Quil  http://quil.info/

Clojure/ClojureScript interactive animation library

Based on Processing
   Software sketchbook used to teach programming to visual artists

LightTable instructions

https://github.com/quil/quil/wiki/Dynamic-Workflow-%28for-LightTable%29
(ns quil-test.simple-example
  (:require [quil.core :as q]))

(defn setup []
  (q/frame-rate 2))  ; draw 2 frames/second

(defn draw-state []
  (q/line 0 0 100 0)  ; x1 y1 x2 y2
  (q/line 100 0 100 50)
  (q/rect 100 100 50 20) ; x1 y1 (upper right) width height
  (q/ellipse 100 200 50 50))  ; x1 y1 (center) width height

(q/defsketch quil-test
  :title "Lines"
  :size [300 300]
  :setup setup       ; setup function
  :draw draw-state   ; draw function
  :features [:keep-on-top])
(ns quil-test.simple-example
 (:require [quil.core :as q]))

(defn setup []
  (q/frame-rate 2))

(defn draw-state []
  (q/fill 0 255 0) ; rgb color for fill
  (q/stroke 255 0 0) ; line color
  (q/line 0 0 100 0)
  (q/line 100 0 100 50)
  (q/rect 100 100 50 20)
  (q/ellipse 100 200 50 50))

(q/defsketch quil-test
  :title "Lines"
  :size [300 300]
  :setup setup
  :draw draw-state
  :features [:keep-on-top])
Basic Functions

defsketch
    Defines and starts a sketch
    Many option see doc on sketch

draw
    Function to draw the sketch

update
    Function called just before draw
    Use to update state

setup
    Called once
    Setup and initialize state
Print does not work

(defn draw-state []
  (print "In draw")
  (q/line 0 0 100 0))

(q/defsketch quil-test
  :title "Lines"
  :size [300 300]
  :setup setup
  :draw draw-state
  :features [:keep-on-top])

Draw-state run in other thread
Will not see output
Writing to a file

(ns quil-test.simple-example
 (:require [quil.core :as q]))

(defn setup []
  (q/frame-rate 10))

(defn log 
  [& args]
  (spit "log.txt" (str args "\n") :append true))

(defn draw-state []
  (log "this is frame" (q/frame-count))
  (q/line 0 0 100 0))

(q/defsketch quil-test
  :title "Lines"
  :size [300 300]
  :setup setup
  :draw draw-state
  :features [:keep-on-top])
(def y-value (atom 0))

(defn setup []
  (q/frame-rate 60)
  (reset! y-value 0))

(defn draw-state []
  (q/line 0 0 150 @y-value)
  (swap! y-value inc))

(q/defsketch quil-test
  :title "Lines"
  :size [300 300]
  :setup setup
  :draw draw-state
  :features [:keep-on-top])
Clearing the Screen

(def y-value (atom 0))

(defn setup []
  (q/frame-rate 60)
  (reset! y-value 0))

(defn draw-state []
  (q/background 240) ; Clear screen set background color
  (q/line 0 0 150 @y-value)
  (swap! y-value inc))

(q/defsketch quil-test
  :title "Lines"
  :size [300 300]
  :setup setup
  :draw draw-state
  :features [:keep-on-top])
(defn setup []
  (q/frame-rate 2))

(defn draw-state []
  (q/background 240)
  (doseq [size [150 100 50 30 20 10]]
    (q/ellipse 150 150 size size)))

(q/defsketch quil-test
  :title "Lines"
  :size [300 300]
  :setup setup
  :draw draw-state
  :features [:keep-on-top])
Translate

(defn setup []
  (q/frame-rate 2))

(defn draw-state []
  (q/background 240)
  (q/translate
    (/ (q/width) 2)
    (/ (q/height) 2))
  (q/ellipse 0 0 50 50))

(q/defsketch quil-test
  :title "Lines"
  :size [300 300]
  :setup setup
  :draw draw-state
  :features [:keep-on-top])
Translates Add up

(defn setup []
  (q/frame-rate 2))

(defn draw-state []
  (q/background 240)
  (q/translate 50 50)
  (q/ellipse 0 0 50 50)
  (q/translate 50 50)
  (q/line 0 0 100 0))

(q/defsketch quil-test
  :title "Lines"
  :size [300 300]
  :setup setup
  :draw draw-state
  :features [:keep-on-top])

Translations are reset when draw function
is called again
(defn draw-state []
  (q/background 240)
  (q/translate 50 50)
  (q/ellipse 0 0 50 50)
  (q/reset-matrix)
  (q/translate 50 50)
  (q/line 0 0 100 0))

