algoraphics Documentation

Dan Munro

Oct 12, 2019

CONTENTS:

A library for creating graphics using algorithms and randomness.

CHAPTER

ONE

USER GUIDE

Note: The images in this guide can usually be generated by prefixing the corresponding code example with:

```
import algoraphics as ag
w, h = 400, 400
```

and appending this:

```
c = ag.Canvas(w, h)
c.add(x)
c.png("test.png")
```

1.1 Components

1.1.1 Canvas

The algoraphics library uses the Cartesian coordinate system. This means that while SVG and other computer graphics systems often define the top of the canvas as y=0 with y increasing downward, here y=0 is the bottom of the canvas with y increasing upward. It also means that while the coordinate units are pixels, negative and non-integer values are allowed.

The Canvas object holds a list of objects to be drawn. It has a `width`, `height`, and `background` color (or `None` for transparent background). It has methods to `add` one or more objects and `clear` the canvas (for convenience, `new` clears the canvas and then adds the objects). The SVG representation of a filled canvas can be retrieved with `get_svg`, saved to file with `svg`, or rendered to PNG with `png`. Likewise, animated graphics can be saved as a GIF with `gif`.

1.1.2 Shapes

Primitive visible objects are represented with a few Shape subclasses:

Object	Attributes
Circle	c (point), r (float)
Group	members (list), clip (list)
Line	points (list)
Polygon	points (list)
Spline	points (list), smoothing (float), circular (bool)

Shapes can be stored in nested lists without affecting their rendering. The purpose of groups, on the other hand, is to apply things like clipping and shadows to its members. Convenience functions like `rectangle` exist to define one of these shapes in other ways.

1.1.3 Styles

SVG attributes are used to style *shapes* and are stored as a dictionary in each *shape's* `style` attribute. Use the `set_style` function to set one style for one or more shapes, or `set_styles` to set separate copies of a style to each shape in the list, e.g. to sample from a Color object defined by random parameters.

1.1.4 Parameters

Many algoraphics functions accept abstract parameters that specify a distribution from which to randomly sample.

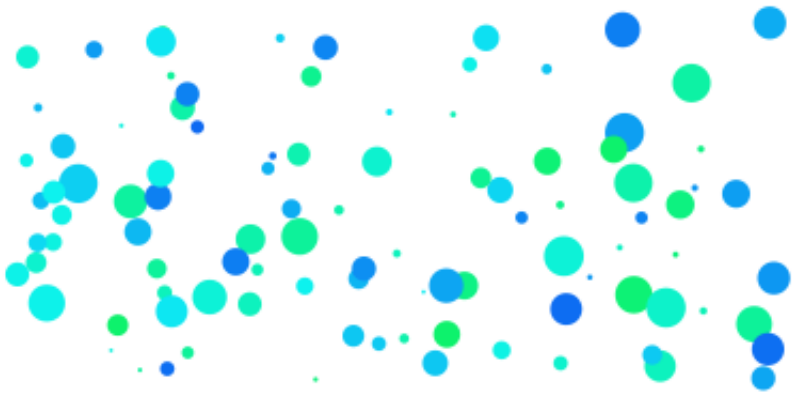
Algoraphics represents location and other properties declaratively. For example, the end of a tree tranch would be defined relative to the position of its start, and the position of its start would be defined with the same position object as the end of its parent branch.

This makes it simple to animate compound objects in a natural way. In the tree described above, when one branch sways, all of its child branches sway with it.

Parameters can also be defined with a distribution. This makes it easy to incorporate subtle or not-so-subtle randomness into the graphics. It also allows basic functions to be used in multiple ways to create different patterns.

For example, a simple command to draw 100 circles can produce this:

```
w, h = 400, 200
x = 100 * ag.Circle(
    c=(ag.Uniform(10, w - 10), ag.Uniform(10, h - 10)),
    r=ag.Uniform(1, 10),
    fill=ag.Color(hue=ag.Uniform(0.4, 0.6), sat=0.9, li=0.5),
)
```



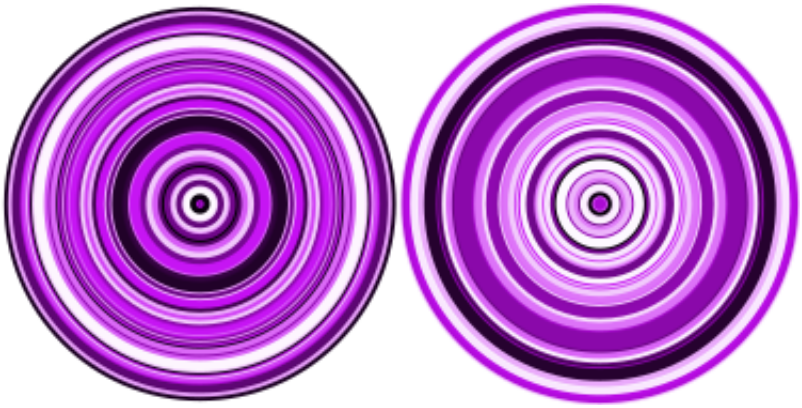
or this:

```
w, h = 400, 200
x = [
    ag.Circle(
        c=(ag.Param([100, 300]), 100),
        r=r,
        fill=ag.Color(hue=0.8, sat=0.9, li=ag.Uniform(0, 1)),
    )
```

(continues on next page)

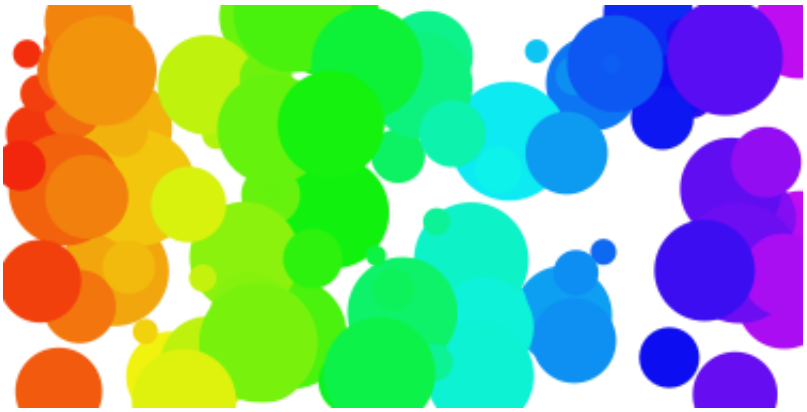
(continued from previous page)

```
    for r in range(100, 0, -1)
]
```



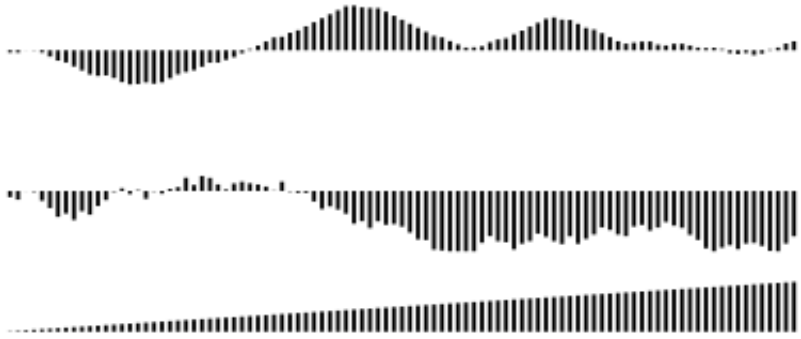
or this:

```
w, h = 400, 200
x = 100 * ag.Circle(c=(ag.Uniform(0, 400), ag.Uniform(0, h)), r=ag.Uniform(5, 30))
for circle in x:
    color = ag.Color(hue=circle.c.state(0)[0] / 500, sat=0.9, li=0.5)
    ag.set_style(circle, "fill", color)
```



Parameters can be defined relative to other parameters:

```
p2y = ag.Delta(start=170, delta=-0.25)
x.append([ag.line((i * 4, 170), (i * 4, p2y)) for i in range(100)])

p2y = ag.Delta(start=100, min=70, max=130, delta=ag.Uniform(-5, 5))
x.append([ag.line((i * 4, 100), (i * 4, p2y)) for i in range(100)])

p2y = ag.Delta(
    start=30,
    min=0,
    max=60,
    delta=ag.Delta(start=0, min=-2, max=2, delta=ag.Uniform(-2, 2)),
)
x.append([ag.line((i * 4, 30), (i * 4, p2y)) for i in range(100)])

ag.set_style(x, 'stroke-width', 2)
```



