and how the types are passed around and transformed. The steps are commented in the source, try to understand them by reading the highlighted lines, the object bindings and the output they produce!

```
main :: IO ()
main = do
    -- Uncomment 'mainSteps' or 'mainLoop'!
    --mainLoop  -- Runs the complete reactimate loop.
    --mainSteps -- Tests each reactimate step individually.
  where
    mainLoop :: IO ()
    mainLoop = do
        reactimate initialize input output (process objs)
        SDL.quit
       where
         playerObj   = playerObject (Point2 16 16)
                                    (SDL.Rect (-8) (-8) 8 8)
                                    (SDL.Pixel 0x00000000)
         obstacleObj = staticObject (Point2 48 48)
                                    (SDL.Rect (-8) (-8) 8 8)
                                    (SDL.Pixel 0x000000FF)
         objs = (listToIL [playerObj, obstacleObj])

    mainSteps :: IO ()
    mainSteps = do
        -- initialize :: IO Input
        -- Poll first 'SDL.Event's (should only be 'LostFocus').
        events <- initialize

        -- input :: IO (DTime, Maybe Input)
        -- Poll 'SDL.Event's at each step (probably []).
        events <- input False

        -- hits :: [(ILKey, State)] -> [ILKey]
        -- Testing player over obstacle => collision event.
        putStrLn $ "hits 1: " ++ (show $ hits $ assocsIL $ fmap ooState oos1)

        -- Testing player over enemy => no event.
        putStrLn $ "hits 2: " ++ (show $ hits $ assocsIL $ fmap ooState oos2)

        -- route :: (Input, IL ObjOutput) -> IL sf -> IL (ObjEvents, sf)
        -- Routes 'key' SDL.Event to all 'Object's and
        -- previous object 'State's, if there are any.

        -- First routing step.
        -- No collision events are checked as there are no 'State's yet.
        putStrLn "first route: "
        --mapM putStrLn $ showILObjEvents $ route ([key], emptyIL) objs
        putStrLn $ show $ assocsIL $ route ([key], emptyIL) objs
```

(continues on next page)

(continued from previous page)

```
        -- Intermediate routing step.
        -- Assuming player over obstacle object => create collision event.
        putStrLn "route step: "
        putStrLn $ show $ assocsIL $ route ([key], oos1) objs

        -- killAndSpawn :: (Input, IL ObjOutput)
        --              -> (Yampa.Event (IL Object -> IL Object))
        -- Kill and spawn new objects corresponding to 'ObjOutput' requests.
    -- Note how 'ooObstacle' defined a kill and spawn request
        putStr "objs before kill&Spawn: "
        putStrLn $ show $ keysIL objs
        putStr "objs after kill&Spawn: "
        putStrLn $ show $ keysIL $
            case (killAndSpawn (([], emptyIL), oos1)) of
                (Event d) -> d objs
                _         -> objs

        -- output :: IL ObjOutput -> IO Bool
        -- Just render the 'State's or quit if there is none.
        o1 <- output False oos1
        putStrLn $ show o1
        o2 <- output False oos2
        putStrLn $ show o2
        o3 <- output False emptyIL
        putStrLn $ show o3

        SDL.quit
      where
        key = KeyDown (Keysym
            { symKey = SDL.SDLK_RIGHT
            , symModifiers = []
            , symUnicode = '\0'
            })
        playerObj   = playerObject (Point2 16 16)
                                   (SDL.Rect (-8) (-8) 8 8)
                                   (SDL.Pixel 0x00000000)
        obstacleObj = staticObject (Point2 48 48)
                                   (SDL.Rect (-8) (-8) 8 8)
                                   (SDL.Pixel 0x000000FF)
        objs = (listToIL [playerObj, obstacleObj])

        enemyObj = staticObject (Point2 80 80)
                                (SDL.Rect (-8) (-8) 8 8)
                                (SDL.Pixel 0x00FF0000)
        ooPlayer = defaultObjOutput
            { ooState = Rectangle (Point2 48 48)
                                  (SDL.Rect (-8) (-8) 8 8)
                                  (SDL.Pixel 0x00000000)
            }
        ooObstacle = defaultObjOutput
            { ooState = Rectangle (Point2 48 48)
```

(continues on next page)

(continued from previous page)

```
                                (SDL.Rect (-8) (-8) 8 8)
                                (SDL.Pixel 0x000000FF)
            , ooKillRequest   = Event ()
            , ooSpawnRequests = Event [enemyObj]
            }
        ooEnemy = defaultObjOutput
            { ooState = Rectangle (Point2 80 80)
                                (SDL.Rect (-8) (-8) 8 8)
                                (SDL.Pixel 0x00FF0000)
            }
        oos1 = listToIL [ooPlayer, ooObstacle]
        oos2 = listToIL [ooPlayer, ooEnemy]
```

3.3 Output from mainSteps

... slightly modified for better readability.

0 = playerObject, 1 = obstacleObject, 2 = enemyObject

```
initialize (sense): [LostFocus [MouseFocus]]
input (sense): []

hits 1: [1,0]
hits 2: []

first route:
[(1, (ObjEvents { oeInput = [KeyDown (Keysym { symKey = SDLK_RIGHT, ... })]
                , oeLogic = NoEvent
                }, SF)),
 (0, (ObjEvents {oeInput = [KeyDown (Keysym { symKey = SDLK_RIGHT, ... })],
                , oeLogic = NoEvent
                }, SF))]

route step:
[(1, (ObjEvents { oeInput = [KeyDown (Keysym { symKey = SDLK_RIGHT, ... })]
                , oeLogic = LogicEvent: ()
                }, SF)),
 (0, (ObjEvents {oeInput = [KeyDown (Keysym { symKey = SDLK_RIGHT, ... })]
                , oeLogic = LogicEvent: ()
                }, SF))]

objs before kill&Spawn: [1,0]
objs after  kill&Spawn: [2,0]

output (actuate) + 500ms delay: False
output (actuate) + 500ms delay: False
output (actuate) + 500ms delay: True
```

