Functional Reactive Programming

for Natural User Interface

"I have no special talents. I am only passionately curious."

- Albert Einstein

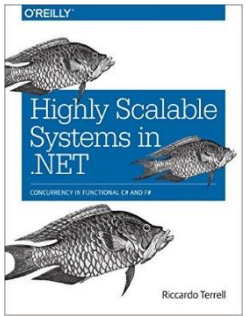
Riccardo Terrell

Agenda

- What is Functional Reactive Programming – FRP vs RP
- Functional Reactive Programming foundations and motivations
- Functional Reactive Programming implemented in F#
- A little of Reactive programming in Functional style
- Natural User Interface with Leap Motion in Action

Something about me - Riccardo Terrell

- Originally from Italy, currently - Living/working in Washington DC ~10 years
- +/- 19 years in professional programming
  - C++/VB → Java → .Net C# → Scala → Haskell → C# & F# → ??
- Organizer of the DC F# User Group
- Working @ statmuse
- Polyglot programming believer the mechanism to find the right tool for the job



Authoring book on Concurrency adopting the Functional Paradigm in C# & F#

Available for presale: http://tinyurl.com/zzpv9gx

@trikace *rickyterrell.com* *tericcardo@gmail.com*

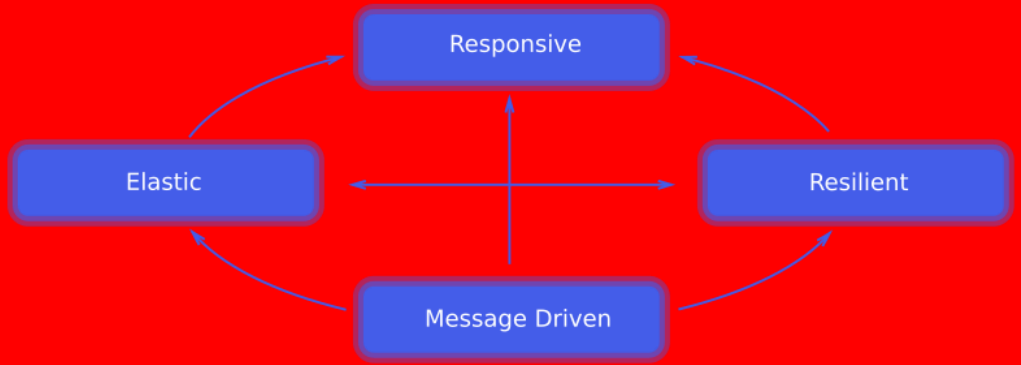
Functional Reactive Programming

Functional Reactive Programming

- declarative
- stateless
- side-effects free

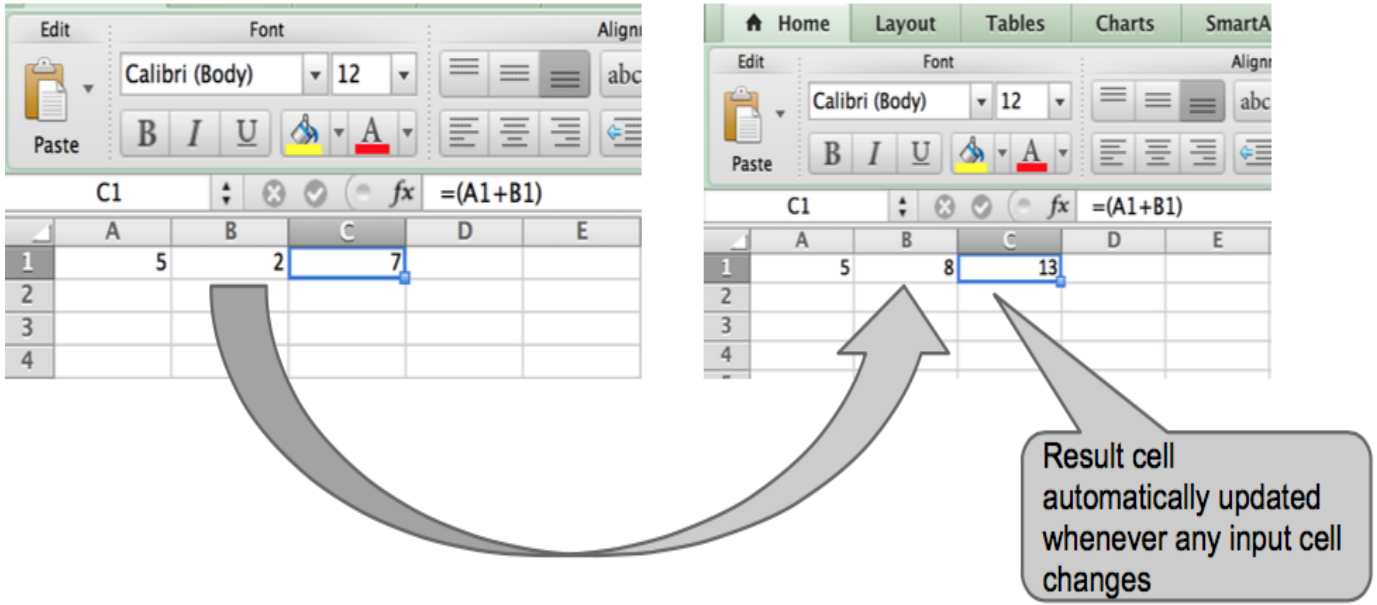
- referential transparency
- immutable
- composition

Functional Reactive Programming



http://www.reactivemanifesto.org

SpreadSheet == Mother of All Reactive Programming



Functional Recative Programming

Dynamic evolving values over time

Functional Reactive Animation

Functional Reactive Animation

Conal Elliott
Microsoft Research
Graphics Group
conal@microsoft.com

Paul Hudak
Yale University
Dept. of Computer Science
paul.hudak@yale.edu

Abstract

Fran (Functional Reactive Animation) is a collection of data types and functions for composing richly interactive, multimedia animations. The key ideas in Fran are its notions of *behaviors* and *events*. Behaviors are time-varying, reactive values, while events are sets of arbitrarily complex conditions, carrying possibly rich information. Most traditional values can be treated as behaviors, and when images are thus treated, they become animations. Although these no-

- capturing and handling sequences of motion input events, even though motion input is conceptually continuous;
- time slicing to update each time-varying animation parameter, even though these parameters conceptually vary in parallel; and

By allowing programmers to express the "what" of an interactive animation, one can hope to then automate the "how" of its presentation. With this point of view, it should

http://conal.net/papers/icfp97/icfp97.pdf

Functional Reactive Programming adoption

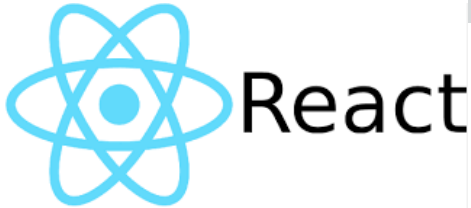
- *Graphical User Interfaces (GUI)*
- *Digital Music*
- *Robotics*
- *Graphical Animation*
- *Sound Synthesis*
- *Virtual Reality Environments*
- *Games*