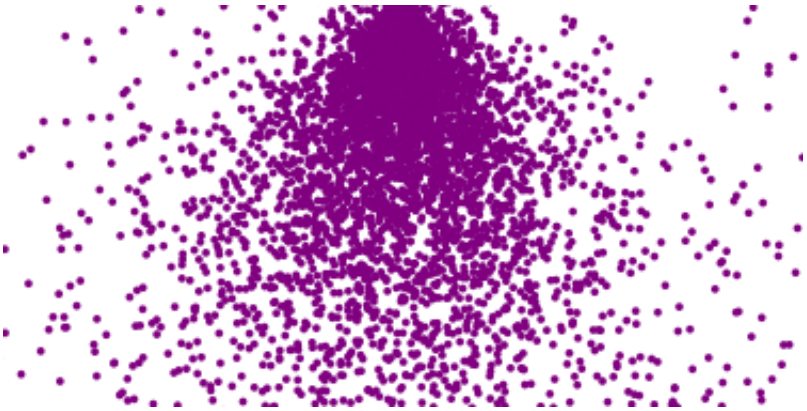
By adding random values to another parameter, you get random-walk-type behavior (middle row of lines above). This chaining doesn't have to apply directly to the shapes: you can use it to update a parameter representing the change in another parameter, resulting in second-order dynamics (bottom row of lines above).

A *parameter* can also be defined as a list, from which it will choose the value randomly, or with an arbitrary function, which will be called with no arguments to generate the value.

1.1.5 Location Parameters

While Param and its subclasses are used for one-dimensional *parameters*, two-dimensional, location-based parameters are handled with Point objects:

```
place = ag.Place(
    ref=(200, 0),
    direction=ag.Normal(90, 30),
    distance=ag.Exponential(mean=50, stdev=100, sigma=3),
)
x = [ag.circle(p, 2, fill="purple") for p in place.values(10000)]
```



As with single-dimension parameters, Points can be defined relative to others:

```
x = []
center = (0, 200)
direc = 0
deltadirec = 0
for i in range(300):
    x.append(ag.Circle(center, r=ag.Uniform(2, 4)))
    deltadirec = ag.Clip(deltadirec + ag.Uniform(-5, 5), -10, 10)
```

(continues on next page)

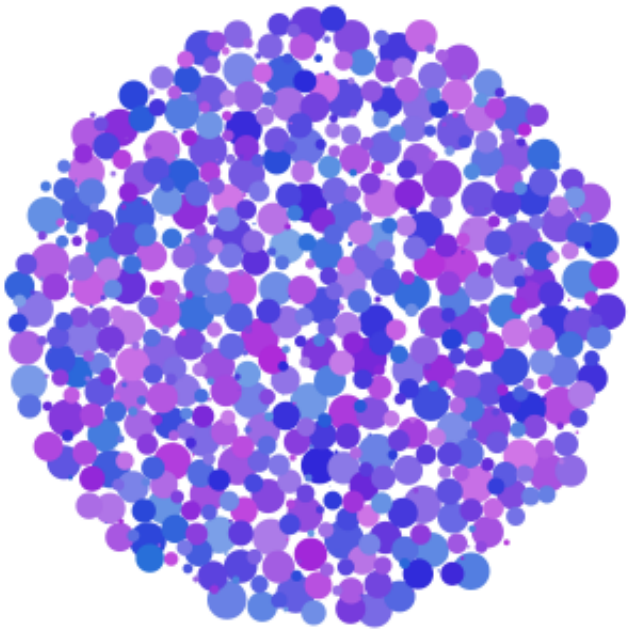
(continued from previous page)

```
    direc = direc + deltadirec
    center = ag.Move(center, direction=direc, distance=ag.Uniform(8, 12))
```

1.1.6 Colors

Colors are represented as objects of the Color class. They are generally defined in the HSL (hue, saturation, lightness) color space. These can be supplied as Param objects:

```
outline = ag.Circle(c=(200, 200), r=150)
color = ag.Color(
    hue=ag.Uniform(min=0.6, max=0.8), sat=0.7, li=ag.Uniform(min=0.5, max=0.7)
)
x = ex.fill_spots(outline)
ag.set_styles(x, "fill", color)
```



Shape color attributes like `fill` and `stroke` can be set with a string, which will be used as-is in the SVG file. This will work for hex codes, named colors, etc.

1.1.7 Animation

The Dynamic parameter type allows parameters to update at each frame of an animation. For each Dynamic parameter, you define its initial state with a parameter, and then the amount added or multiplied by the previous value to update over time. Every other parameter defined relative to that one is updated too. This allows structures to move in natural ways:

```
x = []
center = (0, 200)
direc = 0
deltadirec = 0
for i in range(100):
    deltar = ag.Dynamic(0, ag.Uniform(-0.2, 0.2, static=False), min=-0.2, max=0.2)
    r = ag.Dynamic(ag.Uniform(2, 4), delta=deltar, min=2, max=4)
    x.append(ag.Circle(center, r))
    # deltadirec = ag.Clip(deltadirec + ag.Uniform(-5, 5), -10, 10)
    deldeldir = ag.Dynamic(
        ag.Uniform(-5, 5), delta=ag.Uniform(-0.2, 0.2, static=False), min=-5, max=5
    )
    deltadirec = ag.Clip(deltadirec + deldeldir, -10, 10)
    direc = direc + deltadirec
    deltadist = ag.Dynamic(
        ag.Uniform(-2, 2), delta=ag.Uniform(-0.5, 0.5, static=False), min=-2, max=2
    )
    dist = ag.Dynamic(ag.Uniform(8, 12), delta=deltadist, min=8, max=12)
```

(continues on next page)

(continued from previous page)

```
    center = ag.Move(center, direction=direc, distance=dist)
c = ag.Canvas(400, 400)
c.add(x)
c.gif("png/param9.gif", fps=12, seconds=4)
```

1.1.8 SVG Representation

Shapes are converted to SVG for export. Each type of *shape* corresponds to a SVG object type or a specific form of one.

algoraphics	SVG
circle	circle
group	g
line	polyline
polygon	polygon
spline	path made of bezier curves

SVG-rendered effects like shadows applied to objects become references to SVG filters, which are defined at the beginning of the SVG file.

By default, the SVG code is optimized using `svgo`, but this can be skipped for more readable SVG code, e.g. for debugging.

1.2 Extras

You can create your own package of reusable functions built upon algoraphics. The `extras` subpackage is an example of this containing structures, fill functions, and more.

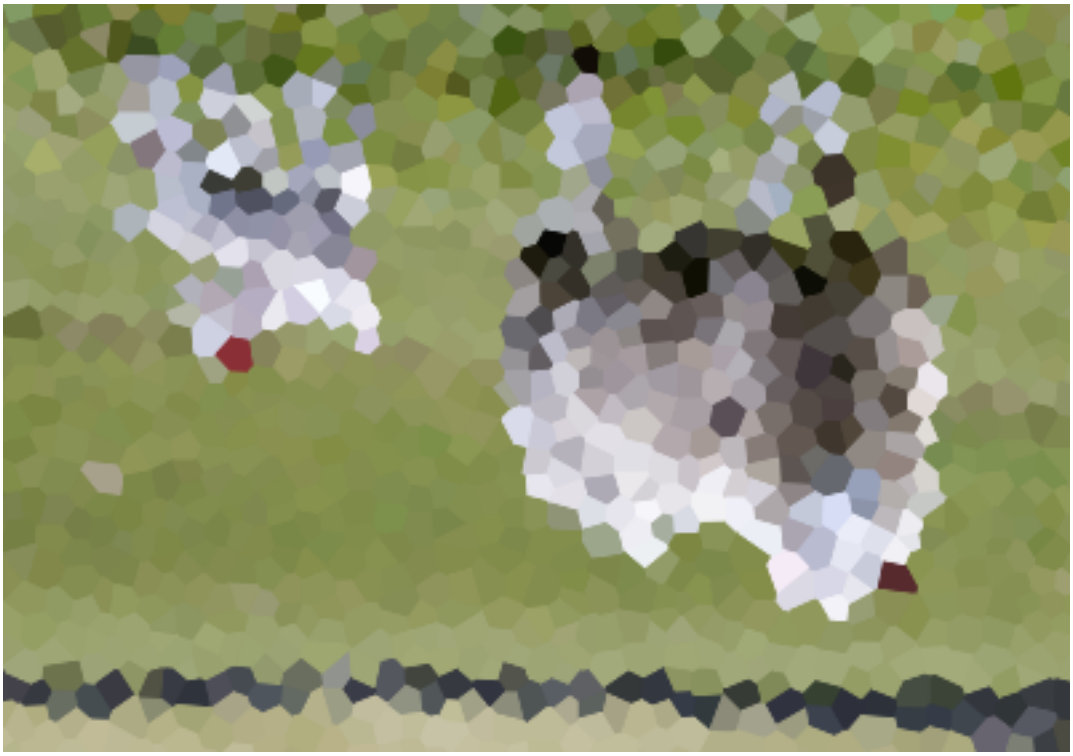
These code snippets are preceded by:

```
import algoraphics.extras as ex
```

