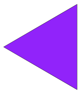
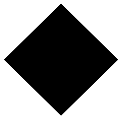
Reference shapes

```
(def purple-triangle
  (poly
        0
        0
        0.75
        3
        {:stroke (p-color 0 0 0),
          :fill (p-color 150 0 255),
          :stroke-weight 1}))
```



```
(def black-square
  (poly
        0
        0
        0.75
        4
        {:stroke (p-color 0),
          :fill (p-color 0)}))
```



patterning.layouts

alt-cols

`(alt-cols n groups1 groups2)`

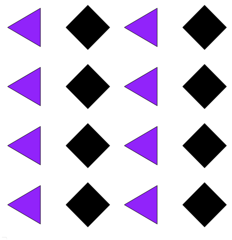
Fills a group-stream with cols from alternative group-streams

alt-cols-grid-layout

`(alt-cols-grid-layout n groups1 groups2)`

Every other column from two streams

```
(alt-cols-grid-layout 4 (repeat purple-triangle) (repeat black-square))
```



alt-rows

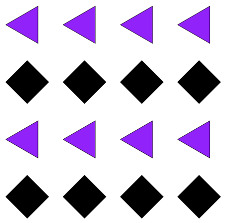
```
(alt-rows n groups1 groups2)
```

Fills a group-stream with rows from alternative group-streams

alt-rows-grid-layout

```
(alt-rows-grid-layout n groups1 groups2)
```

Every other row from two streams



cart

```
(cart colls)
```

Cartesian Product of two collections

check-seq

```
(check-seq n groups1 groups2)
```

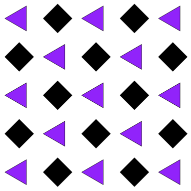
returns the appropriate lazy seq of groups for constructing a checked-layout

checked-layout

```
(checked-layout number groups1 groups2)
```

does checks using grid layout

```
(checked-layout 5 (repeat purple-triangle) (repeat black-square))
```

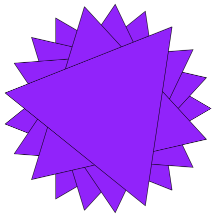


clock-rotate

```
(clock-rotate n group)
```

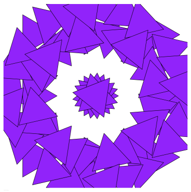
Circular layout. Returns n copies in a rotation

```
(clock-rotate 7 purple-triangle)
```



This didn't do what I expected when I tried to pass a grid-layout to it…todo: figure out how this one works

```
(clock-rotate 7 (grid-layout 3 (repeat purple-triangle)))
```



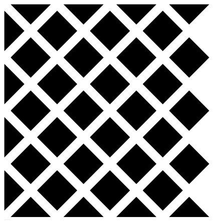
diamond-layout

```
(diamond-layout n groups)
```

Like half-drop

```
(diamond-layout 4 (repeat black-square))
```

Tighter spacing than half-drop, rather than rows and columns being placed next to each other they are interleaved.



diamond-layout-positions

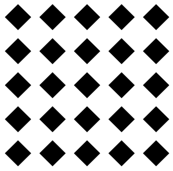
```
(diamond-layout-positions number)
```

Diamond grid, actually created like a half-drop

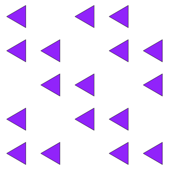
drop-every

```
(drop-every n xs)
```

```
(grid-layout 5 (drop-every 2 (cycle [black-square purple-triangle])))
```



```
(drop-every 3 (grid-layout 5 (repeat purple-triangle)))
```



flower-of-life-positions

```
(flower-of-life-positions r depth [cx cy])
```

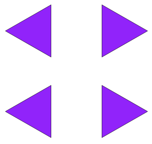
Flower of Life layout ... these are recursive developments of circles

four-mirror

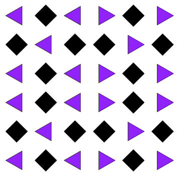
```
(four-mirror group)
```

Four-way mirroring. Returns the group repeated four times reflected vertically and horizontally

```
(four-mirror purple-triangle)
```



```
(four-mirror (checked-layout 3 (repeat purple-triangle) (repeat black-square)))
```