Origin of Functional Reactive Programming

- 1997 – Functional Reactive Animation. Elliot and Hudak and the FRAN Library
- 2001 – Genuinely Functional User Interfaces. Courtney, Elliot (Arrowized-FRP)
- 2002 – Functional Reactive Programming, Continued… Nilsson, Courtney, and Peterson (Yampa)
- 2003 – Yampa Arcade. Courtney, Nilsson and Peterson
- 2009 – Push-Pull functional reactive programming. Elliot

Has evolved in a number of directions and into different concrete implementations

FRP becomes Main-Stream



Let's be Mainstream - Evan Czaplicki
https://www.youtube.com/watch?v=oYk8CKH7OhE

What is Functional Reactive Programming

"FRP is about handling time-varying values like they were regular values."

- Haskell Wiki

Functional Reactive Programming is:

- Temporally continuous *(Natural & Composable)*
- Denotative *(Elegant & Rigorous)*

Denotational Semantics

Denotational Semantics map each part of a program to a mathematical object (denotation), which represents the meaning of the program in question.

These mathematical objects can be represented by the integer *45*

- 45
- 33 + 12
- 5 * 9
- sum [1..9]

Semantic Domain

Denotational Semantics = Simple Design

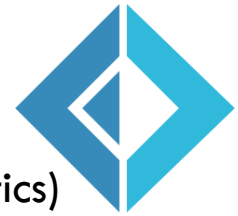
Denotational Semantics map each part of a program to a mathematical object (denotation), which represents the meaning of the program in question.

Denotational Semantics properties

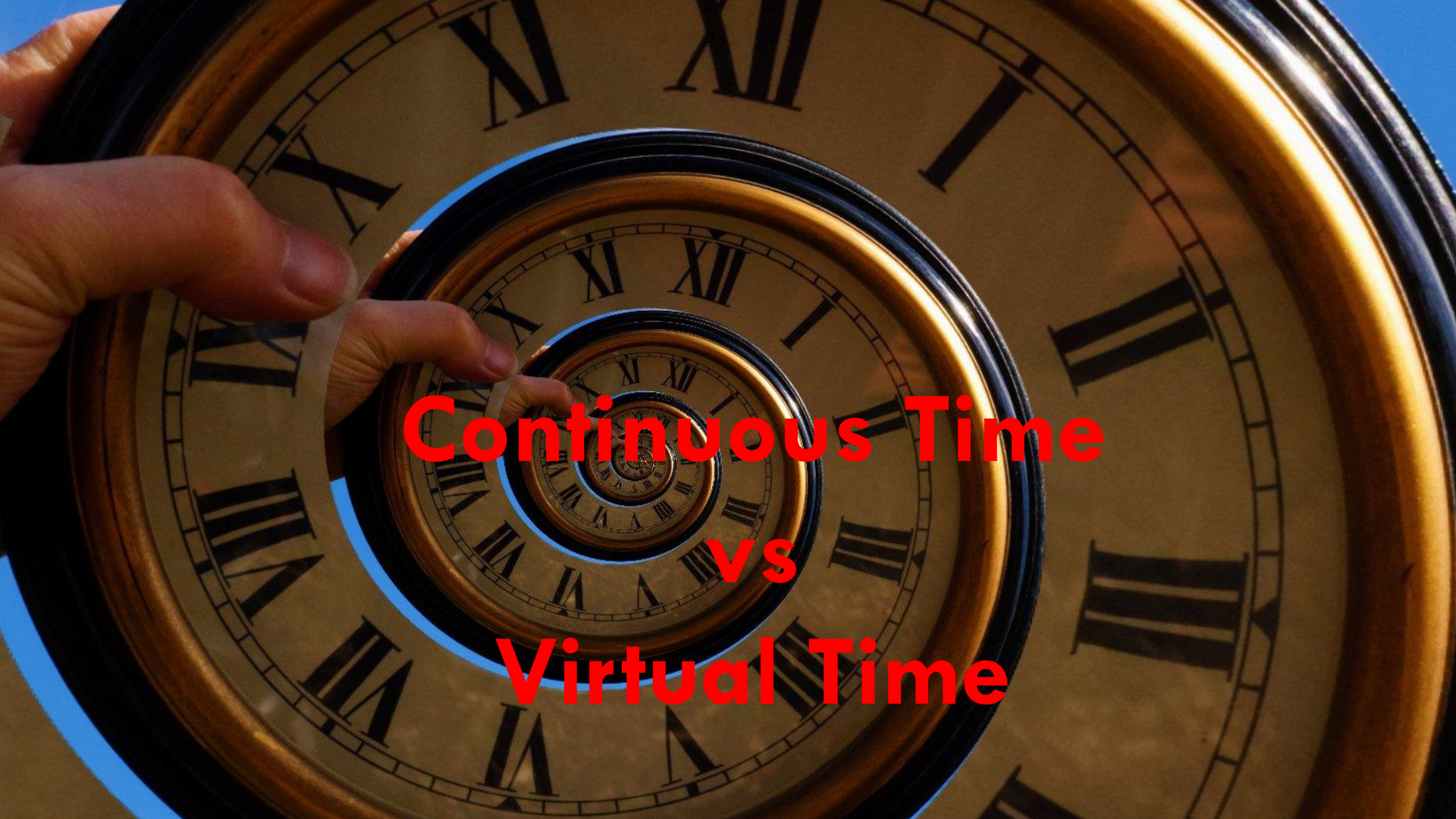
- leads to simple design
- emphasizes declarative programming style (What vs How)
- uses math to prove a property of a program
- proofs that compositionality holds for all building blocks

Foundation of FRP – Time

(precise & simple semantics)

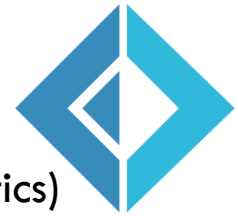


```
type Time = float
```


Continuous Time
vs
Virtual Time

Foundation of FRP - Behavior

(precise & simple semantics)



```
type Time = float
```

```
type 'a Behavior = Behavior of (Time -> 'a)
```

Foundation of FRP - Behavior

```
type Time = float
```

```
type 'a Behavior = Behavior of (Time -> 'a)
```

```
// The time itself
let time = Behavior (fun t -> t)   // identity function (id)

// Behavior constant over time
let conBeh = Behavior (fun _ -> "Hello FRP!")

// Behavior that increase at 2.5 the rate of time
let incrSpeedBeah = Behavior (fun t -> t * 2.5)
```