3.4 Reactimation IO (sense and actuate)

The IO steps are very simple. `initialize` and `input` just collect the input events (line 10, 22) and `output` defines the rendering to draw a rectangle or print a debug string and maps over the object output states to draw them.

```
initialize :: IO Input
initialize = do
    SDL.init [SDL.InitVideo]
    screen <- SDL.setVideoMode windowWidth windowHeight
                               windowDepth [SDL.HWSurface]
    SDL.setCaption windowCaption []

    SDL.fillRect screen Nothing (SDL.Pixel 0x006495ED) -- 0x00RRGGBB
    SDL.flip screen
    events <- unfoldWhileM (/= SDL.NoEvent) SDL.pollEvent

    putStrLn $ "initialize (sense): " ++ show events
    return events
  where
    windowWidth   = 96
    windowHeight  = 96
    windowDepth   = 32
    windowCaption = "Yampa/SDL Stub"

input :: Bool -> IO (DTime, Maybe Input)
input _ = do
    events <- unfoldWhileM (/= SDL.NoEvent) SDL.pollEvent
    putStrLn $ "input (sense): " ++ show events
    return (1.0, Just events)

output :: Bool -> IL ObjOutput -> IO Bool
output _ oos = do
    putStrLn $ "output (actuate) + " ++ (show delayMs) ++ "ms delay: "

    screen <- SDL.getVideoSurface
    SDL.fillRect screen Nothing (SDL.Pixel 0x006495ED) -- Pixel 0x--RRGGBB

    mapM_ (\oo -> render (ooState oo) screen) (elemsIL oos) -- render 'State'!

    SDL.flip screen
    SDL.delay delayMs

    return $ null $ keysIL oos
  where
    delayMs = 500

    render :: State -> SDL.Surface -> IO ()
    render (Rectangle pos rect color) screen = do
        SDL.fillRect screen gRect color
        return ()
      where
        -- center rectangle around position
        x0 = round (point2X pos) + (rectX rect)
```

(continues on next page)

(continued from previous page)

```
        y0 = round (point2Y pos) + (rectY rect)
        x1 = round (point2X pos) + (rectW rect)
        y1 = round (point2Y pos) + (rectH rect)
        gRect = Just (SDL.Rect x0 y0 (x1 - x0) (y1 - y0))
    render (Debug s) screen = putStrLn s
```

3.5 Reactimation process (SF)

This is the most important step in `reactimate` (in -> SF in out -> out) and **took me a while to understand**. Again, try to get an overview first with the *Activity diagram* and *Dataflow diagram*!

`process` actually just wraps the `core` to be consistent with the `reactimate` signature and also feeds the previous output states back into `core`. The last expression is very interesting as it applies a **list of insertIL and deleteIL functions** (which are composited together in `killAndSpawn`) to the object list and switches into the **new core**. We can say the `core` is valid as long as the same objects exist.

```
process :: IL Object -> SF Input (IL ObjOutput)
process objs0 = proc input -> do
    rec
        -- 'process' stores the 'State's (note: rec) and
        -- passes them over to core
        oos <- core objs0 -< (input, oos)
    returnA -< oos
```

Note that `core` actually takes Input AND the previous object states (`IL ObjOutput`) as input signals. The `dpSwitch` is performed on a **SF collection** (hence parallel and the `p`) and the result is **observable** and **applied at the next step** (hence delayed and the `d`).

```
core :: IL Object -> SF (Input, IL ObjOutput) (IL ObjOutput)
core objs = dpSwitch route
                     objs
                     (arr killAndSpawn >>> notYet)
                     (\sfs' f -> core (f sfs'))
```

The `route` function actually has 2 tasks:

1. Reason about the previous object state (if any) and generate logical events like collisions etc.
2. Distribute input- and logical-events to the corresponding objects.

```
route :: (Input, IL ObjOutput) -> IL sf -> IL (ObjEvents, sf)
route (input, oos) objs = mapIL routeAux objs
  where
    hs = hits (assocsIL (fmap ooState oos)) -- process all object 'State's
    routeAux (k, obj) = (ObjEvents
        { oeInput = input
        -- hit events are only routed to the objects they belong to (hence: routing)
        , oeLogic = if k `elem` hs then Event () else Yampa.NoEvent
        }, obj)

hits :: [(ILKey, State)] -> [ILKey]
hits kooss = concat (hitsAux kooss)
```

(continues on next page)

(continued from previous page)

```
  where
    hitsAux [] = []
    -- Check each object 'State' against each other
    hitsAux ((k,oos):kooss) =
        [ [k, k'] | (k', oos') <- kooss, oos `hit` oos' ]
        ++ hitsAux kooss

    hit :: State -> State -> Bool
    (Rectangle p1 _ _) `hit` (Rectangle p2 _ _) = p1 == p2
    _ `hit` _ = False
```

killAndSpawn is actually pretty simply once you know what it is doing. It just looks up every object for kill and spawn requests and produces a function composition of deleteIL and insertIL which – in case of a event – is performed on the objects. Remember the expression from `core: (\sfs' f -> core (f sfs'))`

```
killAndSpawn :: ((Input, IL ObjOutput), IL ObjOutput)
             -> Yampa.Event (IL Object -> IL Object)
killAndSpawn ((input, _), oos) =
    if any checkEscKey input
        then Event (\_ -> emptyIL) -- kill all 'State' on [Esc] => quit
        else foldl (mergeBy (.)) noEvent events
  where
    events :: [Yampa.Event (IL Object -> IL Object)]
    events = [ mergeBy (.)
                       (ooKillRequest oo `tag` (deleteIL k))
                       (fmap  (foldl (.) id . map insertIL_)
                              (ooSpawnRequests oo))
             | (k, oo) <- assocsIL oos ]
    checkEscKey (SDL.KeyDown (SDL.Keysym SDL.SDLK_ESCAPE  _ _)) = True
    checkEscKey _ = False
```

3.6 Objects

The interesting parts here are that a Object can take parameters just like any other function to produce signal functions. Here it is used to specify the initial position for example. The actual position is calculated by a simple integrator based on the user input.

```
playerObject :: Position2 -> SDL.Rect -> SDL.Pixel -> Object
playerObject p0 rect color = proc objEvents -> do
    -- .+^ is Point-Vector-addition
    -- ^+^ is Vector-Vector addition
    -- here we sum up all vectors based on the possibly multiple
    -- user inputs, thus allowing diagonal moves
    p <- (p0 .+^) ^<< integral -<
        foldl (^+^) (vector2 0 0) $ mapMaybe checkKey (oeInput objEvents)
    returnA -< defaultObjOutput { ooState = Rectangle p rect color }
    where
        checkKey (SDL.KeyUp (SDL.Keysym SDL.SDLK_UP    _ _)) =
            Just $ vector2    0 (-32)
        checkKey (SDL.KeyUp (SDL.Keysym SDL.SDLK_LEFT  _ _)) =
```

(continues on next page)

(continued from previous page)

```
            Just $ vector2 (-32)   0
        checkKey (SDL.KeyUp (SDL.Keysym SDL.SDLK_DOWN  _ _)) =
            Just $ vector2    0   32
        checkKey (SDL.KeyUp (SDL.Keysym SDL.SDLK_RIGHT _ _)) =
            Just $ vector2   32    0
        checkKey _ = Nothing

staticObject :: Position2 -> SDL.Rect -> SDL.Pixel -> Object
staticObject p0 rect color = proc objEvents -> do
    returnA -< defaultObjOutput { ooState         = Rectangle p0 rect color
                                , ooKillRequest   = (oeLogic objEvents)
                                , ooSpawnRequests = (debugIfKilled objEvents)
                                }
  where
    debugIfKilled objEvents =
        case (oeLogic objEvents) of
            Yampa.Event () -> Event [debugObject "hit"]
            _              -> Event []

debugObject :: String -> Object
debugObject s = proc objEvents -> do
    returnA -< defaultObjOutput { ooState       = Debug s
                                , ooKillRequest = Event ()
                                }
```

