```
(ns c2.geo.t-core
  (:use midje.sweet
        c2.geo.core)

(def geojson {:type "MultiPolygon", :coordinates
[[[[-89.54221,41.90248],[-89.53985,41.90259],[-89.53815,41.90271],[-89.53576,41.90268],[-89.5
139,41.90227],[-89.6298,41.90175],[-89.58124,41.90213],[-89.57225,41.90217],[-89.56239,41.9
0222],[-89.54221,41.90248]

(fact "geojson should convert to svg string"
  (geo->svg geojson) =>
"M-89.54221,41.90248L-89.53985,41.90259L-89.53815,41.90271L-89.53576,41.90268L-89.51
39,41.90227L-89.6298,41.90175L-89.58124,41.90213L-89.57225,41.90217L-89.56239,41.9022
2L-89.54221,41.90248Z")
  (fact "haversine should calculate distance"
  (haversine {:lat 55.449780 :lon 11.823392} {:lat 55.451021 :lon 11.803007}) => (roughly
1.2943206245538126)

(ns c2.util
  (:require [singult.core :as singult]))

(defn ->coll
  "Convert something into a collection, if it's not already."
  [x]
  (if (coll? x) x [x]))

(ns c2.dom
  (:refer-clojure :exclude [val])
  (:use-macros [c2.util :only [p pp timeout bind!]])
  (:require [clojure.string :as string]
            [singult.core :as singult]
            [goog.dom :as gdom]
            [goog.dom.forms :as gforms]
            [goog.dom.classes :as gclasses]
            [goog.style :as gstyle]))

;;Going down a terrible, terrible road here...
(js* "Element.prototype.matchesSelector = Element.prototype.webkitMatchesSelector ||
Element.prototype.mozMatchesSelector || Element.prototype.msMatchesSelector ||
Element.prototype.oMatchesSelector")

;;Seq over native JavaScript node collections
(when (js* "typeof NodeList != \"undefined\"")
  (extend-type js/NodeList
    ISeqable
    (-seq [array] (array-seq array 0))))

(extend-type js/HTMLCollection
```

```
  ISeqable
  (-seq [array] (array-seq array 0)))

(declare select)

(defprotocol IDom
  (->dom [x] "Converts x to a live DOM node"))

(extend-protocol IDom
  string
  (->dom [selector] (select selector))

  PersistentVector
  (->dom [v] (singult/render v)))

(when (js* "typeof Node != \"undefined\"")
  (extend-type js/Node
    IDom
    (->dom [node] node)))

(defn select
  "Select a single DOM node via CSS selector, optionally scoped by second arg."
  ([selector] (.querySelector js/document selector))
  ([selector container] (.querySelector (->dom container) selector)))
(defn select-all
  "Like select, but returns a collection of nodes."
  ([selector] (.querySelectorAll js/document selector))
  ([selector container] (.querySelectorAll (->dom container) selector)))

(defn matches-selector?
  "Does live `node` match CSS `selector`?"
  [node selector]
  (when node
    (.matchesSelector node selector)))

(defn children
  "Return the children of a live DOM element."
  [node]
  (when node
    (.-children (->dom node))))

(defn parent
  "Return parent of a live DOM node."
  [node]
  (when parent
    (.-parentNode (->dom node))))

(defn append!
```

```
  "Make element last child of container.
   Returns live child."
  [container el]
  (let [el (->dom el)]
    (gdom/appendChild (->dom container) el)
    el))

(defn prepend!
  "Make element first child of container.
   Returns live DOM child."
  [container el]
  (let [el (->dom el)]
    (gdom/insertChildAt (->dom container) el 0)
    el))

(defn remove!
  "Remove element from DOM and return it.
   > *el* CSS selector or live DOM node"
  [el]
  (gdom/removeNode (->dom el)))

(defn replace!
  "Replace live DOM node with a new one, returning the latter.
   > *old* CSS selector or live DOM node
   > *new* CSS selector, live DOM node, or hiccup vector"
  [old new]
  (let [new (->dom new)]
    (gdom/replaceNode new (->dom old))
    new))

(defn style
  "Get or set inline element style.
    `(style el)`               map of inline element styles
    `(style el :keyword)`      value of style :keyword
    `(style el {:keyword val})` sets inline style according to map, returns element
    `(style el :keyword val)`  sets single style, returns element"
  ([el] (throw (js/Error. "TODO: return map of element styles")))
  ([el x]
     (let [el (->dom el)]
       (cond
         (keyword? x) (gstyle/getComputedStyle el (name x))
         (map? x) (do
                    (doseq [[k v] x] (style el k v))
                    el))))
  ([el k v]
     (gstyle/setStyle (->dom el) (name k)
                      (cond
                        (string? v) v
```

```
                    (number? v) (if (#{:height :width :top :left :bottom :right} (keyword k))
                                  (str v "px")
                                  v)))
    el))

(defn attr
  "Get or set element attributes.