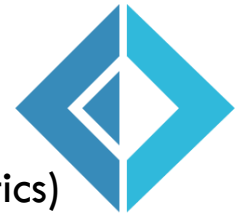
Behavior API – Original Implementation

```
let  lift0 a = Behavior a

let  lift1 (a -> b) = Behavior a -> Behavior b

let  lift2 (a -> b -> c) =
           Behavior a -> Behavior b -> Behavior c
```

```
let time =  Behavior (fun t -> t)

let ``time7`` = lift1 ((*) 7.0) time

let createBeh f:(Time -> 'a) = (lift1 f) time
```

Foundation of FRP - Event

(precise & simple semantics)



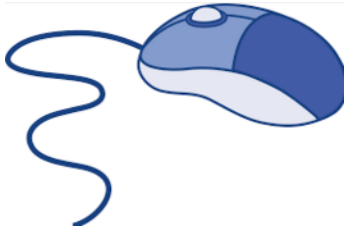
```
type Time = float
```

```
type 'a Behavior = Behavior of (Time -> 'a)
```

```
type 'a Event = Event of [(Time * 'a)] - non-decreasing time
```

```
// When the Event passes 3 secs increase its speed
let event = Event (fun t -> if (t > 3.) then Some(t*2.5) else None)
```

FRP - Mouse Position



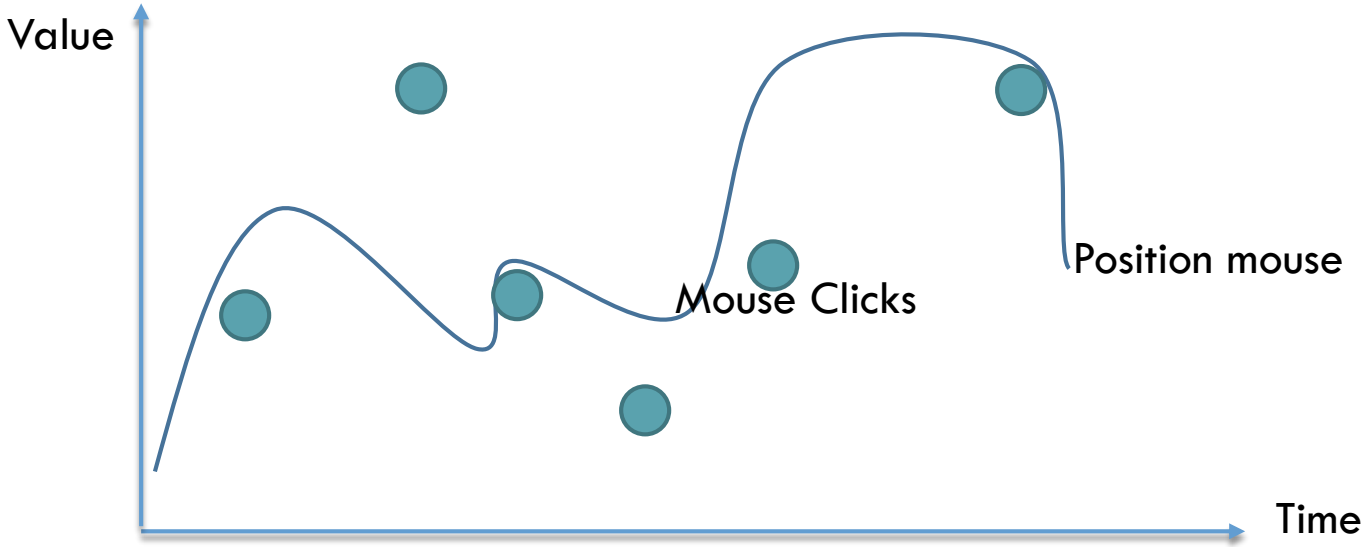
```
Event Based view
MouseMovedEvent (position: Position)
```


```
FRP view – at any point in time represents the current mouse position
mousePosition = Behavior [Position]

inRectangleBeh(ul: Position , lr:Position) : Behavior [bool] =
        let position = mousePosition()
        Behavior [ul <= position && position <= lr]
```

Foundation of FRP - Behavior

```
type 'a Behavior = Behavior of (Time -> 'a)
```

```
type 'a Event = Event of [Time -> 'a]
```



“So, what is FRP? You could have invented it yourself, start with these ideas:”

http://stackoverflow.com/questions/1028250/what-is-functional-reactive-programming - Conal Elliot

- *Temporal modeling*
  - *Composable Behavior first class values*
- *Event modeling*
  - *Composable Event first class values*
- *Declarative reactivity*
  - *Semantic in terms of temporal composition*
- *Polymorphic media*
  - Set of combinators applicable to any types of time-varying values

Push-Pull Functional Reactive Programming

Push-Pull Functional Reactive Programming

Conal Elliott

LambdaPix

conal@conal.net

Abstract

Functional reactive programming (FRP) has simple and powerful semantics, but has resisted efficient implementation. In particular, most past implementations have used demand-driven sampling, which accommodates FRP's continuous time semantics and fits well with the nature of functional programming. Consequently, values are wastefully recomputed even when inputs don't change, and reaction latency can be as high as the sampling period.

more composable than their finite counterparts, because they can be scaled arbitrarily in time or space, before being clipped to a finite time/space window.

While FRP has simple, pure, and composable semantics, its efficient implementation has not been so simple. In particular, past implementations have used demand-driven (pull) sampling of reactive behaviors, in contrast to the data-driven (push) evaluation typically used for reactive systems, such as GUIs. There are at least two strong reasons for choosing pull over push for FRP:

http://conal.net/papers/push-pull-frp/push-pull-frp.pdf

Chains of simple phases

```
type 'a Behavior = Behavior of
                   (Time -> ('a * (unit -> 'a Behavior)))
```

```
let rec pureBeh value = Behavior(fun time ->
                                 (value, fun () -> pureBeh value))
```

Chains of simple phases

```
type 'a Behavior = Behavior of
                   (Time -> ('a * (unit -> 'a Behavior)))
```

```
let rec pureBeh value = Behavior(fun time ->
                        (value, fun () -> pureBeh value))
```

```
let rec timeBeh = Behavior(fun time ->
                           (time, fun () -> timeBeh ))
```

Chains of simple phases

```
type 'a Event = Event of
                (Time -> ('a option * (unit -> 'a Event)))
```

```
let rec pureEvt value = Event(fun time ->
                    (Some(value), fun () -> pureEvt value))
```

FRP Behavior can compose

```
fmap :: (a -> b) -> Behavior a -> Behavior b
```

```
pure :: a -> Behavior a
(<*>) :: Behavior (a -> b) -> Behavior a -> Behavior b
```

- Less learning and more leverage
- Specifications and laws for “free”