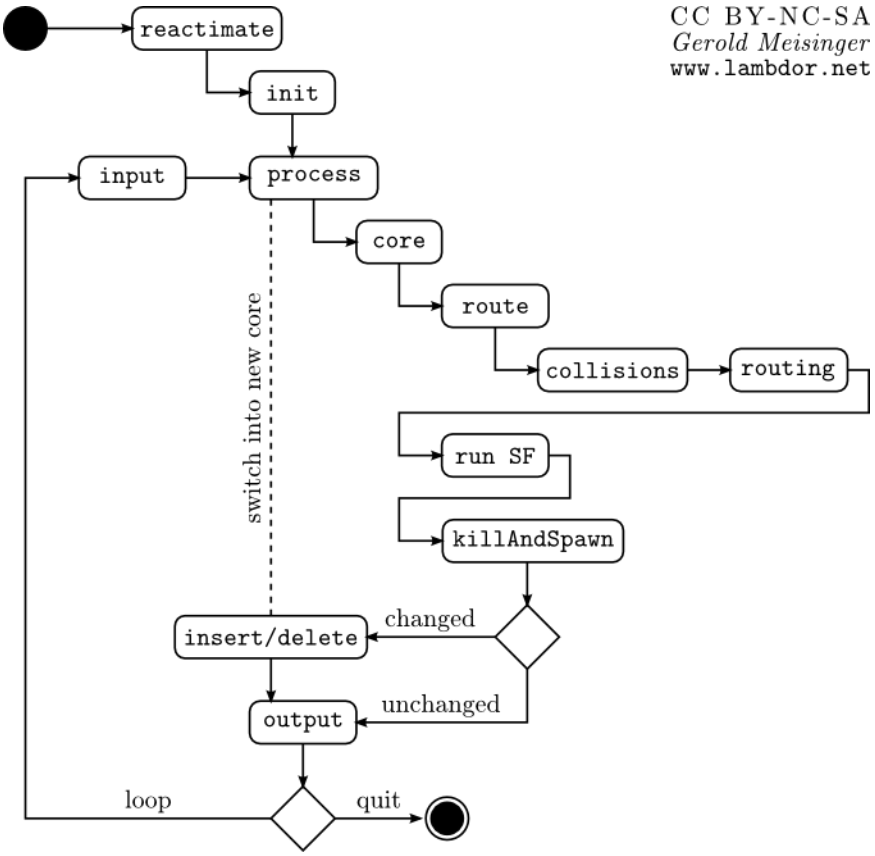
Download the whole source file! (.hs)

CHAPTER

FOUR

ACTIVITY DIAGRAM OF YAMPA REACTIMATE

This blog post was originally written back in 2010-07-10



download: `diagram (.svg)`, fonts (`cmr10.ttf`, `cmtt10.ttf`)

1. Collect the input events (`init` and `input`) in an IO task.
2. Pass them to `process` (which is purely functional) and…
3. `core` is a `Yampa.dpSwitch` which consists of `route`, the object list `IL sf` and `killAndSpawn`.
4. `route` first reasons about all **previous object states** to produce logical events (collisions etc.) and…
5. secondly bundles the input and logical events to the objects (`IL (ObjEvents, sf)`).
6. Run all the signal functions which in turn may produce kill or spawn requests of new objects…
7. which are applied in `killAndSpawn` to possibly get a new object collection (`IL Object`) which are then fed-back into `core`.

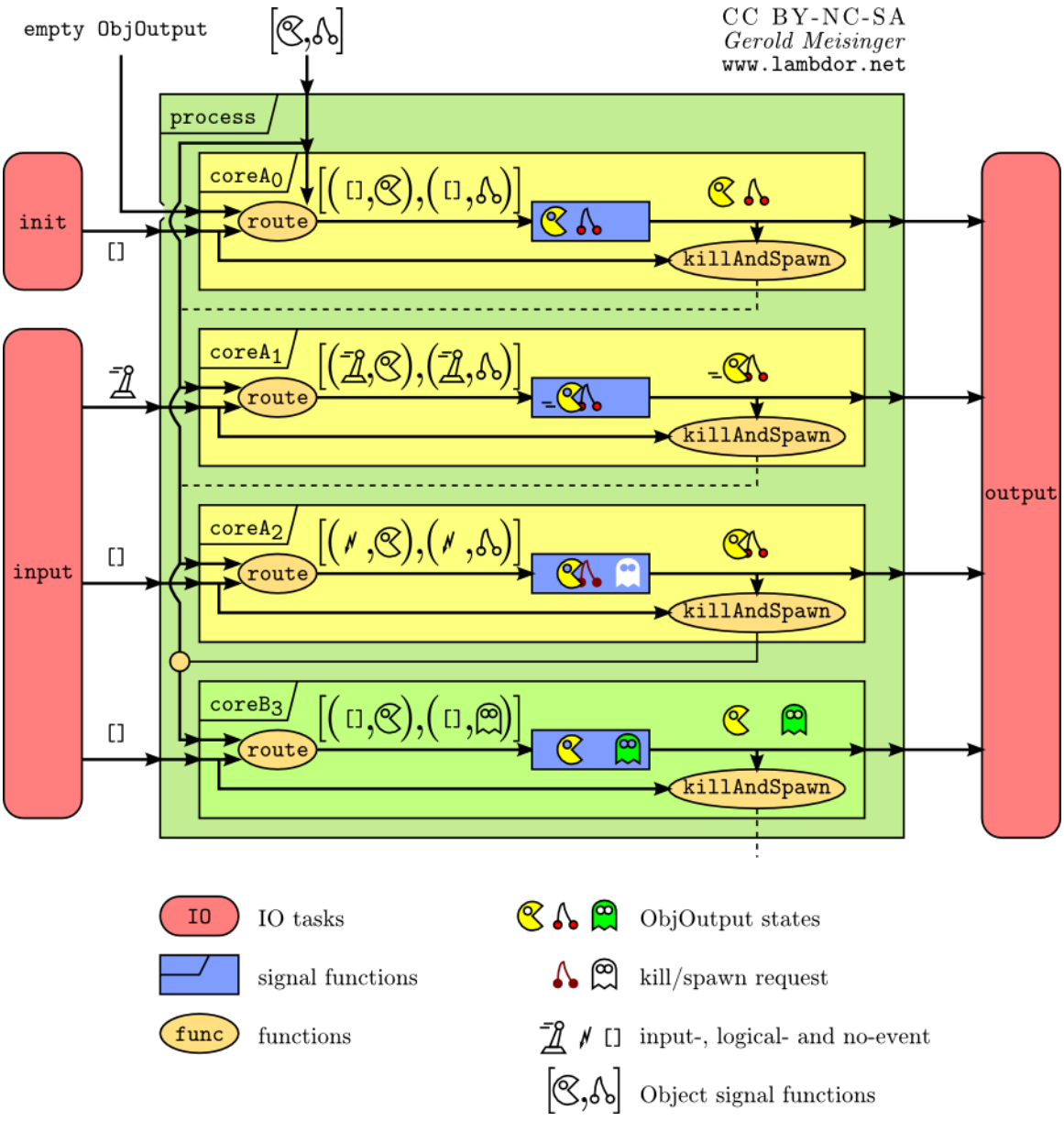
8. Render all objects states and loop.