   `(attr el)`               map of element attributes
   `(attr el :keyword)`      value of attr :keyword
   `(attr el {:keyword val})` sets element attributes according to map, returns element
   `(attr el :keyword val)`   sets single attr, returns element"
  ([el] (let [attrs (.-attributes (->dom el))]
          (into {} (for [i (range (.-length attrs))]
                     [(keyword  (.-name (aget attrs i)))
                      (.-value (aget attrs i))]))))
  ([el x]
    (let [el (->dom el)]
      (cond
        (keyword? x) (.getAttribute el (name x))
        (map? x) (do (doseq [[k v] x] (attr el k v))
                     el))))
  ([el k v]
    (let [el (->dom el)]
      (if (nil? v)
        (.removeAttribute el (name k))
        (if (= :style k)
          (style el v)
          (.setAttribute el (name k) v)))
      el)))

(defn text
  "Get or set element text, returning element"
  ([el]
    (gdom/getTextContent (->dom el)))
  ([el v]
    (let [el (->dom el)]
      (gdom/setTextContent el v)
      el)))

(defn val
  "Get or set element value."
  ([el]
    (gforms/getValue (->dom el)))
  ([el v]
    (let [el (->dom el)]
      (gforms/setValue el v)
      el)))
```

```
(defn classed!
  "Add or remove `class` to element based on boolean `classed?`, returning element."
  [el class classed?]
  (gclasses/enable (->dom el) (name class) classed?)
  el)

;;TODO: make these kind of shortcuts macros for better performance.
(defn add-class! [el class] (classed! el class true))
(defn remove-class! [el class] (classed! el class false))

;;Call this fn with a fn that should be executed on the next browser animation frame.
(def request-animation-frame
  (or (.-requestAnimationFrame js/window)
      (.-webkitRequest)##
lines (155 sloc)  6.98 KB
;;Collection of helpers for dealing with scalable vector graphics.
;;
;;Coordinates to any fn can be 2-vector `[x y]` or map `{:x x :y y}`.
^:clj (ns c2.svg
        (:use [c2.core :only [unify]]
              [c2.maths :only [Pi Tau radians-per-degree
                               sin cos mean]]))

^:cljs (ns c2.svg
         (:use [c2.core :only [unify]]
               [c2.maths :only [Pi Tau radians-per-degree
                                sin cos mean]])
         (:require [c2.dom :as dom]))

(defn ->xy
  "Ensure that coordinates (potentially map of `{:x :y}`) are a seq or vector pair."
  [coordinates]
  (cond
   (and (sequential? coordinates) (= 2 (count coordinates))) coordinates
   (map? coordinates) [(:x coordinates) (:y coordinates)]))

(defn translate [coordinates]
  (let [[x y] (->xy coordinates)]
    (str "translate(" (float x) "," (float y) ")")))

(defn scale [coordinates]
  (if (number? coordinates)
    (str "scale(" (float coordinates) ")")
    (let [[x y] (->xy coordinates)]
      (str "scale(" (float x) "," (float y) ")"))))

(defn rotate
  ([angle] (rotate angle [0 0]))
```

```
  ([angle coordinates]
     (let [[x y] (->xy coordinates)]
       (str "rotate(" (float angle) "," (float x) "," (float y) ")"))))

(defn ^:cljs get-bounds
  "Returns map of `{:x :y :width :height}` containing SVG element bounding box.
   All coordinates are in userspace. Ref [SVG
spec](http://www.w3.org/TR/SVG/types.html#InterfaceSVGLocatable)"
  [$svg-el]
  (let [b (.getBBox $svg-el)]
    {:x (.-x b)
     :y (.-y b)
     :width (.-width b)
     :height (.-height b)}))

(defn transform-to-center
  "Returns a transform string that will scale and center provided element `{:width :height :x :y}`
within container `{:width :height}`."
  [element container]
  (let [{ew :width eh :height x :x y :y} element
        {w :width h :height} container
        s (min (/ h eh) (/ w ew))]
    (str (translate [(- (/ w 2) (* s (/ ew 2)))
                     (- (/ h 2) (* s (/ eh 2)))]);;translate scaled to center
         " " (scale s) ;;scale
         " " (translate [(- x) (- y)]) ;;translate to origin
         )))

(defn ^:cljs transform-to-center!
  "Scales and centers `$svg-el` within its parent SVG container.
   Uses parent's width and height attributes only."