FRP Behavior can compose

```
fmap :: (a -> b) -> Behavior a -> Behavior b
```

```
pure :: a -> Behavior a
(<*>) :: Behavior (a -> b) -> Behavior a -> Behavior b
```

```
type Position = Position of (float*float)

let inRectangleBeh (ul:Position, lr:Position) : bool Behavior =
        pureBeh (fun (position:Position) ->
        if ul <= position && lr <= position then true
        else false) <*> mousePositionBeh // Position Behavior
```

FRP Event API

```
never :: 'a Event
(.|.) :: 'a Event -> 'a Event -> 'a Event
whenEvent :: bool Behavior -> unit Event
whileEvent :: bool Behavior -> unit Event
(.&.) :: `a Event -> 'b Event -> ('a * `b) Event
(=>>) :: `a Event -> ('a -> `b) -> 'b Event
```

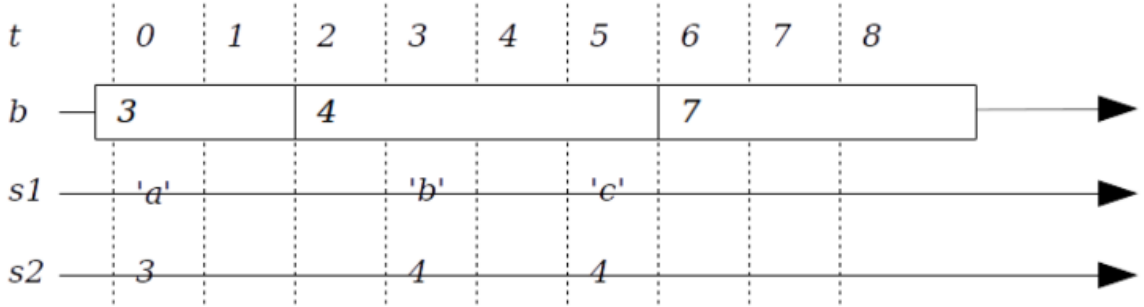
FRP – Combinators

```
type Event ::
// (‘a -> ‘b) -> Event ‘a -> Event ‘b
let map(f : 'a -> 'b) : Event<'b> = // …
// (‘a -> bool) -> Event ‘a
let filter(f : 'a -> bool) : Event<'a> = // …
// (Event ‘a * Event ‘a) -> Event ‘a
let merge(ea : Event<'a>, eb : Event<'a>) : Event<'a> = // …
let (.|.) = merge
// ‘a -> Event<‘a -> ‘a> -> Behavior ‘a
let accum (value:’a) (evt:Event<'a->’a>) : Behavior<'a> = // …
```

FRP – Event snapshot

```
type Event
// Behavior c -> (a -> b -> c) -> Event c
let snapshot(b : Behavior<'b>, f : 'a -> 'b -> 'c) : Event<'c> =
```

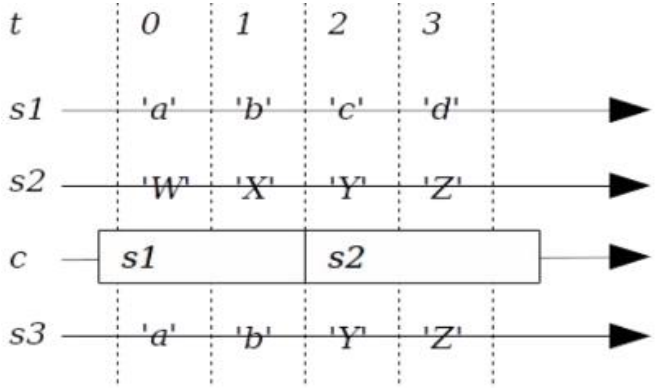
```
let c = Hold 3 (MkStream([([1],4), ([5],7)]) [0]
let s1 = MkStream [([0], ‘a’), ([3], ‘b’), ([5], ‘c’)]
let s2 = snapshot (flip const) s1 c
```



FRP – Behavior switch

```
type Behavior
// Behavior<Behavior<‘b>> -> Behvavior<‘b>
let switchB(beh:Behavior<Behavior<'b>>) : Behavior<'b> =
```

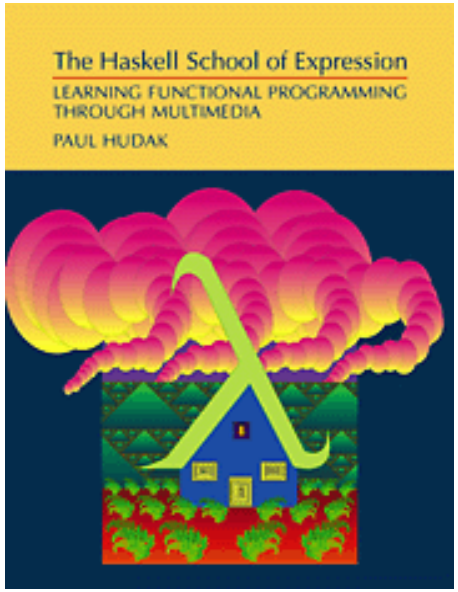
```
let s1 = MkStream [([0],’a’), ([1], ‘b’), ([2], ‘c’)]
let s2 = MkStream [([0], ‘W’), ([1], ‘X’), ([2], ‘Y’)]
// hold :: ‘a -> ‘a Event -> ‘a Behavior
let c = hold s1 (MkStream[([1], s2)]) [0]
let s3 = Switch c
```