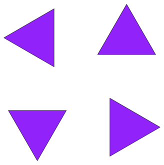


four-round

```
(four-round group)
```

Four squares rotated

```
(four-round purple-triangle)
```

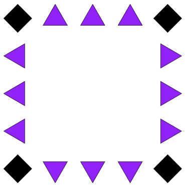


frame

```
(frame grid-size corners edges)
```

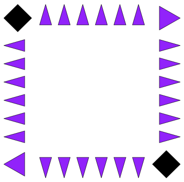
Frames consist of corners and edges.

```
(frame 5 (repeat black-square) (repeat purple-triangle))
```



Note: I was expecting it to treat corners and edges as a list (so you could use, say, a random sequence for edge and corners and have them all be different). Looks like it does this for corners, but not edges. Example:

```
(frame 5 (cycle [black-square purple-triangle]) (cycle [(v-mirror purple-triangle)
purple-triangle]))
```

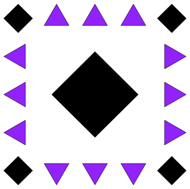


framed

```
(framed grid-size corners edges inner)
```

Puts a frame around the other group

```
(framed 5 (repeat black-square) (repeat purple-triangle) black-square)
```

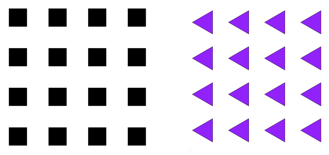


grid-layout

```
(grid-layout n groups)
```

Takes an n and a group-stream and returns items from the group-stream in an n X n grid

```
(grid-layout 4 (repeat shape))
```



grid-layout-positions

```
(grid-layout-positions number)
```

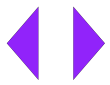
calculates the positions for a grid layout

h-mirror

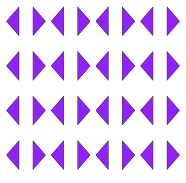
```
(h-mirror group)
```

Reflect horizontally and stretch

```
(h-mirror purple-triangle)
```



```
(grid-layout 4 (repeat (h-mirror shape)))
```

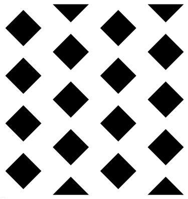


half-drop-grid-layout

```
(half-drop-grid-layout n groups)
```

Like grid but with half-drop

```
(half-drop-grid-layout 4 (repeat black-square))
```



half-drop-grid-layout-positions

`(half-drop-grid-layout-positions number)`

Like a grid but with a half-drop every other column

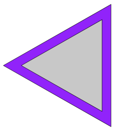
nested-stack

`(nested-stack styles group reducer)`

superimpose smaller copies of a shape

PLACEHOLDER (TODO: figure out styles parameter)

```
(nested-stack {:stroke (p-color 0) :fill (p-color 200)} purple-triangle (fn [x] (* x 0.75)))
```



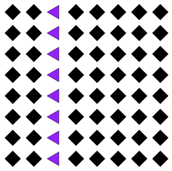
one-col-layout

`(one-col-layout n i groups1 groups2)`

Takes a total number of cols, an index i and two group-streams.
Makes an n X n square where col i is from group-stream2 and everything else is group-stream1

uses one-x-layout with rows

```
(one-col-layout 8 2 (repeat black-square) (repeat purple-triangle))
```



one-row-layout

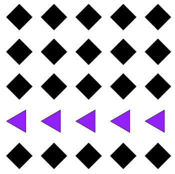
`(one-row-layout n i groups1 groups2)`

Takes a total number of rows, an index i and two group-streams.
Makes an n X n square where row i is from group-stream2 and everything else is group-stream1

uses one-x-layout with rows

```
(one-row-layout 5 3 (repeat black-square) (repeat purple-triangle))
```

n is number of rows/columns, i is row index at which to place the unique row



one-x-layout

`(one-x-layout n i f groups1 groups2)`

```
Takes a total number of rows, an index i and two group-streams.
Makes an n X n square where row or col i is from group-stream2 and everything else is
group-stream1
```

place-groups-at-positions

`(place-groups-at-positions groups positions)`

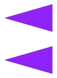
```
Takes a list of groups and a list of positions and puts one of the groups at each position
```

q1-rot-group

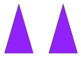
`(q1-rot-group group)`

Used in random-turn-groups.

