functional
3D game design

James Reeves
why a game?
why clojure?
You cannot correctly represent change without immutability

Rich Hickey
concurrency
concurrency
concurrency
concurrency
can functional programming be performant?
Reliability

- Brute Force
- Caching
- Mutability
When in doubt, use brute force.

Ken Thompson
### Performance

**assoc on 1000 element map**

<table>
<thead>
<tr>
<th>Method</th>
<th>Data Structure</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute Force</td>
<td>PersistentArrayMap</td>
<td>4600ns</td>
</tr>
<tr>
<td>Caching</td>
<td>PersistentHashMap</td>
<td>180ns</td>
</tr>
<tr>
<td>Mutability</td>
<td>TransientHashMap</td>
<td>85ns</td>
</tr>
</tbody>
</table>
caching is often good enough
16ms per frame

...is actually a sizable chunk of time
caveats

- Performance can't be eyeballed
- I/O typically too costly
- Hard time limit
3D space
vector  translation in space

#math/vector  [0 0 0]

quaternion  rotation in space

#math/quaternion  [0 0 0 1]
near
FOV
far
material
data

camera : location : rotation

frustum : fov : near : far

mesh : location : rotation : scale : vertices : tex-coords : indexes

material : definition : color-map
game loop
user input → ? → display

other I/O
(system time, resources, etc.)

audio
abstractions

display  series of frames over time

frame    snapshot of a changing value

➡️ reference
(defn display []
  (loop []
    (render @scene)
    (recur))))
abstractions

user input series of discrete events

event immutable value

→ channel
(def events (chan))
user input → display

channel ↔ reference
option 1

callbacks + mutable state
mutable state

(def events (chan))
(def world (atom {}))
(def callbacks (atom #{}))
callbacks

(go (loop []
  (when-let [evt (<! events)]
    (doseq [cb callbacks]
      (swap! world cb evt))
    (recur))))
so what's the problem?
it's complex
it's complex

- any callback can change any part of the world
- few constraints on mutation
- no isolation between components
option II

functional
reactive programming
reagi introduces two new reference types
behavior  continuous change

events  discrete changes
behavior

wraps a zero argument function

(r/behavior-call #(+ 1 1))
(r/behavior (+ 1 1))
behavior

most useful when dependent on time

(def time
  (behavior (/ (System/nanoTime) 1e9))

(defn delta []
  (let [t0 @time]
    (behavior (- @time t0))))
events

wraps an asynchronous channel

(def nums (r/events))

(r/push! evts 1)

@evts ;; => 1
events

seq-like transform functions

(def nums (r/events))
(def incs (r/map inc nums))
(r/push! evts 2)
@incs ;; => 3
feedback

(r/reduce
  (fn [pos v] (+ pos v))
init-position
velocities)
feedback

(r/reduce
  (fn [pos v]
   (let [p @pos]
     (r/behavior (+ p v))))))
(r/behavior init-position) velocities)
feedback

(r/reduce
  (fn [pos v]
    (let [p @pos, t (r/delta)]
      (r/behavior (+ p (* v t))))))
(r/behavior init-position)
velocities)
example

position

velocities

\[
\begin{bmatrix}
1 & 0 \\
0 & 1 \\
-1 & 0 \\
0 & -1
\end{bmatrix}
\]
refactor

(r/reduce
  (fn [pos v]
    (let [p @pos, t (r/delta)]
      (r/behavior (+ p (* v t))))))
  (r/behavior init-position)
velocities)
(defn graph [f & args]
  (fn [pos]
    (let [p @pos, t (r/delta)]
      (r/behavior (apply f @t p args))))

(defn linear [t p v]
  (+ p (* v t)))
refactor

(->> velocities
  (r/map (fn [v] (graph linear v))))
(r/accum (r/behavior init-pos))
live coding
libraries

reagi   FRP for clojure
euclidean   vector and quaternion math
crumpets   color representation
criterium   benchmarking
questions?