  [$svg-el]
  (let [$svg (.-ownerSVGElement $svg-el)
        t (transform-to-center (get-bounds $svg-el)
                               {:width (js/parseFloat (dom/attr $svg :width))
                                :height (js/parseFloat (dom/attr $svg :height))})]
    (dom/attr $svg-el :transform t)))

(defn axis
  "Returns axis <g> hiccup vector for provided input `scale` and collection of `ticks` (numbers).
   Direction away from the data frame is defined to be positive; use negative margins and widths
to render axis inside of data frame.
   Kwargs:
   > *:orientation* &in; (`:top`, `:bottom`, `:left`, `:right`), where the axis should be relative to the
data frame, defaults to `:left`
   > *:formatter* fn run on tick values, defaults to `str`
   > *:major-tick-width* width of ticks (minor ticks not yet implemented), defaults to 6
   > *:text-margin* distance between axis and start of text, defaults to 9
```

```
  > *:label* axis label, centered on axis; :left and :right orientation labels are rotated by +/- pi/2,
respectively
  > *:label-margin* distance between axis and label, defaults to 28"
  [scale ticks & {:keys [orientation
                         formatter
                         major-tick-width
                         text-margin
                         label
                         label-margin]
                  :or {orientation :left
                       formatter str
                       major-tick-width 6
                       text-margin 9
                       label-margin 28}}]

  (let [[x y x1 x2 y1 y2] (case orientation
                            (:left :right) [:x :y :x1 :x2 :y1 :y2]
                            (:top :bottom) [:y :x :y1 :y2 :x1 :x2])

        parity (case orientation
                 (:left :top) -1
                 (:right :bottom) 1)]

    [:g {:class (str "axis " (name orientation))}
     [:line.rule (apply hash-map (interleave [y1 y2] (:range scale)))]
     [:g.ticks
      ;;Need to weave scale into tick stream so that unify updates nodes when the scale changes.
      (unify (map vector ticks (repeat scale))
             (fn [[d scale]]
               [:g.tick.major-tick {:transform (translate {x 0 y (scale d)})}
                [:text {x (* parity text-margin)} (formatter d)]
                [:line {x1 0 x2 (* parity major-tick-width)}]]))]

     (when label
       [:text.label {:transform (str (translate {x (* parity label-margin)
                                                 y (mean (:range scale))})
                                     " "
                                     (case orientation
                                       :left (rotate -90)
                                       :right (rotate 90)
                                       ""))}
        label])
     ]))

(defn line
  "Return a Hiccup path SVG element with the [x,y] coordinates in the points sequence
connected by lines"
  [points]
```

```
  (let [[[x y] & xs] points]
    [:path {:d (apply str "M" x "," y
                      (for [[x y] xs] (str "L" x "," y)))}]))

(def ArcMax (- Tau 0.0000001))

(defn circle
  "Calculate SVG path data for a circle of `radius` starting at 3 o'clock and sweeping in positive
y."
  ([radius] (circle [0 0] radius))
  ([coordinates radius]
     (let [[x y] (->xy coordinates)]
       (str "M"  (+ x radius) "," y
            "A" (+ x radius) "," (+ y radius) " 0 1,1" (- (+ x radius)) "," y
            "A" (+ x radius) "," (+ y radius) " 0 1,1" (+ x radius) "," y))))

(defn arc
  "Calculate SVG path data for an arc."
  [& {:keys [inner-radius, outer-radius
             start-angle, end-angle, angle-offset]
      :or {inner-radius 0, outer-radius 1
           start-angle 0, end-angle Pi, angle-offset 0}}]
  (let [r0 inner-radius
        r1 outer-radius
        [a0 a1]  (sort [(+ angle-offset start-angle)
                        (+ angle-offset end-angle)])
        da (- a1 a0)
        large-arc-flag (if (< da Pi) "0" "1")

        s0 (sin a0), c0 (cos a0)
        s1 (sin a1), c1 (cos a1)]

    ;;SVG "A" parameters: (rx ry x-axis-rotation large-arc-flag sweep-flag x y)
    ;;see http://www.w3.org/TR/SVG/paths.html#PathData
    (if (>= da ArcMax)
      ;;Then just draw a full annulus
      (str "M0," r1
           "A" r1 "," r1 " 0 1,1 0," (- r1)
           "A" r1 "," r1 " 0 1,1 0," r1
           (if (not= 0 r0) ;;draw inner arc
             (str "M0," r0
                  "A" r0 "," r0 " 0 1,0 0," (- r0)
                  "A" r0 "," r0 " 0 1,0 0," r0))
           "Z")

      ;;Otherwise, draw the wedge
      (str "M" (* r1 c0) "," (* r1 s0)
           "A" r1 "," r1 " 0 " large-arc-flag ",1 " (* r1 c1) "," (* r1 s1) ( WC)22.R
```