CHAPTER

FIVE

DATAFLOW DIAGRAM OF YAMPA REACTIMATE

This blog post was originally written back in 2010-07-10



download: `diagram (.svg)`, fonts (`cmr10.ttf`, `cmtt10.ttf`)

For me as a Haskell beginner the biggest problem in understanding Yampa reactimate was how the objects are actually passed around and transformed as all the signatures are very, very… very generic. This diagram shows an example scenario staring Pacman (the player), a cherry (enemy trigger) and a ghost (the enemy).

1. Starting at the upper left corner.
2. Collect **input events** in `init` (which are empty here) and pass them through `process`, core all the way down to `route` and `killAndSpawn`.
3. `core` is called with an initial empty object state (which is **fed-back in recursively!** (1)) and an initial list of object signal functions. It is very important to separate the logic of the objects (signal functions) and the output they produce (state).
4. `route` gets the empty state and no input events, effectively **keeping the object collection** the same. Note that `killAndSpawn` doesn't switch in this step. The object states are passed to `output` where they are rendered.
5. In the **next step** (t=1), still having the **same core** (core=A), the user produces an **input event** which is routed to all objects and makes the Pacman **move** to the cherry. route only checks for collision events in the previous state, thus **no collision events** are recognized in this step.
6. In t=2, still having the **same core** (core=A), `route` detects a collision between Pacman and the cherry and produces **collision events**, which are only routed to the objects in charge. This causes `killAndSpawn` to kill the cherry, spawn the ghost and therefore generate a **switching event**, which results in the creation of a new `core` in `process`.

(I didn't really know how to illustrate the recursion of core.)

CHAPTER

SIX

MASTERTHESIS

Game-Engine-Architektur mit funktional-reaktiver Programmierung in Haskell/Yampa (german) (2010) by Gerold Meisinger

6.1 Abstract

Development of computer games comes at a high cost, thus reusable and extensible functionality is very essential in software development. A recent trend in computer game development is establishing Rapid Application Development and use of prototyping software, emphasizing fast development of playable prototypes and new functionality. Especially dynamic modification of functionality during runtime without limiting the programming environment compared to general purpose programming languages is important. Reusability, extensibility and dynamic functionality could hardly be achieved with object-oriented programming and component-based game engine architectures and the techniques used actually constrained programming.

In this thesis, general game engine architecture and various basic features of computer games are examined using functional-reactive programming in Haskell/Yampa. Due to pure functional programming, functionality can be encapsulated into completely independent and reusable functions, while reactive programming abstracts the flow of time and functionality can be composed intuitively via reactive elements. Furthermore, continuous time semantics are possible with Yampa. Overall, a game engine can be more clearly defined and divided into its basic features, thus allowing the data flow to be comprehensibly illustrated on the input-processing-output model. Finally, the principle limits of component-based game engine architectures and dynamic functionality are discussed.

6.2 Kurzfassung

Die Entwicklung von Computerspielen ist mit hohen Kosten verbunden, weshalb in der Softwareentwicklung wiederverwendbare und erweiterbare Funktionalität angestrebt wird. Ein aktueller Trend in der Computerspielentwicklung ist der Einsatz von Rapid Application Development und Prototyping-Software, wobei die schnelle Entwicklung spielbarer Prototypen und neuer Funktionalität in den Vordergrund gestellt wird. Dabei ist vor allem die dynamische Veränderung der Funktionalität zur Laufzeit wichtig, ohne dabei die Programmierumgebung gegenüber allgemeinen Programmiersprachen einzuschränken. Mit den bisher verwendeten objektorientierten Programmiersprachen und den komponentenbasierten Game-Engine-Architekturen konnten aber kaum Wiederverwendbarkeit, Erweiterbarkeit oder dynamische Funktionalität erreicht werden und schränkten die Programmierung durch die verwendeten Techniken zusätzlich stark ein.

In dieser Arbeit wird eine allgemeine Game-Engine-Architektur und verschiedene Basisfunktionalitäten von Computerspielen mittels funktional-reaktiver Programmierung in Haskell/Yampa behandelt. Durch die rein funktionale Programmierung kann die Funktionalität in vollständig unabhängige und wiederverwendbare Funktionen ausgelagert werden, während durch die reaktive Programmierung der Zeitverlauf völlig abstrahiert wird und die Funktionalität intuitiv aus reaktiven Elementen verknüpft werden kann. Weiters ermöglicht Yampa die Modellierung mit konzeptionell kontinuierlicher Zeit. Eine Game-Engine lässt sich dadurch insgesamt besser in deren elementare Bereiche aufteilen,

wodurch sich der Datenfluss am Eingabe-Verarbeitung-Ausgabe-Modell veranschaulichen lässt und die grundlegende Funktionalität einer Game-Engine definiert werden kann. Abschließend werden die grundsätzlichen Grenzen von komponentenbasierten Game-Engine-Architekturen und dynamischer Funktionalität behandelt.

CHAPTER

SEVEN

INSTALLING HASKELL PROFILE LIBRARIES

This blog post was originally written back in 2010-11-23

Profiling with GHC is done by compiling with the `-prof` parameter, which compiles the program with profile code that can be activated by executing the program with `./myprogram +RTS -p -RTS`. This creates log file called `myprogram.prof`.

```
>>> ghc -prof --make -O2 myfile.hs
>>> ./myfile +RTS -p -RTS
>>> cat myfile.prof
```

Unfortunately this doesn't work out-of-the-box. You have to install the profiling versions of all libraries and all depending libraries. And even more unfortunately, cabal doesn't resolve dependencies for profiling libraries.

```
>>> cabal install --reinstall -p libraryname
```

Update: To automatically install the profiling libraries for **future installations** edit the `~/.cabal/config` and set the profiling options to `True`. (thanks Erik)

Update2: You may only do this for libraries. I got `leksah-server` complaining about `-N2` option not being available with profiling executable.