For reference: `(v-mirror purple-triangle)`



`(q1-rot-group (v-mirror purple-triangle))`

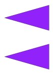


q2-rot-group

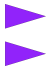
`(q2-rot-group group)`

Used in random-turn-groups.

For reference: `(v-mirror purple-triangle)`



`(q2-rot-group (v-mirror purple-triangle))`



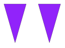
q3-rot-group

```
(q3-rot-group group)
```

Used in random-turn-groups.

For reference: `(v-mirror purple-triangle)`

```
(q3-rot-group (v-mirror purple-triangle))
```

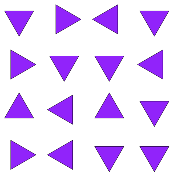


random-grid-layout

```
(random-grid-layout n groups)
```

Takes a group and returns a grid with random quarter rotations

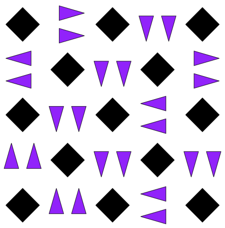
```
(random-grid-layout 4 (repeat purple-triangle))
```



random-turn-groups

```
(random-turn-groups groups)
```

```
(checked-layout 5 (repeat black-square) (random-turn-groups (repeat (v-mirror
purple-triangle))))
```

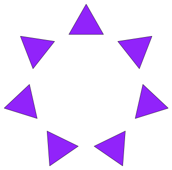


ring

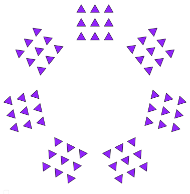
```
(ring n offset groups)
```

Better clock-rotate

```
(ring 7 0.5 (repeat shape))
```



```
(ring 7 0.5 (repeat (grid-layout 3 (repeat purple-triangle))))
```



(to research: what exactly does the offset parameter do?)

scale-group-stream

```
(scale-group-stream n groups)
```

sshape-as-layout

```
(sshape-as-layout sshape group-stream scalar)
```

Looks like it draws at positions defined by an sshape (but what is an sshape?)

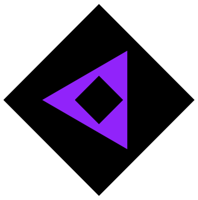
sshape-to-positions

```
(sshape-to-positions {:keys [style points], :as sshape})
```

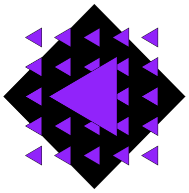
Used by sshape-as-layout

stack

```
(stack & groups)
superimpose a number of groups
(stack black-square (scale 0.6 purple-triangle) (scale 0.25 black-square))
```



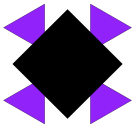
```
(stack black-square (scale 0.6 (grid-layout 5 (repeat purple-triangle))) (scale 0.5
purple-triangle))
```



superimpose-layout

`(superimpose-layout group1 group2)`

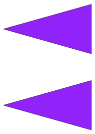
```
simplest layout, two groups located on top of each other

(superimpose-layout (four-mirror purple-triangle) black-square)
```



v-mirror

`(v-mirror group)`

```
Reflect vertically and stretch
```



patterning.groups

bottom

`(bottom group)`

clip

`(clip p? group)`

```
clips all sshapes in a group
```

clip-sshape

`(clip-sshape p? {:keys [style points]})` takes a predicate and a sshape, splits the sshape at any point which doesn't meet the predicate, return group