1.2.1 Images

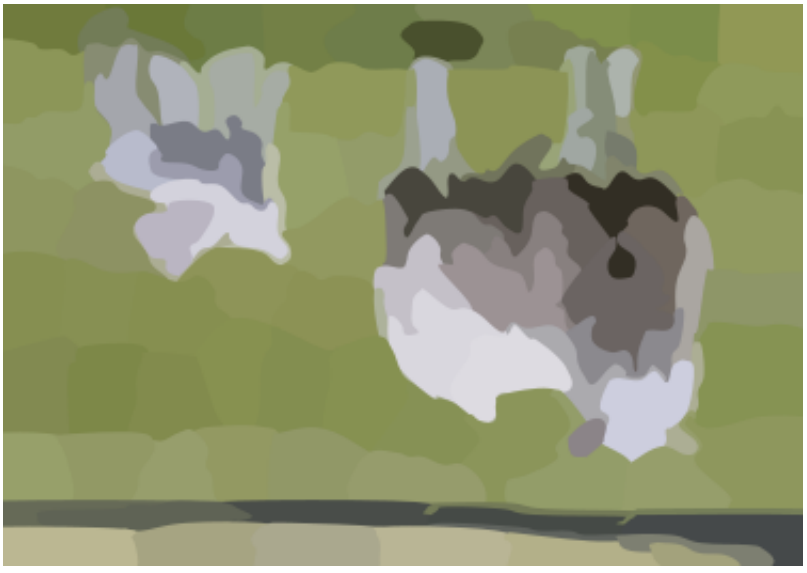
Images can be used as templates for use with patterns or textures. The simplest strategy is to sample colors from the *image* to color *shapes* at corresponding locations:

```
image = ag.open_image("test_images.jpg")
ag.resize_image(image, 800, None)
w, h = image.size
x = ag.tile_canvas(w, h, shape='polygon', tile_size=100)
ag.fill_shapes_from_image(x, image)
```



Images can also be segmented into *regions* that correspond to detected color boundaries with some smoothing, but are constrained to not be too large:

```
image = ag.open_image("test_images.jpg")
ag.resize_image(image, 800, None)
w, h = image.size
x = ag.image_regions(image, smoothness=3)
for outline in x:
    color = ag.region_color(outline, image)
    ag.set_style(outline, 'fill', color)
ag.add_paper_texture(x)
```



Fill functions can be applied and passed representative colors:

```
image = ag.open_image("test_images.jpg")
ag.resize_image(image, 800, None)
w, h = image.size
x = ag.image_regions(image, smoothness=3)
for i, outline in enumerate(x):
    color = ag.region_color(outline, image)
    maze = ag.Maze_Style_Pipes(rel_thickness=0.6)
    rot = color.value()[0] * 90
    x[i] = ag.fill_maze(outline, spacing=5, style=maze, rotation=rot)
    ag.set_style(x[i]["members"], "fill", color)
    ag.region_background(x[i], ag.contrasting_lightness(color, light_diff=0.2))
```

1.2.2 Text

Text can be created and stylized. Characters are generated as nested lists of *points* (one list per continuous pen stroke) along their form:

```
color = ag.Color(hue=ag.Uniform(0, 0.15), sat=0.8, li=0.5)

points = ex.text_points("ABCDEFG", 50, pt_spacing=0.5, char_spacing=0.15)
ag.jitter_points(points, 2)
size = ag.Exponential(2.2, stdev=1).values(len(points))
x1 = [ag.circle(c=p, r=size[i], fill=color) for i, p in enumerate(points)]
ag.reposition(x1, (w / 2, h - 50), "center", "top")
c.new(ag.shuffled(x1))

points = ex.text_points("HIJKLM", 50, pt_spacing=0.5, char_spacing=0.15)
ag.jitter_points(points, 2)
size = ag.Exponential(2.2, stdev=1).values(len(points))
x2 = [ag.circle(c=p, r=size[i], fill=color) for i, p in enumerate(points)]
ag.reposition(x2, (w / 2, h - 150), "center", "top")
c.add(ag.shuffled(x2))

points = ex.text_points("0123456789", 50, pt_spacing=0.5, char_spacing=0.15)
ag.jitter_points(points, 2)
size = ag.Exponential(2.2, stdev=1).values(len(points))
```

(continues on next page)

(continued from previous page)

```
x3 = [ag.circle(c=p, r=size[i], fill=color) for i, p in enumerate(points)]
ag.reposition(x3, (w / 2, h - 250), "center", "top")
c.add(ag.shuffled(x3))

x = []
```

These *points* can then be manipulated in many ways:

```
points = ex.text_points("NOPQRST", 40, pt_spacing=0.3, char_spacing=0.15)
ag.jitter_points(points, 8)
size = ag.Exponential(2.2, stdev=1).values(len(points))
x1a = [ag.circle(c=p, r=size[i], fill="black") for i, p in enumerate(points)]

points = ex.text_points("NOPQRST", 40, pt_spacing=1, char_spacing=0.15)
ag.jitter_points(points, 2)
size = ag.Exponential(1.5, stdev=0.5).values(len(points))
x1b = [ag.circle(c=p, r=size[i], fill="white") for i, p in enumerate(points)]

ag.reposition([x1a, x1b], (w / 2, h - 50), "center", "top")
c.new(x1a, x1b)

points = ex.text_points("UVWXYZ", 40, pt_spacing=0.3, char_spacing=0.15)
ag.jitter_points(points, 8)
size = ag.Exponential(2.2, stdev=1).values(len(points))
x2a = [ag.circle(c=p, r=size[i], fill="black") for i, p in enumerate(points)]

points = ex.text_points("UVWXYZ", 40, pt_spacing=1, char_spacing=0.15)
ag.jitter_points(points, 2)
size = ag.Exponential(1.5, stdev=0.5).values(len(points))
```

(continues on next page)

(continued from previous page)

```
x2b = [ag.circle(c=p, r=size[i], fill="white") for i, p in enumerate(points)]

ag.reposition([x2a, x2b], (w / 2, h - 150), "center", "top")
c.add(x2a, x2b)

points = ex.text_points(".,!?:;'\"/", 40, pt_spacing=0.3, char_spacing=0.15)
ag.jitter_points(points, 8)
size = ag.Exponential(2.2, stdev=1).values(len(points))
x3a = [ag.circle(c=p, r=size[i], fill="black") for i, p in enumerate(points)]

points = ex.text_points(".,!?:;'\"/", 40, pt_spacing=1, char_spacing=0.15)
ag.jitter_points(points, 2)
size = ag.Exponential(1.5, stdev=0.5).values(len(points))
x3b = [ag.circle(c=p, r=size[i], fill="white") for i, p in enumerate(points)]

ag.reposition([x3a, x3b], (w / 2, h - 250), "center", "top")
c.add(x3a, x3b)
```

Currently only the characters displayed in these examples are provided, though additional ones can be added on request:

```
pts = ex.text_points("abcdefg", height=50, pt_spacing=1/4, char_spacing=0.15)
dists = ag.Uniform(0, 10).values(len(pts))
points = [ag.endpoint(p, ag.Uniform(0, 360).value(), dists[i]) for i, p in enumerate␣
↪(pts)]
radii = 0.5 * np.sqrt(10 - np.array(dists))
x1 = [ag.circle(c=p, r=radii[i]) for i, p in enumerate(points)]
ag.reposition(x1, (w / 2, h - 100), "center", "top")
```

(continues on next page)

(continued from previous page)

```
ag.set_style(x1, "fill", "green")

pts = ex.text_points("hijklm", height=50, pt_spacing=1/4, char_spacing=0.15)
dists = ag.Uniform(0, 10).values(len(pts))
points = [ag.endpoint(p, ag.Uniform(0, 360).value(), dists[i]) for i, p in␣
↪enumerate(pts)]
radii = 0.5 * np.sqrt(10 - np.array(dists))
x2 = [ag.circle(c=p, r=radii[i]) for i, p in enumerate(points)]
ag.reposition(x2, (w / 2, h - 250), "center", "top")
ag.set_style(x2, "fill", "green")

c.new(x1, x2)
```