But the reinstallation produces errors like:

```
Directory.hs:2:4:
    The export item `Permissions(Permissions, readable, writable,
                                 executable, searchable)'
    attempts to export constructors or class methods that are not visible here
cabal: Error: some packages failed to install:
MissingH-1.1.0.3 depends on haskell98-1.0.1.1 which failed to install.
haskell98-1.0.1.1 failed during the building phase. The exception was:
ExitFailure 1
```

or

```
>>     Could not find module `Data.List.Utils':
>>       Perhaps you haven't installed the profiling libraries for package
>> missingh-1.1.0.3?
>>       Use -v to see a list of the files searched for.
```

I also tried removing and recompiling some packages completely like so

```
>>> ghc-pkg unregister MissingH    # case-sensitive
>>> cabal unpack MissingH
```

(continues on next page)

(continued from previous page)

```
>>> cd MissingH-1.1.0.3/
>>> cabal configure -prof
>>> cabal build
>>> cabal install
```

There are some fundamental libraries which cannot be installed via cabal (like `parsec3`), so you rather have to:

```
>>> sudo apt-get install libghc6-*-prof
```

But eventually I found out that my packages are broken anyway (check with `ghc-pkg check`) and I had to reinstall Haskell platform, which also didn't work because:

```
>>> sudo apt-get purge ghc6    # didn't work
```

Just do:

```
>>> mv ~/.ghc6 ~/.ghc6.bak
>>> sudo apt-get purge ghc6
>>> sudo apt-get install --reinstall haskell-platform
```

Some links which were quite helpful:

- haskell-cafe@haskell.org – Profiling
- Stackoverflow – Mysterious cabal install problems

CHAPTER

EIGHT

CONNECTING YAMPA WITH LAMBDACUBE-ENGINE

This blog post was originally written back in 2012-06-11

8.1 Abstract

In this post I present a minimal LambdaCube-Engine program which connects a 3D rendering engine to Yampa FRP. The code is an adaption of the `lambdacube-basic.hs` example delivered with `lambdacube-examples`. Some changes to Yampa were necessary to allow `MonadIO` transformations in order to connect with LambdaCube. The goal is to get quickstarted with Yampa and LambdaCube, so the code was intentionally kept simple. Full source code is available at bottom of post.

8.1.1 LambdaCube Examples

First we're going to setup LamdaCube, run the lambdacube-basic example and see how it does work. Note that the latest version (0.2.4) of LamdaCube is available at GoogleCode (instead of Hackage (0.2.3)).

```
>>> cabal unpack lambdacube-engine # old version
>>> svn checkout http://lambdacube.googlecode.com/svn/trunk/ lambdacube # new version
>>> cd lambdacube/lambdacube-examples
>>> dist/build/lambdacube-basic/lambdacube-basic
# ls lambdacube-examples.cabal
# ls src/lambdacube-basic.hs
# ls media/
```

It should show 1 frame of a bluesky scene. However it stops immediately because of an issue in Elerea FRP with the following message (bug report): `thread blocked indefinitely in an MVar operation`

This is unfortunate, however we're going to integrate Yampa anyway. The important parts of the example are outlined here:

lambdacube-basic.hs (outlined)

```
import FRP.Elerea.Param
...

main = do
    openWindow
    ...
    runLCM renderSystem [Stb.loadImage] $ do
    ...
    addScene [ node "Root" "Obj" ...]
    ...
    addRenderWindow "MainWindow" 640 480 ...
    ...
    sc <- liftIO $ start $ scene ...
    driveNetwork sc (readInput ...)

scene :: RenderSystem r vb ib q t p lp
      => ...
      -> SignalGen FloatType (Signal (LCM (World r vb ib q t p lp) e ()))
scene = do
        ...
        return $ drawGLScene <$> ...

drawGLScene :: RenderSystem r vb ib q t2 p lp
            => ...
            -> FloatType
            -> LCM   (World r vb ib q t2 p lp) e ()
drawGLScene ... time = do
    updateTransforms $ ...
    updateTargetSize "MainWindow" w h
    renderWorld (realToFrac time) "MainWindow"
```

What happens here is that a LCM world (`LambdaCubeMonad`) is created and several changes are made by "shoving" through the World via `runLCM`. Finally Elerea's `driveNetwork` starts an infinite loop to make repeated `renderWorld` calls within the `LCM` context. So all we have to do is replace Elerea's `driveNetwork` with Yampa's `reactimate` and

"shove" the LCM into `reactimate`. Unfortunately the types don't match: (`IO` vs `LCM`). Also note how an external `renderWorld` is called within an `LCM` context. What's going on? Let's look up the definition! Because there are no public docs available we'll generate them ourselves:

```
>>> cd lambdacube/lambdacube-engine
>>> cabal haddock
>>> firefox dist/doc/html/lambdacube-engine/index.html
```

```
data LCM w e a

Instances
Monad (LCM w e)
Functor (LCM w e)
Applicative (LCM w e)
MonadIO (LCM w e)
```

What's interesting here is that `LCM` instances `MonadIO` and provides a free type variable `a`. Thus `LCM` has an IO context and can be extended with other Monad contexts and types. All we have to do therefore is to make Yampa's `reactimate` less strict and do some Monad lifting.

```
>>> cabal install transformers
>>> cabal unpack Yampa
>>> cd Yampa-0.9.3/src/FRP
```

Yampa.hs

```
reactimate :: IO a
           -> (Bool -> IO (DTime, Maybe a))
           -> (Bool -> b -> IO Bool)
           -> SF a b
           -> IO ()
```

change to:

```
import Control.Monad.IO.Class (MonadIO)
...

reactimate :: MonadIO m
           => m a
           -> (Bool -> m (DTime, Maybe a))
           -> (Bool -> b -> m Bool)
           -> SF a b
           -> m ()
```

Now we can call `reactimate` within `runLCM` and use Yampa as usual. Copy the `lambdacube-basic.hs` file and extend `lambdacube-examples.cabal` with the new executable.

YampaCube.hs (outlined)

```
type ObjInput  = (Int, Int)       -- mouse-position
type ObjOutput = (String, Proj4) -- object name, transformation

main :: IO ()
main = do
```

(continues on next page)

(continued from previous page)

```
    mediaPath <- getDataFileName "media"
    renderSystem <- mkGLRenderSystem
    runLCM renderSystem [Stb.loadImage] (reactimate (initput title mediaPath) input␣
↪output (process objs))
    closeWindow

initput :: RenderSystem r vb ib q t p lp
      => Bool
      -> LCM     (World r vb ib q t p lp) e (DTime, Maybe (Bool, ObjInput))
initput _ = do
    ...
    liftIO $ openWindow ...
    ...
    addScene [ node "Root" "Obj" ...]
    ...

output :: RenderSystem r vb ib q t p lp
      => Bool -> [ObjOutput]
      -> LCM   (World r vb ib q t p lp) e Bool

cam :: SF ObjInput ObjOutput
cam = proc _ -> do
    t <- localTime -< ()
    returnA -< ("Cam", translation (Vec3 0 0 (realToFrac t)))
```

```
>>> cd lambdacube/lambdacube-examples
>>> cabal configure
>>> cabal build
>>> dist/build/YampaCube/YampaCube
```

Download full source code: `YampaCube.hs`

Viola! What you should see is a small box with the bluesky texture and the camera slowly moving away from it.