(defn draw-state []
  (q/background 240)
  (q/push-matrix)
  (q/translate 50 50)
  (q/ellipse 0 0 50 50)
  (q/pop-matrix)
  (q/translate 50 50)
  (q/line 0 0 100 0))

(defn draw-state []
  (q/background 240)
  (q/with-translation [50 50]
    (q/ellipse 0 0 50 50))
  (q/translate 50 50)
  (q/line 0 0 100 0))
Translate Fun

(defn draw-state []
  (q/background 240)
  (q/fill 0) ; set text to black
  (q/text "Translate/Rotation Fun" 20 20)
  (q/translate
    (/ (q/width) 2)
    (/ (q/height) 2))
  (q/line 0 0 100 0)
  (q/translate 100 0)
  (q/rotate (q/radians 90))
  (q/line 0 0 100 0)
  (q/translate 100 0)
  (q/rotate (q/radians 135))
  (q/line 0 0 100 0))
(def message (atom "No keyboard"))

(defn keyboard-action
  []
  (let [key (q/key-as-keyword)]
    (reset! message (str "Key " key))))

(defn setup []
  (q/frame-rate 20))

(defn draw-state []
  (q/background 240)
  (q/fill 0)
  (q/scale 2.5)
  (q/translate 30 30)
  (q/text @message 0 0)
  (q/text (str (q/frame-count)) 0 20))

(q/defsketch quil-test
  :title "Lines"
  :size [300 300]
  :setup setup
  :draw draw-state
  :key-pressed keyboard-action
  :features [:keep-on-top])
(def message (atom "No keyboard"))

(defn keyboard-action
  []
  (let [key (q/key-as-keyword)]
    (reset! message (str "Key " key))
    (if (= key :r)
      (q/start-loop)
      (q/redraw))))

(defn setup []
  (q/frame-rate 20)
  (q/no-loop))

(defn draw-state []
  (q/background 240)
  (q/fill 0)
  (q/scale 2.5)
  (q/translate 30 30)
  (q/text @message 0 0)
  (q/text (str (q/frame-count)) 0 20))

(q/defsketch quil-test
  :title "Lines"
  :size [300 300]
  :setup setup
  :draw draw-state
  :key-pressed keyboard-action
  :features [:keep-on-top])
Design Patterns
# The Functional Pattern Joke

<table>
<thead>
<tr>
<th>OO Pattern</th>
<th>Functional Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapter</td>
<td>Functions</td>
</tr>
<tr>
<td>Bridge</td>
<td>Functions</td>
</tr>
<tr>
<td>Chain of responsibility</td>
<td>Functions</td>
</tr>
<tr>
<td>Command</td>
<td>Functions</td>
</tr>
<tr>
<td>Composite</td>
<td>Functions</td>
</tr>
<tr>
<td>Decorator</td>
<td>Just Functions</td>
</tr>
<tr>
<td>Facade</td>
<td>Functions</td>
</tr>
<tr>
<td>Flyweight</td>
<td>Functions</td>
</tr>
<tr>
<td>Mediator</td>
<td>Functions</td>
</tr>
<tr>
<td>Observer</td>
<td>Functions</td>
</tr>
<tr>
<td>Strategy</td>
<td>Functions</td>
</tr>
<tr>
<td>Template method</td>
<td>Still Just Functions</td>
</tr>
</tbody>
</table>
OO data & Functional Data

Person
   First name
   Last name
   age
   List of phone numbers
public class Person {
    private int age;
    private String firstName;
    private String lastName;
    private ArrayList phoneNumbers;

    public Person(String first, String last, int age) {
        this.firstName = first;
        this.lastName = last;
        this.age = age;
        phoneNumbers = new ArrayList();
    }

    public int age() { return age; }
    public void age(int newAge) { age = newAge; }

    etc.
Sample Use

Person example = new Person("Sachin", "Tendulkar", 40);

int lastYearsAge = example.age();
example.age(41);

age gives access to the age value in a person

age is like a key in a hash table

{:first-name "Sachin"
 :last-name "Tendulkar"
 :age 40
 :phone-numbers [] }
# Converting Objects to Clojure data