color-set

`(color-set group)`

empty-group

`(empty-group)`

extract-points

`(extract-points {:keys [style points]})`

filter-group

`(filter-group p? group)`

filter-sshapes-in-group

`(filter-sshapes-in-group p? group)`

this removes entire sshapes from the group that have points that don't match the criteria

flatten-group

`(flatten-group group)(flatten-group style group)`

Flatten all sshapes into a single sshape

group

`(group & sshapes)`

a vector of sshapes

h-centre

`(h-centre group)`

Assumes group is taller than wide so move it to horizontal centre

h-reflect

`(h-reflect group)`

height

`(height group)`

leftmost

`(leftmost group)`

mol=

```
(mol= group1 group2)
```

more or less equal groups

over-style

```
(over-style style group)
```

Changes the style of a group

reframe

```
(reframe group)
```

reframe-scaler

```
(reframe-scaler sshape)
```

Takes a sshape and returns a scaler to reduce it to usual viewport coords [-1 -1][1 1]

rightmost

```
(rightmost group)
```

rotate

```
(rotate da group)
```

scale

```
(scale val group)
```

stretch

```
(stretch sx sy group)
```

style-attribute-set

```
(style-attribute-set group attribute)
```

top

```
(top group)
```

translate

```
(translate dx dy group)
```

translate-to

```
(translate-to x y group)
```

v-reflect

```
(v-reflect group)
```

width

```
(width group)
```

wobble

```
(wobble noise group)
```

patterning.library.std

background

```
(background color pattern)
```

bez-curve

```
(bez-curve points style)(bez-curve points)
```

cross

```
(cross color x y)
```

A cross, can only be made as a group (because sshapes are continuous lines) which is why we only define it now

diamond

drunk-line

h-sin

horizontal-line

nangle

ogee

```
(ogee resolution stretch style)
```

An ogee shape

poly

quarter-ogee

rand-angle

`(rand-angle seed)`

random-rect

`(random-rect style)`

rect

spiral

spiral-points

`(spiral-points a da r dr)`

square

star

vertical-line

patterning.library.symbols

flower-of-life

`(flower-of-life sides style)(flower-of-life style)`

folexample

`(folexample)`

god-pattern

`(god-pattern)`

khatim

`(khatim style)`

ringed-flower-of-life

`(ringed-flower-of-life sides style)(ringed-flower-of-life style)`

seed-of-life

`(seed-of-life style)`

patterning.library.complex_elements

all

`(all count)`

f-left

`(f-left count)`

f-right

`(f-right count)`

face-group

`(face-group [head-sides head-color] [eye-sides eye-color] [nose-sides nose-color] [mouth-sides mouth-color])`

[head, eyes, nose and mouth] each argument is a pair to describe a poly [no-sides color]

petal-group

`(petal-group style dx dy)`

Using bezier curves

petal-pair-group

`(petal-pair-group style dx dy)`

reflected petals

polyflower-group

`(polyflower-group sides-per-poly no-polies radius style)(polyflower-group sides-per-poly no-polies radius)`

number of polygons rotated and superimosed

r-scroll

`(r-scroll d da number style extras)`

scroll

`(scroll [x y] d da number style extras)`

spoke-flake-group

`(spoke-flake-group style)`

The thing from my 'Bouncing' Processing sketch

vase

`(vase d da count style)`

zig-zag

`(zig-zag [x y])`

patterning.library.turtle

basic-turtle

`(basic-turtle start-pos d init-angle d-angle string leaf-map style)`

turns a string from the l-system into a number of lines

l-string-turtle-to-group-r

`(l-string-turtle-to-group-r [ox oy] d angle da string leaf-map style)`

A more sophisticated turtle that renders l-system string but has a stack and returns a group

patterning.library.l_systems

applicable

`(applicable [from to] c)`

apply-rule-to-char

`(apply-rule-to-char rule c)`

apply-rules

`(apply-rules rules string)`

apply-rules-to-char

`(apply-rules-to-char rules c)`

l-system

`(l-system rules)`

multi-apply-rules

```
(multi-apply-rules steps rules string)
```