Since generated *points* are grouped by continuous pen strokes, *points* within each list can be joined:

```
strokes = ex.text_points("nopqrst", 60, pt_spacing=1,
                         char_spacing=0.2, grouping='strokes')
for stroke in strokes:
    ag.jitter_points(stroke, 10)
x1 = [ag.spline(points=stroke) for stroke in strokes]
ag.reposition(x1, (w / 2, h - 100), "center", "top")

strokes = ex.text_points("uvwxyz", 60, pt_spacing=1,
                         char_spacing=0.2, grouping='strokes')
for stroke in strokes:
    ag.jitter_points(stroke, 10)
x2 = [ag.spline(points=stroke) for stroke in strokes]
ag.reposition(x2, (w / 2, h - 250), "center", "top")
```

(continues on next page)

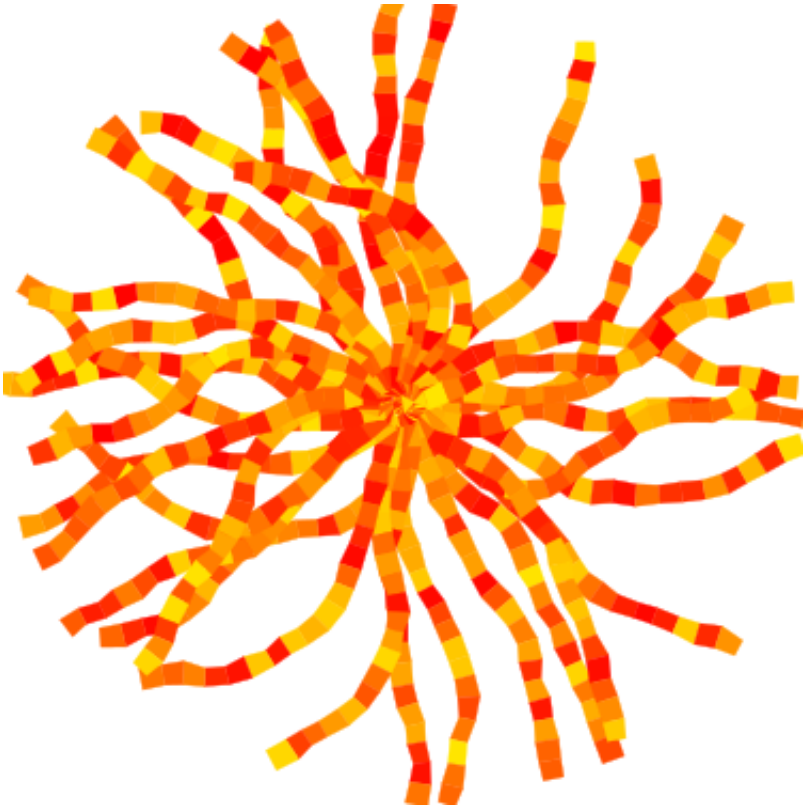
(continued from previous page)

```
c.new(x1, x2)
```

1.2.3 Filaments

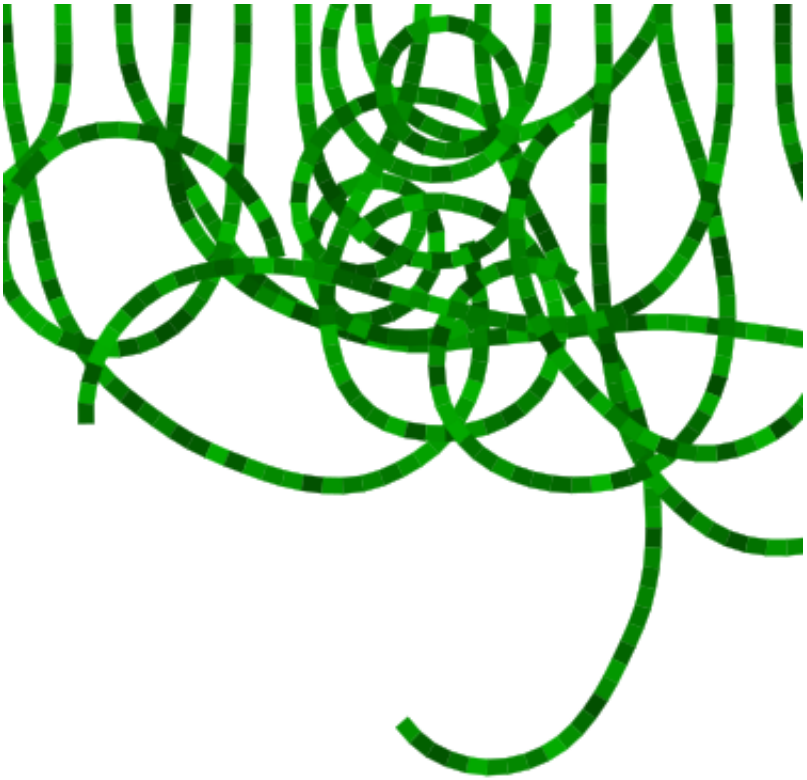
Filaments made of quadrilateral segments can be generated:

```
dirs = [ag.Param(d, delta=ag.Uniform(min=-20, max=20))
        for d in range(360)[::10]]
width = ag.Uniform(min=8, max=12)
length = ag.Uniform(min=8, max=12)
x = [ag.filament(start=(w / 2., h / 2.), direction=d, width=width,
                 seg_length=length, n_segments=20) for d in dirs]
ag.set_style(x, 'fill', ag.Color(hsl=(ag.Uniform(min=0, max=0.15), 1, 0.5)))
```



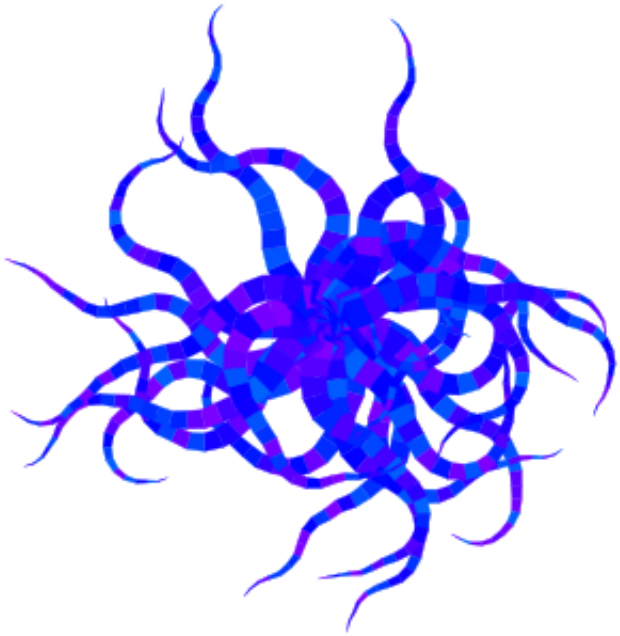
The direction *parameter's* delta or ratio attribute allows the filament to move in different directions. Nested deltas produce smooth curves:

```
direc = ag.Param(90, delta=ag.Param(0, min=-20, max=20,
                                    delta=ag.Uniform(min=-3, max=3)))
x = [ag.filament(start=(z, -10), direction=direc, width=8,
                 seg_length=10, n_segments=50) for z in range(w)[::30]]
ag.set_style(x, 'fill',
             ag.Color(hsl=(0.33, 1, ag.Uniform(min=0.15, max=0.35))))
```



A tentacle is a convenience wrapper for a filament with steadily decreasing segment width and length to come to a point at a specified total length:

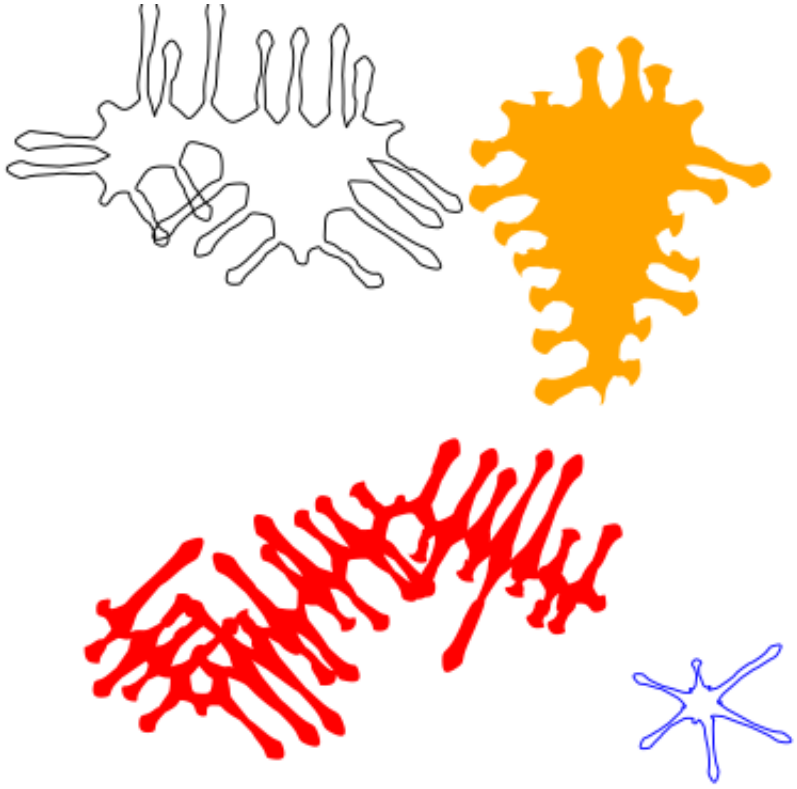
```
dirs = [ag.Param(d, delta=ag.Param(0, min=-20, max=20,
                                   delta=ag.Uniform(min=-30, max=30)))
        for d in range(360)[::10]]
x = [ag.tentacle(start=(w/2, h/2), length=225, direction=d, width=15,
                 seg_length=10) for d in dirs]

ag.set_style(x, 'fill', ag.Color(hsl=(ag.Uniform(min=0.6, max=0.75), 1, 0.5)))
```



1.2.4 Blow paint

Blow painting effects (i.e., droplets of paint blown outward from an object) can be created for 0D, 1D, and 2D forms:

```
pts1 = [(50, 50), (50, 100), (100, 70), (150, 130), (200, 60)]
x1 = ag.blow_paint_area(pts1)

pts2 = [(250, 50), (350, 50), (300, 200)]
x2 = ag.blow_paint_area(pts2, spacing=20, length=20, len_dev=0.4, width=8)
ag.set_style(x2, 'fill', 'orange')

pts3 = [(50, 300), (100, 350), (200, 250), (300, 300)]
y = ag.blow_paint_line(pts3, line_width=8, spacing=15, length=30,
                       len_dev=0.4, width=6)
ag.set_style(y, 'fill', 'red')

z = ag.blow_paint_spot((350, 350), length=20)
ag.set_style(z, 'stroke', 'blue')
```



1.2.5 Trees

Trees with randomly bifurcating branches can be generated:

```
x = [ag.tree((200, 200), direction=d,
             branch_length=ag.Uniform(min=8, max=20),
             theta=ag.Uniform(min=15, max=20),
             p=ag.Param(1, delta=-0.08))
     for d in range(360)[::20]]
ag.set_style(x, 'stroke', ag.Color(hue=ag.Normal(0.12, stdev=0.05),
                                   sat=ag.Uniform(0.4, 0.7),
                                   li=0.3))
```

1.3 Fills

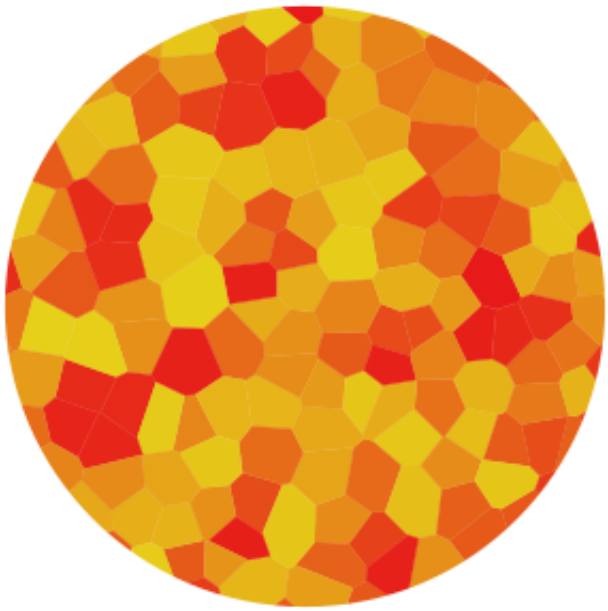
These functions fill a *region* with structures and patterns.

1.3.1 Tiling

These functions divide a *region's* area into tiles.

Random polygonal (i.e. Voronoi) tiles can be generated:

```
outline = ag.circle(c=(200, 200), r=150)
colors = ag.Color(hue=ag.Uniform(min=0, max=0.15), sat=0.8, li=0.5)
x = ag.tile_region(outline, shape='polygon', tile_size=500)
ag.set_style(x['members'], 'fill', colors)
```

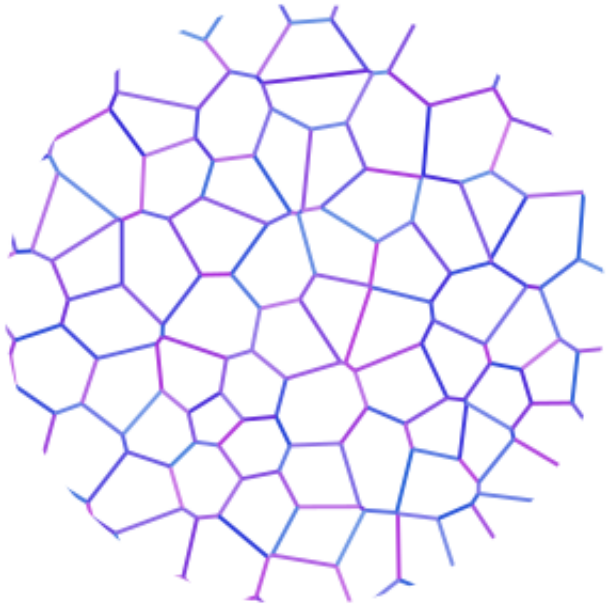


Random triangular (i.e. Delaunay) tiles can be generated:

```
outline = ag.circle(c=(200, 200), r=150)
colors = ag.Color(hue=ag.Uniform(min=0, max=0.15), sat=0.8, li=0.5)
x = ag.tile_region(outline, shape='triangle', tile_size=500)
ag.set_style(x['members'], 'fill', colors)
```

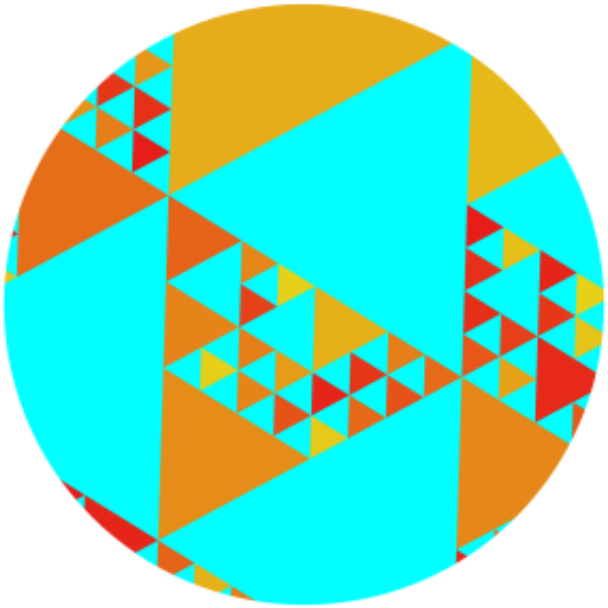
The edges between polygonal or triangular tiles can be created instead:

```
outline = ag.circle(c=(200, 200), r=150)
colors = ag.Color(hue=ag.Uniform(min=0.6, max=0.8), sat=0.7,
                  li=ag.Uniform(min=0.5, max=0.7))
x = ag.tile_region(outline, shape='polygon', edges=True, tile_size=1000)
ag.set_style(x['members'], 'stroke', colors)
ag.set_style(x['members'], 'stroke-width', 2)
```



Nested equilateral triangles can be created, with the level of nesting random but specifiable:

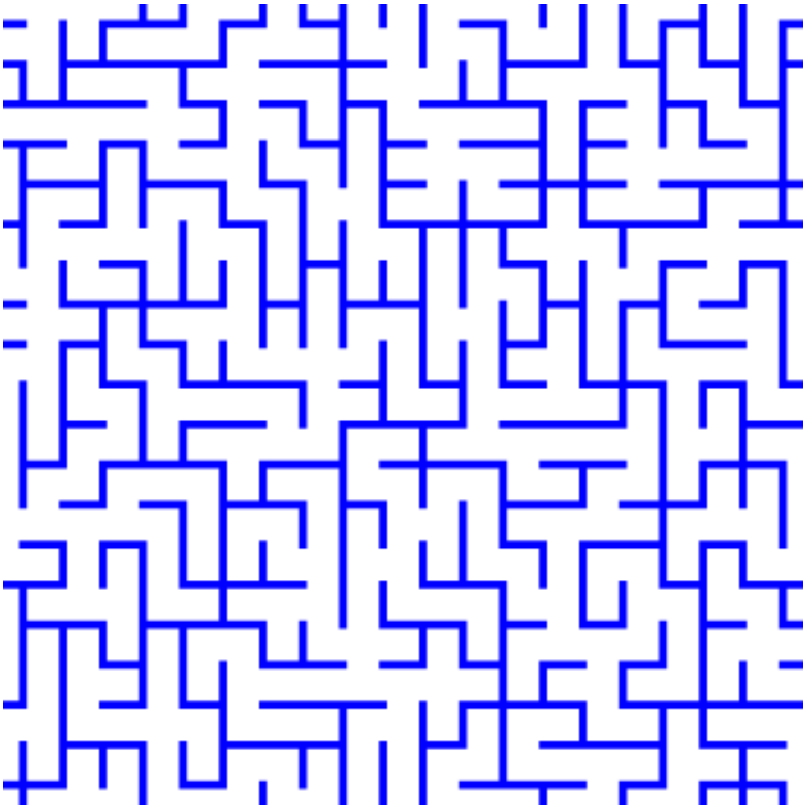
```
outline = ag.circle(c=(200, 200), r=150)
color = ag.Color(hue=ag.Uniform(min=0, max=0.15), sat=0.8, li=0.5)
x = ag.fill_nested_triangles(outline, min_level=2, max_level=5, color=color)
```



1.3.2 Mazes

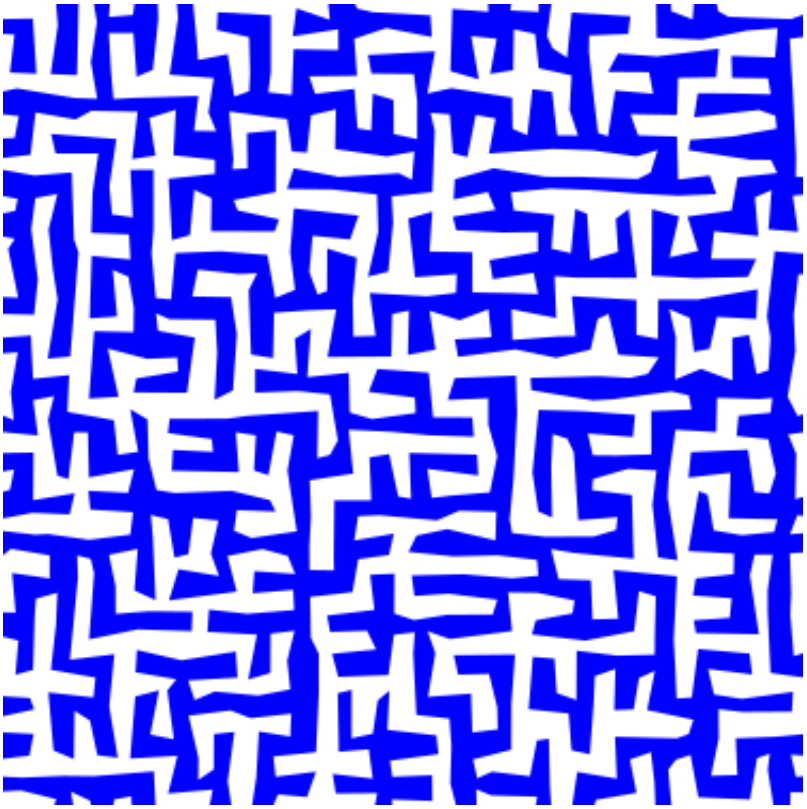
These patterns resemble mazes, but are actually random spanning trees:

```
outline = ag.rectangle(bounds=(0, 0, w, h))
x = ag.fill_maze(outline, spacing=20,
                 style=ag.Maze_Style_Straight(rel_thickness=0.2))
ag.set_style(x['members'], 'fill', 'blue')
```



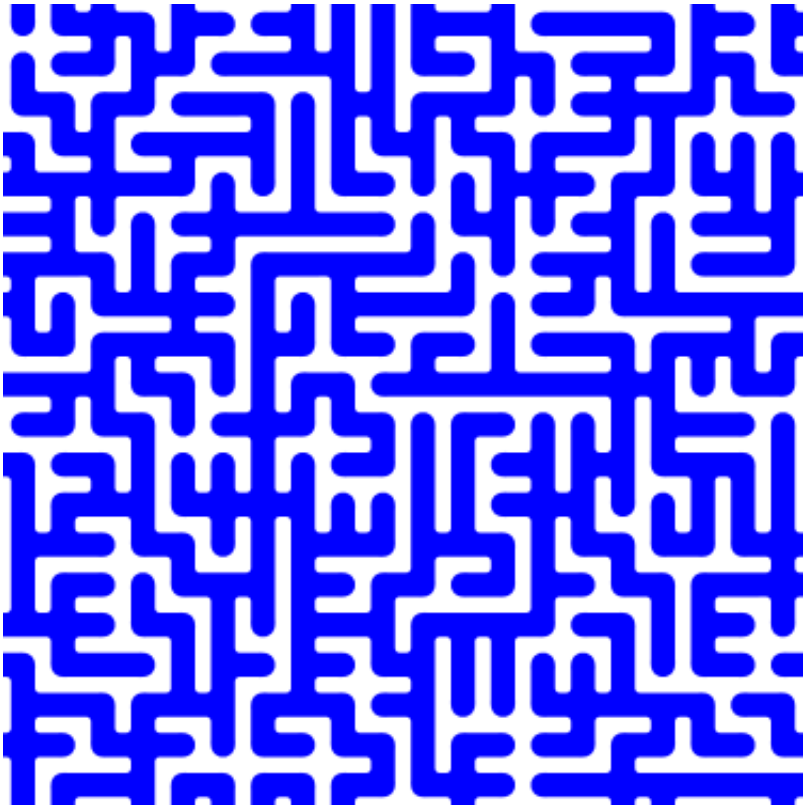
The maze style is defined by an instance of a subclass of `Maze_Style`:

```
outline = ag.rectangle(bounds=(0, 0, w, h))
x = ag.fill_maze(outline, spacing=20,
                 style=ag.Maze_Style_Jagged(min_w=0.2, max_w=0.8))
ag.set_style(x['members'], 'fill', 'blue')
```



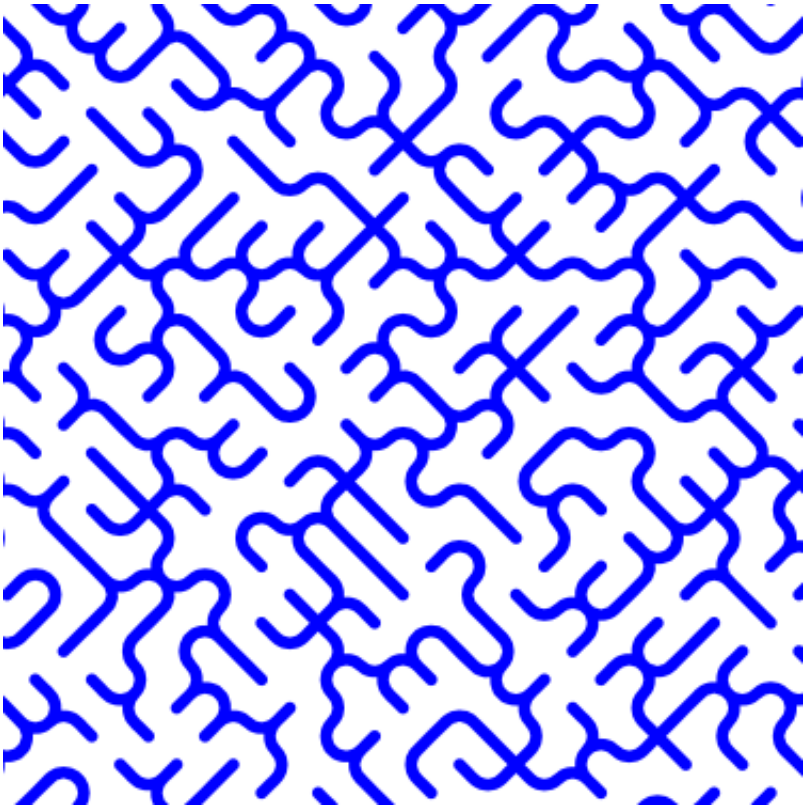
Each style defines the appearance of five maze components that each occupy one grid cell: tip, turn, straight, T, and cross. Each grid cell contains a rotation and/or reflection of one of these components:

```
outline = ag.rectangle(bounds=(0, 0, w, h))
x = ag.fill_maze(outline, spacing=20,
                 style=ag.Maze_Style_Pipes(rel_thickness=0.6))
ag.set_style(x['members'], 'fill', 'blue')
```