CHAPTER

NINE

WHY I SWITCHED FROM COMPONENT-BASED GAME ENGINE ARCHITECTURE TO FUNCTIONAL REACTIVE PROGRAMMING

This blog post was originally written back in 2010-08-16

Components have become pretty popular these days and I'd like to share some issues we had with them in our game projects. I was experimenting with component-based game engine architectures for 2 years and eventually stumbled upon functional reactive programming (FRP) to solve some of the issues of game-object components. I hope to provide some arguments from an *objectoriented programming* (OOP) viewpoint to why I think FRP helps to write more reusable code.

The argument for component-based game-engines usually starts like this: Game-objects should be represented like objects in reality but as the project progresses however, class hierarchies become more and more complex and thus should be split into small, reusable components. Game developers like abstract and re-usable game-engines after all. The following diagram illustrates such a monolithic class hierarchy (adopted from the book *Game Engine Architecture*):

Game-object components address these issues by reducing game-objects to identifiable containers of components, where each component encapsulates some reusable functionality and automatically communicates with other components. A component could be something like: a position, player movement, 3D model, health-points and so on. For the communication of components in our engine we used messages, events or let them directly search for components implementing a specified interface, which is illustrated in the following diagram:

In component-based architecture we were mostly concerned about the component intercommunication, like the player movement for example: Should the Mover-component manipulate the position directly or send movement-messages? Or should the Position-component listen to movement-events? What if we want the player movement to behave just a little differently, like being a little random? Do we implement a new RandomMover-component, or do we need a new component,... again,... just to encapsulate the random function into a component? And how does this fit into the automatic Position-Mover-communication?

In the book *Component-based Software Engineering* (CBSE) they actually define a component as:

> A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. [...] A component model defines **specific interaction and composition standards**. [...]

In plain English: "In order to allow a new component to be added to an existing environment of components in a game-object and automatically communicate with each other, the communication protocol between the components has to be **predefined**". I think the component-based game engine literature mixes-up "combining existing functionality to define game-specific logic" (= reusing functionality) and "automatic communication within unknown game-objects" (= dynamic functionality). Automatic communication is restricted to the known components and new functionality always has to be game-specific (either hard-coded, or in scripts, or any other data-driven method). Even with components you define the player game-object specifically at some point: Player = PositionComponent + VisualComponent + InputComponent.

In FRP everything is based upon time, thus time should be abstracted away into "time dependent functions" (behaviors). A Mover-"component" should just produce a translation vector over time from user-input — nothing more! And if we want it to be a bit random, we just compose the output with the random function in the game-object definition. A pure mathematical function is the most isolated, self-existing and reusable component you can get, as it depends (and only depends!) on the direct input. There is no need the encapsulate a function into a component... which is then managed by a component container (game-object) and defines a lot of messages and events to get the data to the right point.

I'm arguing that game-objects are always game-specific but with time-dependent functions you just have to combine the existing functionality in the right way for every specific game-object. For example, let the movement of a game-object by calculated by: the initial position + some translation over time from user-input + a random factor + ... and so on. You may then compose the movement behavior into more complex behaviors (or complete game-objects if you like) in every way you can imagine: A movable, static image game-object? MovingImage = translation function + draw image function. A movable, animation game-object? Certainly! MovingAnimation = same translation function + draw animation function. Or just a static animation perhaps? StaticAnimation = position + draw animation function.

I'm not going into more detail about FRP for now, you can find more information on my blog or on the internet. Component-based software engineering can still be applied in this environment on higher level, like the communication between subsystems or game-objects, but not on the level of game-object functionality level! If you still think you want to implement game-object components, ask yourself the following questions:

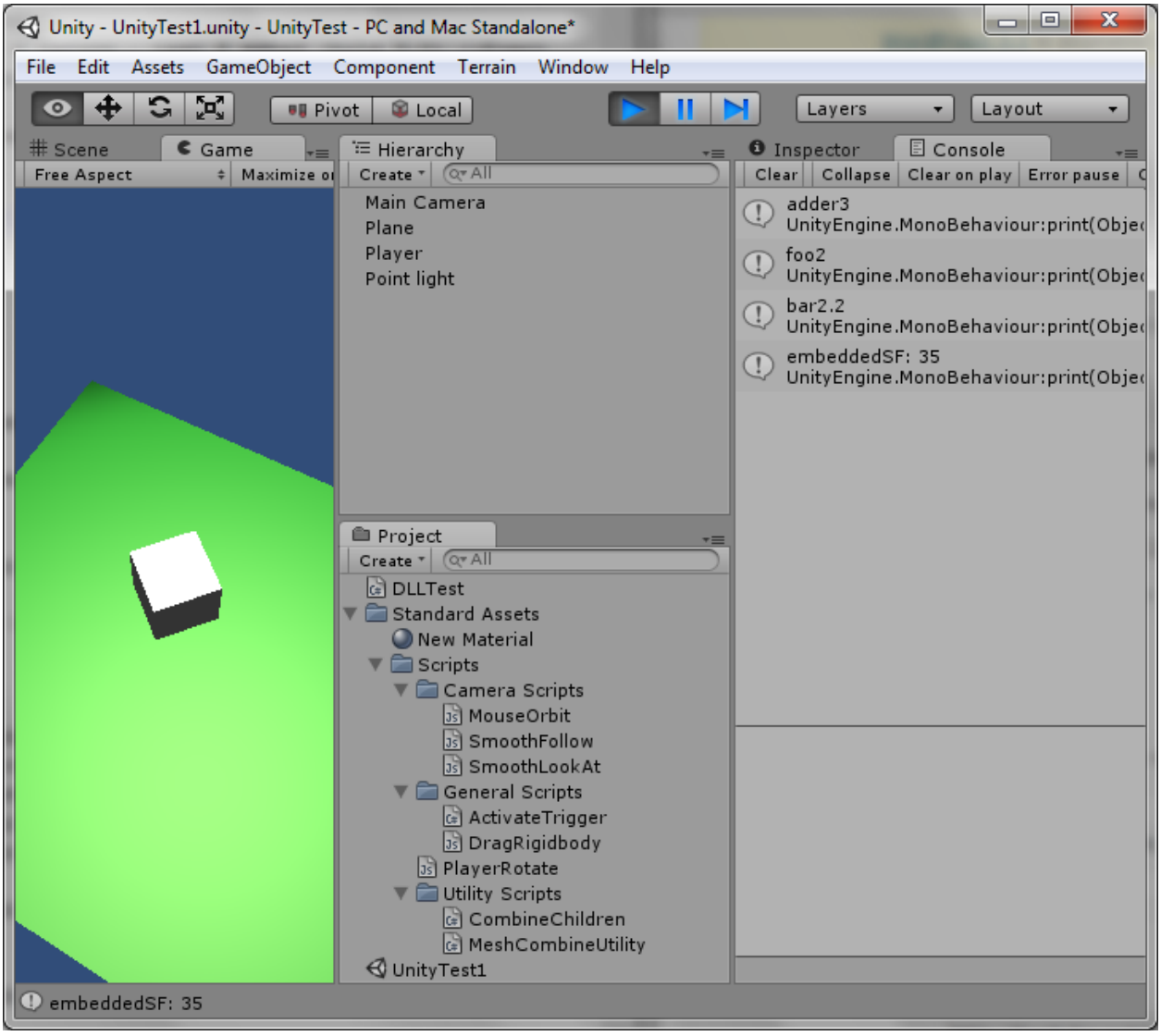
- How can different functionality be defined in terms of relative complex, non-elementary components?
- What are the most elementary components?
- In which way are elementary components different from pure functions?
- How can existing components automatically communicate with new messages from new components?
- What is the point of ignoring a message that a component doesn't know of?
- What happened to the input-process-output model after all?