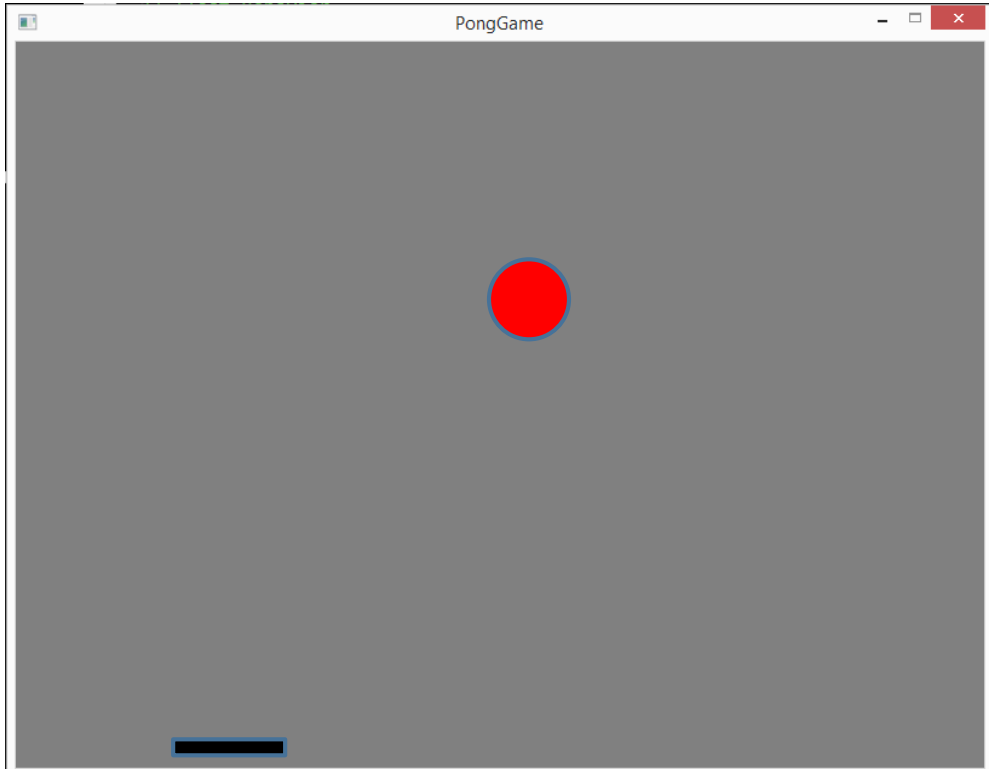


Bank Account

```
type BankAccount() =
    let deposit = Event<int>.newDefault()
    let withdraw = Event<int>.newDefault()
    let bh : Event<int> = merge deposit withdraw
    // Reevaluated for each update
    let bhAcc : Behavior<int> = bh.accum(0, (+))

    member x.Balance with get() = bhAcc.Sample()
    member x.Deposit(amount) = deposit.send(amount)
    member x.Withdraw(amount) = withdraw.send(-amount)
```

Paddle-ball Game with Leap Motion






DEMO

Reactive Programming and Event Processing in F#

Event Processing in F#

SAY GOOD BYE TO THE CALLBACK HELL

Data Processing in F#

```
let myList = [1;2;3]

let myList = List.Cons(1, List.Cons(2, List.Cons(3,
List.Empty)))
```

Data Processing in F#

```
let myList = [1;2;3]

let myList = List.Cons(1, List.Cons(2, List.Cons(3, List.Empty)))

let rec filter lst pred acc =
      match lst with
      | [] -> acc
      | h::t -> if pred h then filter t pred (h::acc)
                else filter t pred acc 2)
filter myList (fun x -> x % 2 = 0) []
```

Data Processing in F#

```
let myList = [1;2;3]

let myList = List.Cons(1, List.Cons(2, List.Cons(3, List.Empty)))
```

```
myList |> List.filter(fun x -> x % 2 = 0)

myList |> List.map(fun x -> x * 2)
```

Data Processing in F#

```
let maxLen (text:string list) =
        text
        |> Seq.map (fun s -> s.Length)
        |> Seq.reduce max


 let searchForPosOrNegWords (text:string list) =
        text
        |> Seq.fold(fun s word -> Set.add word s) Set.empty
        |> Seq.map(fun word ->
             if Set.contains word positive then 1
             elif Set.contains word negative then -1
             else 0)
        |> Seq.sum
```

Event Processing in F#

```
let twitter = Twitter.AuthenticateAppOnly(key, secret)
let sample = twitter.Streaming.SampleTweets()

sample.TweetReceived
    |> Event.filter(fun tweet -> tweet.lang = "en")
    |> Event.choose(fun tweet -> tweet.Text) // Text is option
    |> Event.map(fun text ->
                    if Set.contains text positive then 1
                    elif Set.contains text negative then -1
                    else 0)
    |> Event.scan (+)
    |> Event.add(fun n -> printfn "Mood=%d" n)
```

Few Event HOF

```
Event.map    : ('T -> 'R)   -> IEvent<'T> -> IEvent<'R>
Event.filter : ('T -> bool) -> IEvent<'T> -> IEvent<'T>
Event.add    : ('T -> unit) -> IEvent<'Del,'T> -> unit
Event.merge  : IEvent<'T> -> IEvent<'T> -> IEvent<'T>
Event.scan   : ('St -> 'T -> 'St) -> 'St -> IEvent<'T> ->
                                                IEvent<'St>
```

Observable in F#

- Event<'T> interface inherits from IObservable<'T>
  - We can use the same standard F# Events functions for working with Observable

```
Observable.filter : ('T -> bool) -> IObservable<'T> -> IObservable<'T>
Observable.map    : ('T -> 'R)   -> IObservable<'T> -> IObservable<'R>
Observable.add    : ('T -> unit)   -> IObservable<'T> -> unit
Observable.merge  : IObservable<'T> -> IObservable<'T> -> IObservable<'T>


      Observable.subscribe  : IObservable<'T> -> IDisposable
```

IObservable is the Dual of IEnumerable

```
type IEnumerable<'a> =
    interface IEnumerable

    abstract GetEnumerator : IEnumerator<'a>

type IEnumerator<'a> =
    interface IDisposable
    interface IEnumerator

    abstract Current : 'a with get
    abstract MoveNext : unit -> bool
```

```
type IObservable<'a> =
    abstract Subscribe : IObserver<'a>
                         -> IDisposable

type IObserver<'a> =
    abstract OnNext : 'a with set
    abstract OnCompleted : bool -> unit
    abstract OnError : Exception -> unit
```

Reversing arrows
The input becomes output and <->

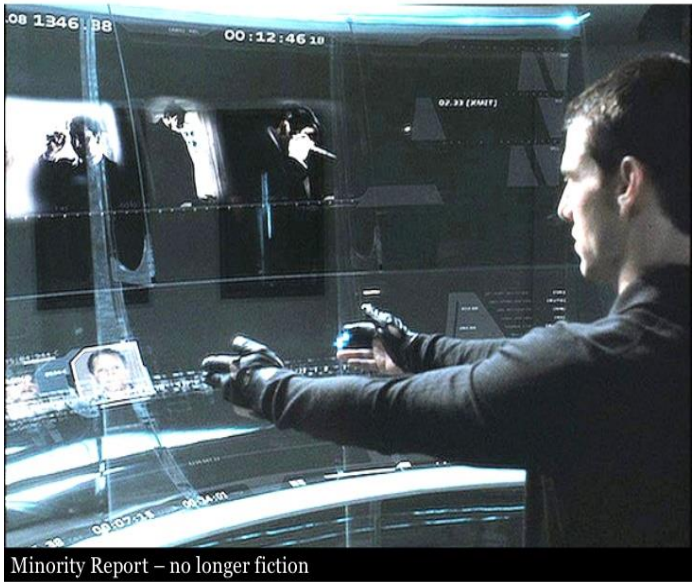
IObserver & IObservable

```
let observable = { new IObservable<string> with
                      member x.Subscribe(observer:IObserver<string>) =
                         { new IDisposable with
                              member x.Dispose() = ()
                              }
                         }

let observer = { new IObserver<string> with
                    member x.OnNext(value) = ()
                    member x.OnCompleted() = ()
                    member x.OnError(exn) = () }
```