The grid can be rotated:

```
outline = ag.rectangle(bounds=(0, 0, w, h))
x = ag.fill_maze(outline, spacing=20,
                 style=ag.Maze_Style_Round(rel_thickness=0.3),
                 rotation=45)
ag.set_style(x['members'], 'fill', 'blue')
```



Custom styles can be used by creating a new subclass of *Maze_Style*.

1.3.3 Doodles

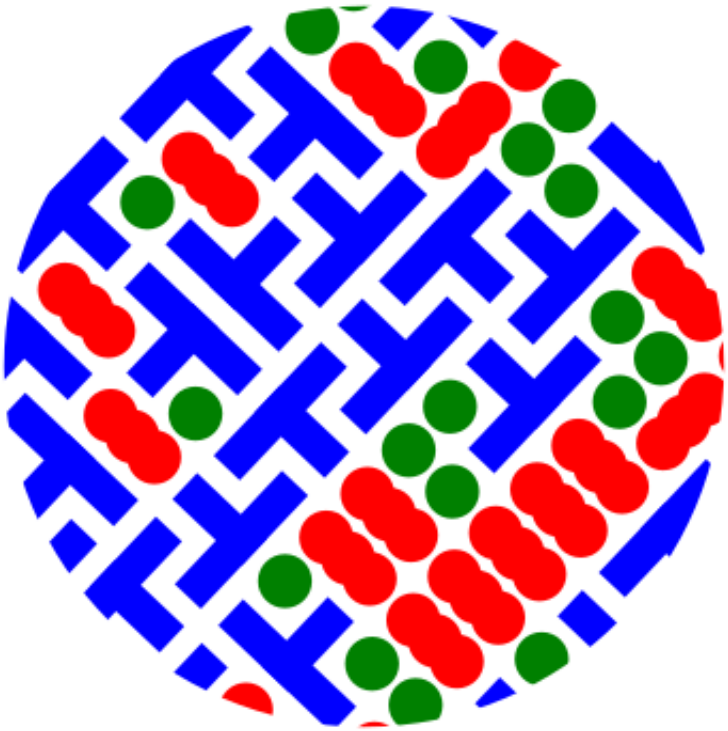
Small arbitrary objects, a.k.a. *doodles*, can be tiled to fill a *region*, creating a wrapping-paper-type pattern. The 'footprint', or shape of grid cells occupied, for each *doodle* is used to place different *doodles* in random orientations to fill a grid:

```
def doodle1_fun():
    d = ag.circle(c=(0.5, 0.5), r=0.45)
    ag.set_style(d, 'fill', 'green')
    return d

def doodle2_fun():
    d = [ag.circle(c=(0.5, 0.5), r=0.45),
         ag.circle(c=(1, 0.5), r=0.45),
         ag.circle(c=(1.5, 0.5), r=0.45)]
    ag.set_style(d, 'fill', 'red')
    return d

def doodle3_fun():
    d = [ag.rectangle(start=(0.2, 1.2), w=2.6, h=0.6),
         ag.rectangle(start=(1.2, 0.2), w=0.6, h=1.6)]
    ag.set_style(d, 'fill', 'blue')
    return d

doodle1 = ag.Doodle(doodle1_fun, footprint=[[True]])
doodle2 = ag.Doodle(doodle2_fun, footprint=[[True, True]])
```

(continues on next page)

(continued from previous page)

```
doodle3 = ag.Doodle(doodle3_fun, footprint=[[True, True, True],
                                            [False, True, False]])
doodles = [doodle1, doodle2, doodle3]
outline = ag.circle(c=(200, 200), r=180)
x = ag.fill_wrapping_paper(outline, 30, doodles, rotate=True)
```

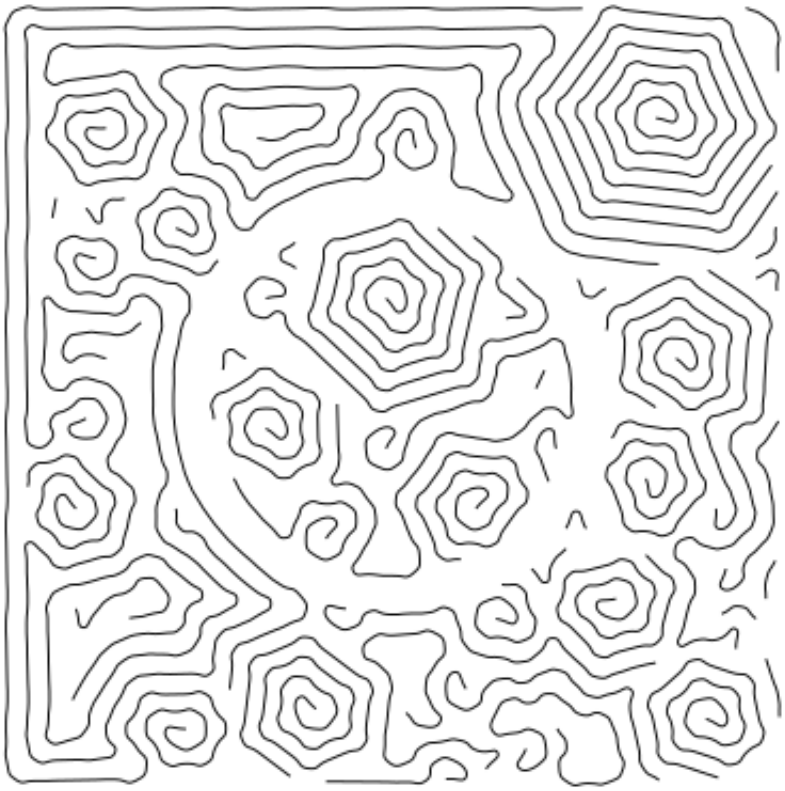


Each *doodle* is defined by creating a Doodle object that specifies a generating function and footprint. This allows each *doodle* to vary in appearance as long as it roughly conforms to the footprint.

1.3.4 Other fills

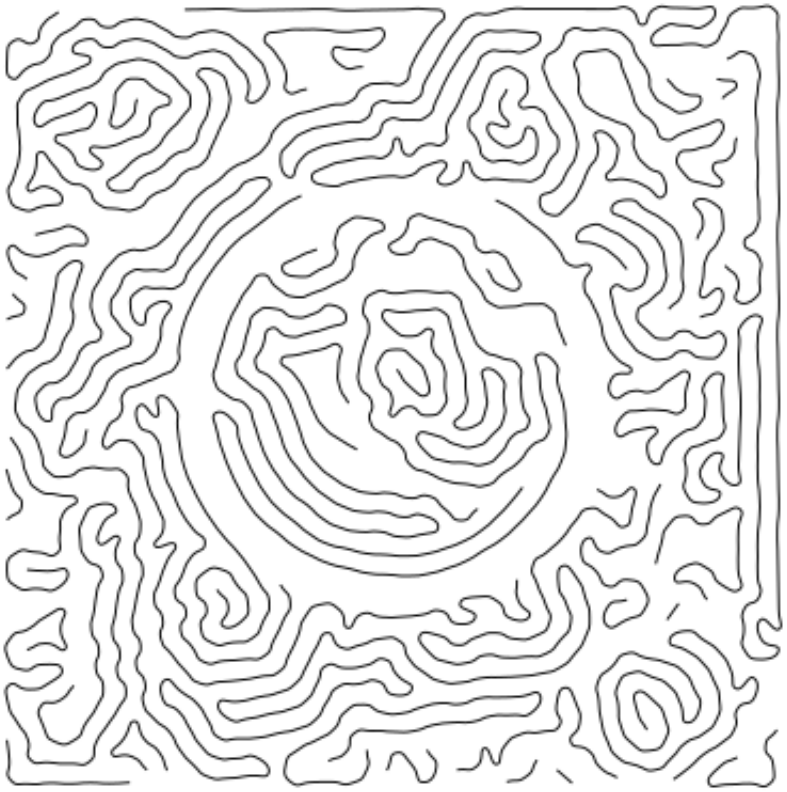
Ripples can fill the canvas while avoiding specified *points*:

```
circ = ag.points_on_arc(center=(200, 200), radius=100, theta_start=0,
                        theta_end=360, spacing=10)
x = ag.ripple_canvas(w, h, spacing=10, existing_pts=circ)
```