CHAPTER

TEN

CONNECTING HASKELL FRP TO THE UNITY3D GAME FRAMEWORK

This blog post was originally written back in 2011-04-02



10.1 Abstract

This post describes how to connect Haskell to Unity3D by compiling a Haskell shared library to a Win32 DLL and dynamically loading the DLL in a scripted Unity3D object via DLLImport. The DLL also uses an additional module (Yampa), thus FRP can be used within a sophisticated game framework (with the exception that this concept doesn't connecte to the Unity3D core yet, which would allow stateful arrows within the game logic... which renders it rather useless right now :/). Unity3D was chosen because it provides a complete game creation workflow and a demo version is available for free. In contrast, there are no full game creation frameworks available for Haskell right now (to my knowledge, but my status is based on). DLLs were chosen over .NET libraries because to my knowledge it is not (yet) possible in Haskell to compile to a .NET library (and I think it was one of the reasons why F# was born). .NET libraries would of course allow for an easier integration in Unity3D. Unity plugins are only available in the pro version... which is a pity. My personal goal was to connect functional temporal programming (functional reactive programming) to a sophisticated game framework to allow the creation of great games with FRP game logic. Besides, it was a great exercise for foreign function interfaces (FFI) and a test on how well Haskell can be integrated into existing applications.

10.2 Instructions

What actually happens when a Haskell program is packed into a shared library and called from an external program? The actual DLL is built from C code, which exports the Haskell functions and also starts up a Haskell runtime, which delegates the external function calls to Haskell. All functions are called within an unsafe environment, like `main :: IO ()` (... this shouldn't necessarily be the case I think – FFI to Miranda?)

1. Install at least GHC 7, because Windows DLLs don't work with GHC 6 (see: GHC 7 change-log, section 1.5.4. DLLs: "Shared libraries are once again supported on Windows.").
2. Download and install Unity3D.
3. Download `DllTest.hs` (Haskell code for game logic) and adopt to your needs.

- As these function get evaluated within a unsafe environment, they are all IO Monads.
- In Windows you have to use `ccall` (citation needed)

4. Download `DllTest.cs` (Unity3D script file) and adopt to your needs.

- Make sure to start and stop the Haskell runtime correctly. Nothing special really, just tag the external functions with `[DllImport]` and call.

5. Visit GHC manual and search for `// StartEnd.c` and save the code to `StartEnd.c`.

- This is the actual C program which is converted to a Windows DLL. It also starts up a Haskell runtime and allows the external program to call functions.

6. Install all dependent modules with shared library option enabled (f.e. yampa)

```
>>> cabal install --reinstall --enable-shared yampa.
```

7. Copy shared libraries of all used modules to your Haskell project directory `C:/Program Files (x86)/Haskell Platform/2011.2.0.0/lib/yampa.../libHSYampa-0.9.2.3-ghc7.0.2.dll.a` (...there may be a better solution for this)
8. Copy the file `C:/Program Files (x86)/Internet Explorer/IEShims.dll` to your Haskell project directory (... I forgot why)
9. Compile step 1

```
>>> ghc -dynamic -fPIC -c DllTest.hs
```

- This is an intermediate step (see Shared libraries that export a C API) which produces the object file (.o) from C code (DllTest.o and DllTest_stub.o)
- The `-fPIC` flag is required for all code that will end up in a shared library.

10. Compile step 2

```
>>> ghc -dynamic -shared DllTest.o DllTest_stub.o StartEnd.o libHSYampa-0.9.2.3-ghc7.0.2.
↪dll.a -o libHsDllTest.dll
```

- The object files, the runtime starter and the archive file for the additional module are compiled together into the final DLL.

11. Use the DependencyWalker tool of the Visual Studio IDE to find errors in the DLL (it may be included in Visual Studio Express or Visual Studio Code which is available for free).
12. Create a Unity3D project and add an object using the `DllTest.cs` as a script file (this is out-of-scope of this article).

10.2.1 Notes on Haskell DLLs

- If you didn't work with shared libraries before, make sure you understand the Foreign function interface (FFI) and all the different types of shared libraries. What's used here is a **dynamically loaded shared library (DLL)** which gets loaded by an external (non-Haskell) programm (Unity3D).
- Read GHC manual on Building and using Win32 DLLs (Question: What is the option `-lfooble` good for?).
- All modules used by the Haskell program have to be compiled into **one-big DLL**. GHC 7 manual on Win32 DLLs: "Adder is the name of the root module in the module tree (as mentioned above, there must be a single root module, and hence a single module tree in the DLL".
- Make sure **shared-library versions** are installed of all used modules (see above). They get installed somewhere into *C:/Program Files (x86)/Haskell Platform/2011.2.0.0/lib...*
- The GHC 7 documentation is outdated (see: GHC Ticket #4825: DLLs not documented consistently).

10.2.2 Notes on Unity3D

- When the DLL file is recompiled, **Unity3D has to be restarted** in order to reload the DLL.
- If Unity3D doesn't find the DLL file, make sure the correct DLL file (in the directory where `ghc` is executed) is located in your Unity3D project where the .exe file is located which gets compiled by Unity3D.
- Unity3D log files are located in: `C:/Users/myuser/AppData/Local/Unity`.
- Unity3D crash reports are located in: `C:/Users/gerold/AppData/Local/Temp/crash_...`
- stdcall instead of ccall on Windows, this appears to be invalid for Unity3D (also see GHC manual on Win32 DLLs.
- Unity3D uses the following script languages: C#, JavaScript, Boo.