<table>
<thead>
<tr>
<th>Class</th>
<th>Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field name</td>
<td>keyword as key in map</td>
</tr>
<tr>
<td>new Person(&quot;Sachin&quot;, &quot;Tendulkar&quot;, 40);</td>
<td>{:first-name &quot;Sachin&quot;</td>
</tr>
<tr>
<td></td>
<td>:last-name &quot;Tendulkar&quot;</td>
</tr>
<tr>
<td></td>
<td>:age 40</td>
</tr>
<tr>
<td></td>
<td>:phone-numbers [] }</td>
</tr>
</tbody>
</table>
Memento

Store an object's internal state, so the object can be restored to this state later without violating encapsulation

undo, rollbacks

Only originator:

Can access Memento’s get/set state methods
Create Memento
Copying Issues

Shallow Copy Verse Deep Copy

Original Objects

Shallow Copy

Sunday, October 25, 15
Memento Pattern & Functional Programming

Immutable data
   No need to copy the data
   Just save current data

(def state-history (atom []))

(defn add-state
  [state]
  (swap! state-history conj state))

(defn previous-state
  []
  (let [last-state (last @state-history)]
    (swap! state-history pop)
    last-state))
Command Pattern

Encapsulates a request as an object
Example

Button in a GUI

When press button remove the current selected row of table
public class RemoveRowCommand extends Command {
    private Table target;

    public RemoveRowCommand(Table target) {
        this.target = target;
    }

    public execute() {
        int selection = target.getSelection();
        target.removeRow(selection);
    }
}

(defn remove-row-command
    [table]
    (fn [] (remove-row table)))
Using the Command

Button removeSelection = new Button();
Command removeRow = new RemoveRowCommand(ourTable);
removeSelection.action(removeRow);

Button class is written to call execute when button is pressed
Quil Example

(q/defsketch quil-test
  :title "Lines"
  :size [300 300]
  :setup setup
  :draw draw-state
  :key-pressed keyboard-action
  :features [:keep-on-top])
Command Pattern Supports Undo

Modify class
  Add undo method

Keep stack of past commands

Undo
  Pop the stack
  Call undo on element removed from stack
public class RemoveRowCommand extends Command {
    private Table target;
    private int rowIndex;
    private Row removedRow;

    public RemoveRowCommand(Table target) {
        this.target = target;
    }

    public void execute() {
        rowIndex = target.getSelection();
        removedRow = target.getRow(rowIndex);
        target.removeRow(rowIndex);
    }

    public void undo() {
        if (removedRow == nil) return;
        target.addRow(removedRow, rowIndex);
        removedRow = nil;
    }
}
Undo - Using maps & multimethods

Store the data needed for undo in a map

Use multimethod to perform undo
Undo - Add Subtract Example

Data needed to undo addition
  Current value
  Value added

  {:command :add :value 10 :amount 2}

Data needed to undo subtraction
  Current value
  Value subtracted

  {:command :subtraction :value 10 :amount 2}
The Multimethod

(defmulti undo :command)
(defmethod undo :add
  [{:keys [value amount]}]
  (- value amount))
(defmethod undo :subtract
  [{:keys [value amount]}]
  (+ value amount))

(def example
  {:command :add :value 10 :amount 2})
(undo example)
Command History

(def command-history (atom []))

(defn save-command
  [command]
  (swap! command-history conj command))

(defn previous-command
  []
  (let [last-command (last @command-history)]
    (swap! command-history pop)
    last-command))
Memento Pattern

Idea - save current state

OO implementation
Copy objects
Deal with information hiding

Functional implementation
Just save current state
Command Pattern

Idea: Save data needed to perform an operation

OO Implementation

Separate class for data

Interface for executing method

Functional implementation

Use map for the data
What is the Pattern?

The idea?

The implementation?

What is important?
Iterator Pattern

Provide a way to access the elements of a collection sequentially without exposing its underlying representation

```java
LinkedList<String> strings = new LinkedList<String>();

for (String element : strings) {
    if (element.size % 2 == 0)
        System.out.println(element);
}

Iterator<String> list = strings.iterator();
while (list.hasNext()){
    String element = list.next();
    if (element.size % 2 == 0)
        System.out.println(element);
}
```
Iterator Pattern - Clojure

sequences
Strategy Pattern

defines a family of algorithms, 
encapsulates each algorithm, and 
makes the algorithms interchangeable within that family.
class OrderableList {
    private Object[] elements;
    private Algorithm orderer;

    public OrderableList(Algorithm x) {
        orderer = x;
    }

    public void add(Object element) {
        elements = orderer.add(elements, element);
    }
}
Clojure Example

(sort-by last {:b 1 :c 3 :a 2})

Just pass in a function