DEMO

Natural User Interface

Natural User Interafaces



Minority Report – no longer fiction

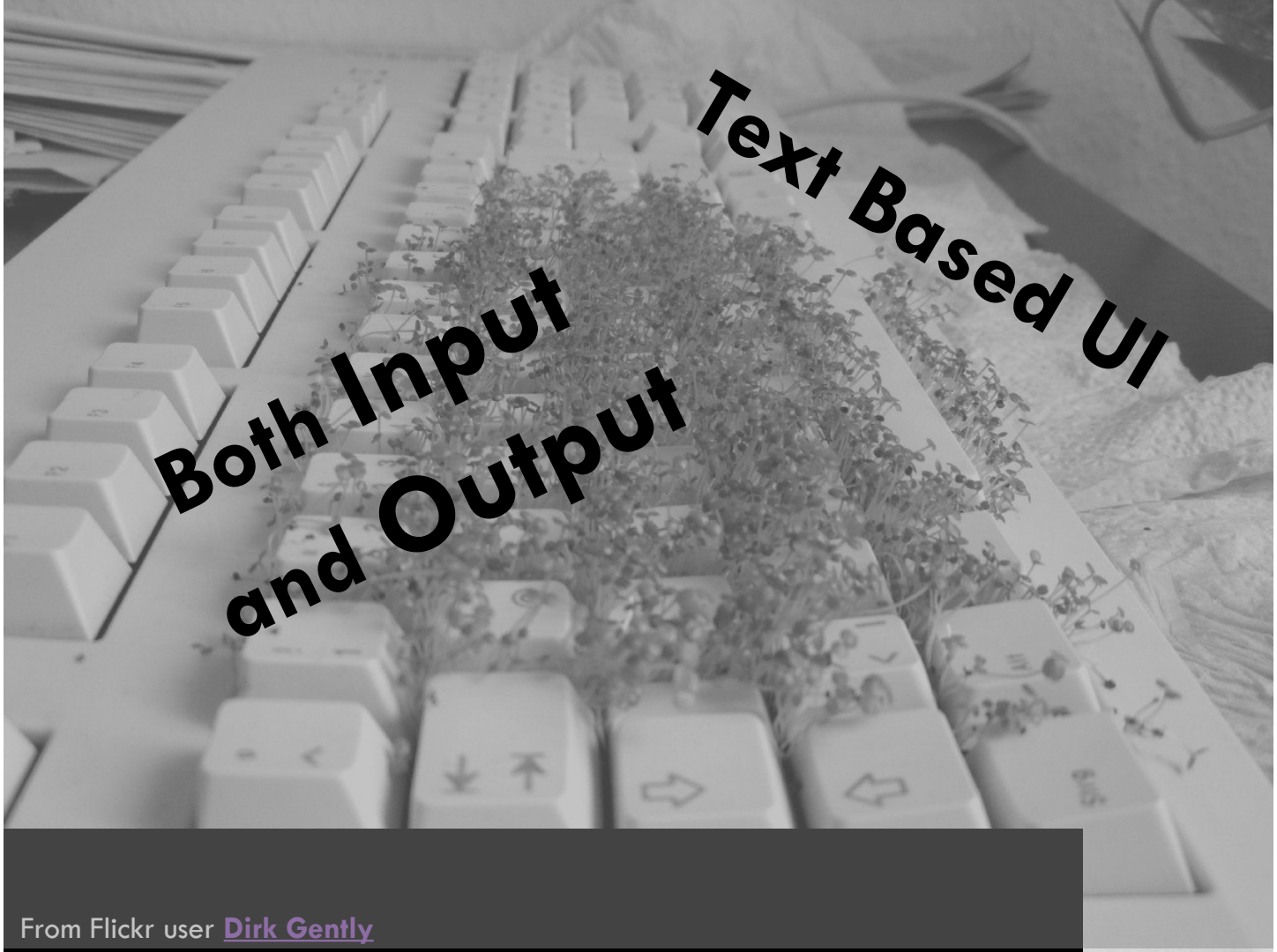
NUI Evolution


Microsoft Word - Document1
File Edit View Insert Format Tools Table Window Help
100%
Draw AutoShapes
Normal Courier 10
B I U