10.3 Description

The actual concept is pretty straight forward. Simply write all Haskell functions and make the function interfacing to the external program an IO monad, which gets called by a potentially inpure program. The program and all depending modules are compiled into one big DLL. This DLL gets imported by the external program (Unity3D) and makes the functions visible. Before an Haskell function can be called, a complete Haskell runtime environment has to be started. In the following example, when an object is created, it starts up a Haskell runtime, runs an embedded arrow and shuts the Haskell runtime down again. This is of course rather useless right now, but it shows an arrow running in Unity3D.

DllTest.hs (short)

```
{-# LANGUAGE ForeignFunctionInterface, Arrows #-}

module Adder (
  embeddedSF,
  ...
) where

import FRP.Yampa

embeddedSF :: Double -> Double -> Double -> IO Double
embeddedSF v0 v1 v2 = return . last $ embed integral (v0, [(1.0, Just v1), (1.0, Just␣
↪v2)])

foreign export ccall embeddedSF :: Double -> Double -> Double -> IO Double
```

DllTest.cs (short)

```
class DLLTest : MonoBehaviour {
[DllImport ("libHSDllTest")]
private static extern void HsStart();
[DllImport ("libHSDLLTest")]
private static extern void HsEnd();
[DllImport ("libHSDLLTest")]
private static extern double embeddedSF(double v0, double v1, double v2);

void Awake () {
  HsStart();
  print ("embeddedSF: " + embeddedSF(12.0, 23.0, 45.0));
  HsEnd();
}
```

10.4 Resources

10.4.1 Haskell DLLs

- Foreign Declarations
- Stack overflow: Haskell compile dll
- GHC wiki – Using the FFI
- GHC manual – Chapter 8. Foreign function interface (FFI)

- GHC manual – 11.6. Building and using Win32 DLLs
- GHC manual – 4.12. Using shared libraries
- Real World Haskell – Chapter 17. Interfacing with C: the FFI
- [Haskell-cafe] FFI, C/C++ and undefined references
- Update: Creating a Windows DLL from a Haskell Program and calling it from C++ (todo)

10.4.2 Unity3D

- Documentation on Scripts
- Programming Introduction
- Extensions
- Event Execution Order
- Scripting forum
- "Plugins are a Pro-only feature [probably developer side only]. For desktop builds, plugins will work in standalones only. They are disabled when building a Web Player for security reasons."
- Using F# with Unity "There are some things that are available to general .NET code that are not supported by Unity."
- What would it take to make F# work with Unity?
- Is it possible to build a standalone .net application that link UnityEngine.dll, Is it possible to start Unity from a C# project?: "This is not possible. UnityEngine.dll is a small .NET wrapper around the native Unity engine. Without the Unity engine itself, UnityEngine.dll is merely an empty wrapper around nothing.There is no way to somehow get access to the Unity API in a homegrown .net application. The UnityEngine API is tightly coupled to its Unity host. They are one, and can't be split."

10.5 Open questions

I don't know how to integrate **stateful arrows** into the game logic yet. Probably the Haskell runtime within the DLL has to be started on some OnGameStart-event. Each object has to run its own independent arrow code and the update ticks propagated to something like the `reactimate` procedure.

CHAPTER

ELEVEN

QUICKSTARTING HIPMUNK

This blog post was originally written back in 2011-04-23

11.1 Abstract

Youtube - Hipmunk tutorial video

This post describes a minimal Hipmunk program, a Haskell binding to the Chipmunk 2D physics engine, featuring a simple falling circle with circle collision shape. The goal was to exemplify how a Hipmunk program is structured corresponding to a FRP game engine model (see The Yampa Arcade (archived)) and testing a sophisticated 2D physics engine (for simplicity) with Haskell. A physics engine could actually be completely pure, investigations retrieved Hpysics, a pure functional physics engine. Didn't use it, looks very basic, yet complete, but it's an abandoned project.

Youtube - Hpysics simple demo

11.2 Description

Most rigid body physics engine define the following objects:

- **Space**: a physical simulation space independent from other spaces, describes gravity. When a body is added, the object becomes dynamic. When the shape is added too, collisions are also handled.
- **Body**: a point which describes the position, using mass, moment of inertia (rotation), velocity, force, torque etc.
- **Shape**: a geometrical figure which describes collision shapes and optionally has an offset to the body (f.e. when using multiple shapes)
- **Constraints**: describe different restrictions on the movement of Bodies, like Pins, Springs, Ropes…

The scene definition thus is pretty straight forward: setup scene, run simulation, handle collisions, loop. What is unfortunate though is that – like all FFI bindings – the binding to the external Chipmunk engine is fundamentally unsafe, so there are lots of IO Monads to do basic stuff. Using Hipmunk in a FRP environment like Yampa would require lots of "task-" (or "message-") definitions and communication from the logic all the way to the physics engine, only to apply forces. Note the many unnecessary IO and State Monads. Here is a small code snipped which does 3 simulation steps:

```
import Control.Monad
import Data.IORef
import Data.StateVar -- defines $=
import qualified Physics.Hipmunk as H

main :: IO ()
```

(continues on next page)

(continued from previous page)

```haskell
main = do
    H.initChipmunk                          -- initChipmunk :: IO ()

    space <- H.newSpace                     -- newSpace :: IO Space
    H.gravity space $= H.Vector 0 (-10)     -- gravity  :: Space -> StateVar Gravity

    body <- H.newBody mass momentOfInertia -- newBody  :: Mass -> Moment -> IO Body
    -- (+:) is a convenient function for vector construction defined in Playground.hs
    -- compare with H.Vector above
    H.position body $= 0 +: 0               -- position :: Body -> StateVar Position

    -- class Entity a
    -- instances: Shape, Body, StaticShape, (Constraint a)
    H.spaceAdd space body                   -- spaceAdd :: Space -> a -> IO ()

    -- newShape :: Body -> ShapeType -> Position -> IO Shape
    shape <- H.newShape body (H.Circle 5) offset

    -- step :: Space -> Time -> IO ()
    H.step space elapsedSeconds -- IO () => y = 0
    H.step space elapsedSeconds -- IO () => y = -10
    H.step space elapsedSeconds -- IO () => y = -30

    pos <- get . H.position $ body
    putStrLn . show $ pos

  where
    mass = 1.0
    momentOfInertia = H.infinity
    offset = H.Vector 0 0
    elapsedSeconds = 1.0

-- | Constructs a Vector.
(+:) :: H.CpFloat -> H.CpFloat -> H.Vector
(+:) = H.Vector
infix 4 +:
```

11.3 References

- Hackage Hipmunk
- Hackage Hipmunk Playground
- Chipmunk official website

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search