KINECT

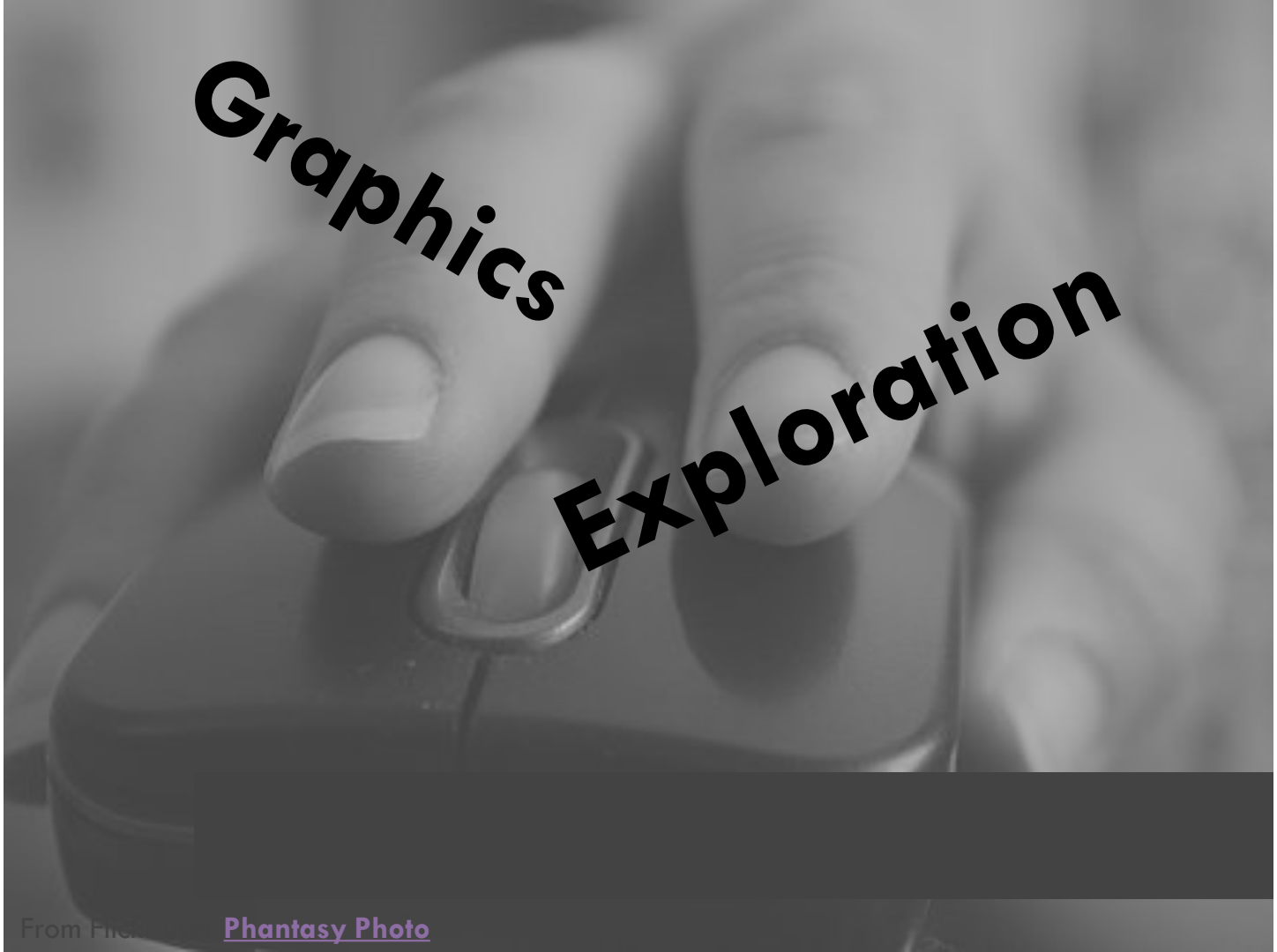
Natural User Interafaces

From Flickr user Dirk Gently

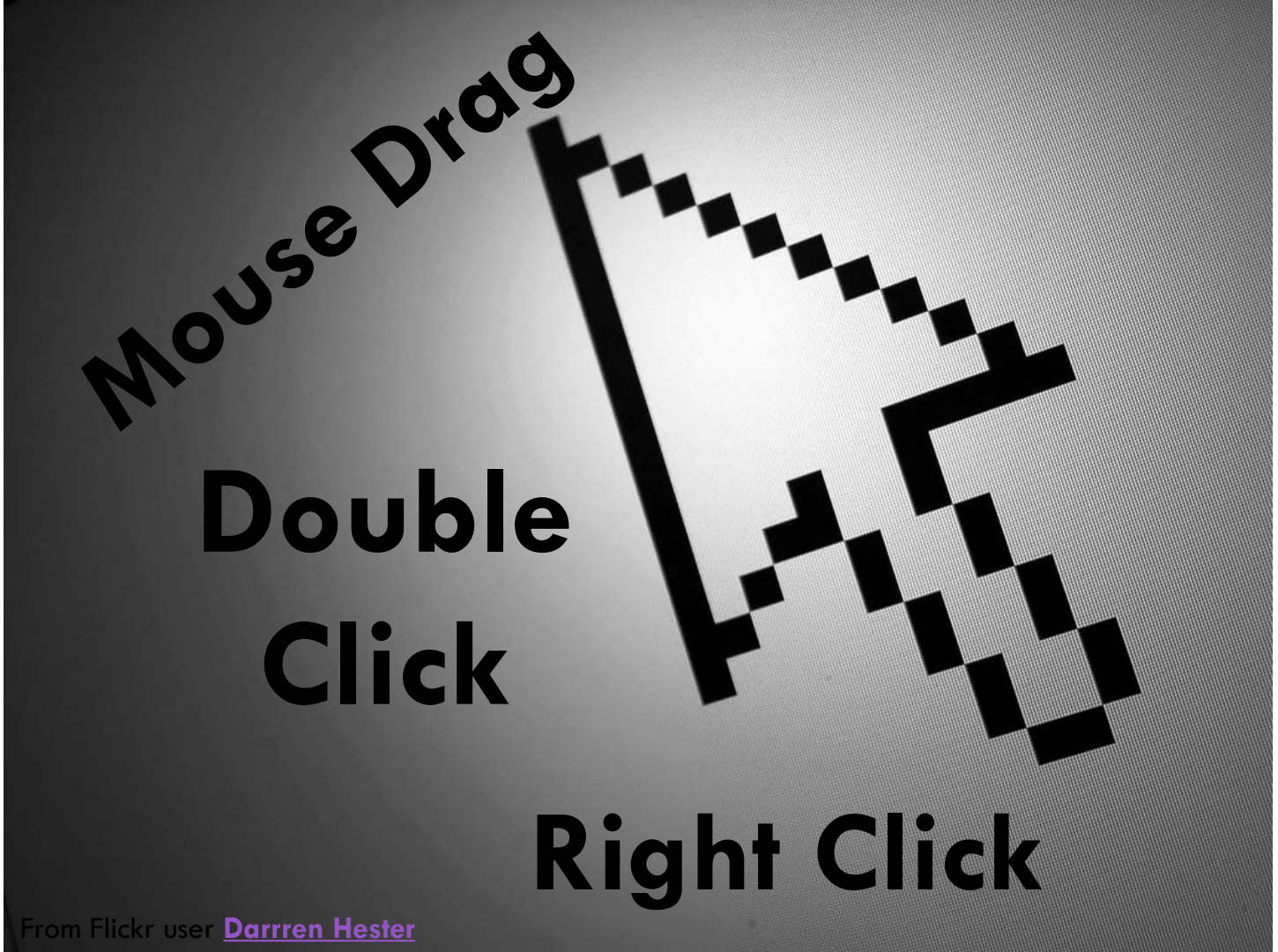


From Flickr user Dirk Gently

Phantasy Photo



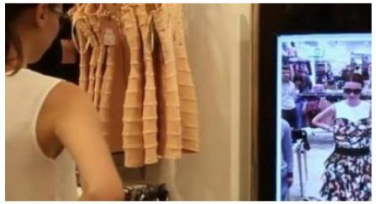
From Flickr: Phantasy Photo



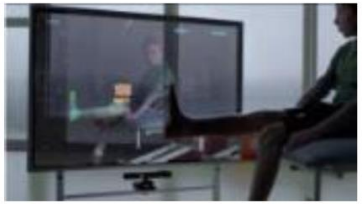
From Flickr user [Darrren Hester]

NUI User Cases

RETAIL



THERAPY



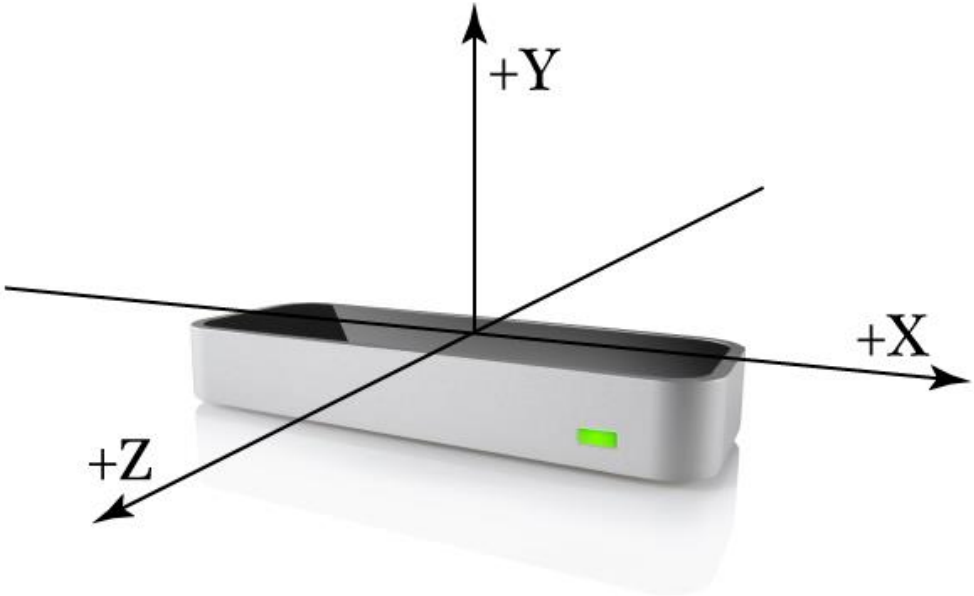
HEALTHCARE

EDUCATION

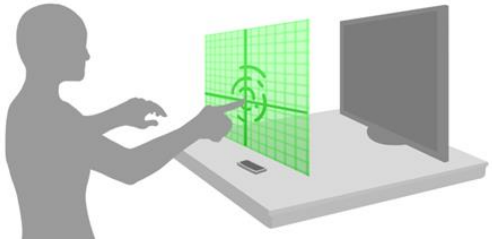
TRAINING

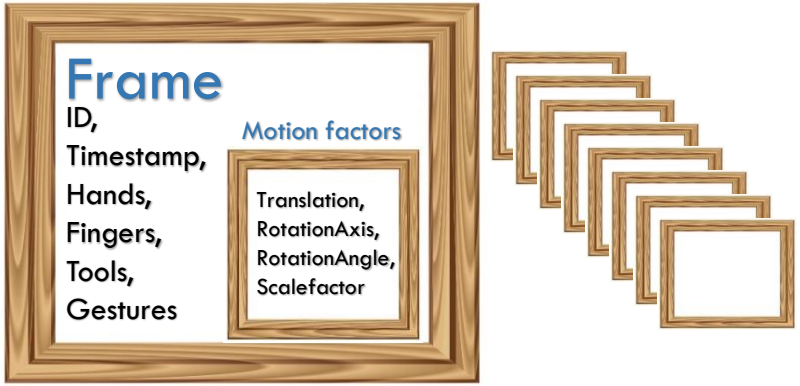

NUI
WARS
LEAP
Myo
STRUCTUR
SENS
intel REALSENSE Develope

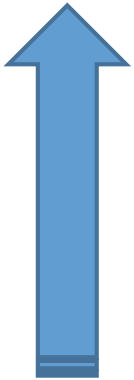
Leap Motion Sensor



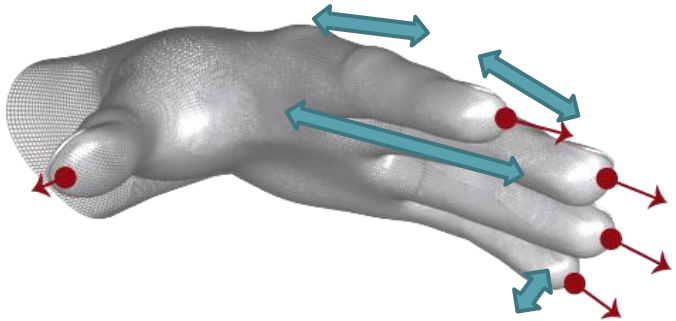
“In just one hand, you have 29 bones, 29 joints, 123 ligaments, 48 nerves, and 30 arteries. That’s sophisticated, complicated, and amazing technology (times two). Yet it feels effortless. The Leap Motion Controller has come really close to figuring it all out.”







```
Hand has 5 fingers, average finger tip position: (8.05207, 69.3328, -17.8252)
Frame id: 2222, timestamp: 10401134, hands: 1, fingers: 5, tools: 0, gestures: 0

Hand has 5 fingers, average finger tip position: (8.29357, 69.3763, -17.896)
Frame id: 2226, timestamp: 10419846, hands: 1, fingers: 5, tools: 0, gestures: 0

Hand has 5 fingers, average finger tip position: (8.44423, 69.4465, -17.9379)
Frame id: 2230, timestamp: 10438557, hands: 1, fingers: 5, tools: 0, gestures: 0

Hand has 5 fingers, average finger tip position: (8.4602, 69.4943, -17.9635)
Frame id: 2234, timestamp: 10457268, hands: 1, fingers: 5, tools: 0, gestures: 0

Hand has 5 fingers, average finger tip position: (8.36989, 69.5165, -17.9892)
Frame id: 2238, timestamp: 10475979, hands: 1, fingers: 5, tools: 0, gestures: 0

Hand has 5 fingers, average finger tip position: (8.19734, 69.4871, -18.0453)
Frame id: 2242, timestamp: 10494690, hands: 1, fingers: 5, tools: 0, gestures: 0

Hand has 5 fingers, average finger tip position: (7.98751, 69.4126, -18.1288)
Frame id: 2246, timestamp: 10513402, hands: 1, fingers: 5, tools: 0, gestures: 0

Hand has 5 fingers, average finger tip position: (7.73525, 69.3156, -18.2251)
Frame id: 2250, timestamp: 10532113, hands: 1, fingers: 5, tools: 0, gestures: 0
```



LEAP & Obserbales – "*Frame*" per sec

```
member x. FramePerSecond() =
         x.Frames()
        |> Observable.timestamp()
        |> Observable.buffer(2)
        |> Observable.subscribe(fun o ->
            let seconds = (o.[1].Timestamp -
                           o.[0].Timestamp).TotalSeconds
            let fps = int(1.0 / seconds)
            printfn "Frame per second %d" fps)
```


DEMO

Summary

- True FRP is about dynamic evolving values over time
  - Precise, simple denotation. (Elegant & rigorous.) Continuous time. (Natural & composable.)
- FRP provide a declarative and elegant programming style for animation, 2D and 3D graphic. IMO - FRP will influence future NUI studies
- F# Observables raise the level abstraction for manipulating stream of events as a chained series of steps

  (Good Bye Callback-Hell)
- NUI is the next generation of UI and it must be explored
- Devices like Leap & Kinect are affordable and fun to play with! Read previous point!


That's all Folks!

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

-- Edsger Dijkstra

How to reach me

github.com/rikace/Presentations/FRP-NUI

@TRikace tericcardo@gmail.com