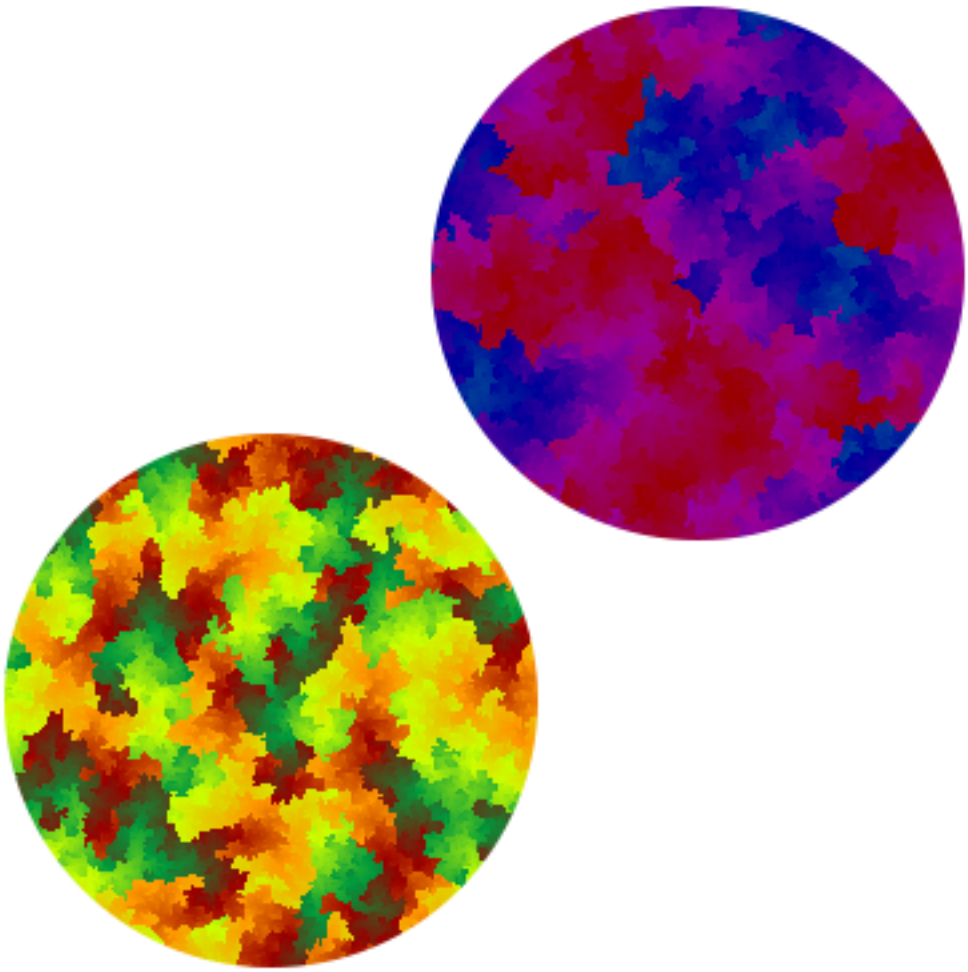
They are generated by a Markov chain telling them when to follow a boundary on the left, on the right, or to change direction. The transition probabilities for the Markov chain can be specified to alter the appearance:

```
trans_probs = dict(S=dict(X=1),
                   R=dict(R=0.9, L=0.05, X=0.05),
                   L=dict(L=0.9, R=0.05, X=0.05),
                   X=dict(R=0.5, L=0.5))
circ = ag.points_on_arc(center=(200, 200), radius=100, theta_start=0,
                        theta_end=360, spacing=10)
x = ag.ripple_canvas(w, h, spacing=10, trans_probs=trans_probs,
                     existing_pts=circ)
```



A billowing texture is produced by generating a random spanning tree across a grid of pixels, and then moving through the tree and coloring them with a cyclical color gradient:

```
outline = ag.circle(c=(120, 120), r=100)
colors = [(0, 1, 0.3), (0.1, 1, 0.5), (0.2, 1, 0.5), (0.4, 1, 0.3)]
x = ag.billow_region(outline, colors, scale=200, gradient_mode='rgb')

outline = ag.circle(c=(280, 280), r=100)
colors = [(0, 1, 0.3), (0.6, 1, 0.3)]
y = ag.billow_region(outline, colors, scale=400, gradient_mode='hsv')
```



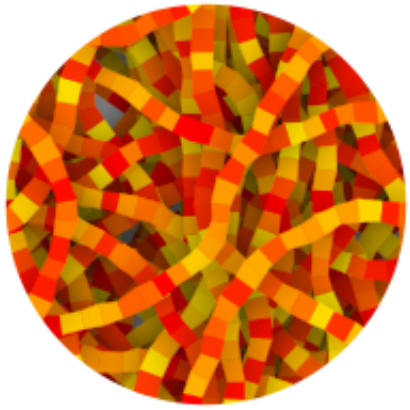
A *region* can be filled with structures such as filaments using a generic function that generates random instances of the structure and places them until the *region* is filled:

```
def filament_fill(bounds):
    c = ((bounds[0] + bounds[2]) / 2, (bounds[1] + bounds[3]) / 2)
    r = ag.distance(c, (bounds[2], bounds[3]))
    start = ag.rand_point_on_circle(c, r)
    dir_start = ag.direction_to(start, c)
    filament = ag.filament(
        start=start,
        direction=ag.Delta(dir_start, delta=ag.Uniform(min=-20, max=20)),
        width=ag.Uniform(min=8, max=12),
        seg_length=ag.Uniform(min=8, max=12),
        n_segments=int(2.2 * r / 10),
    )
    color = ag.Color(hsl=(ag.Uniform(min=0, max=0.15), 1, 0.5))
    ag.set_style(filament, "fill", color)
    return filament


outline = ag.circle(c=(200, 200), r=100)
```

(continues on next page)

(continued from previous page)

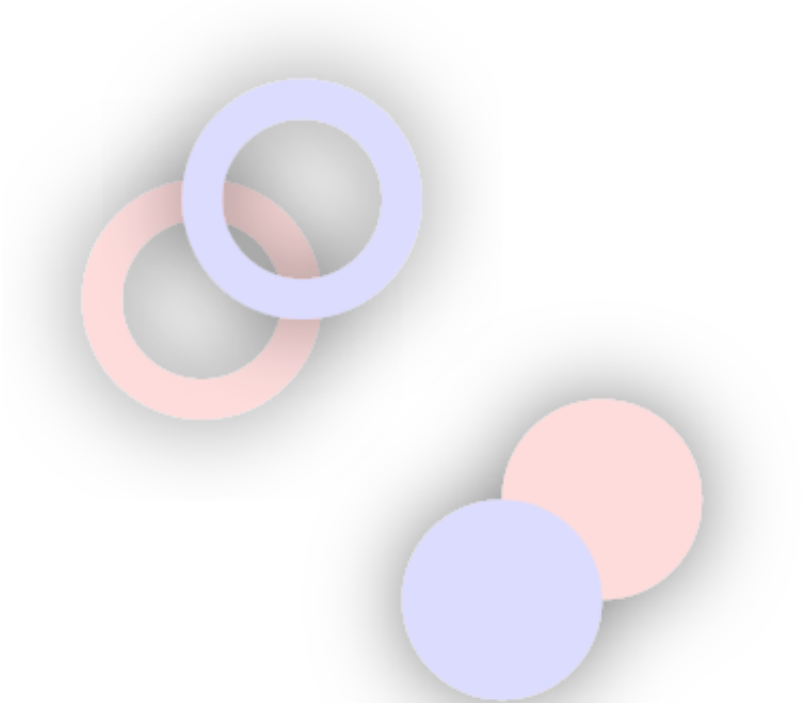
```
x = ag.fill_region(outline, filament_fill)
ag.add_shadows(x["members"])
```



1.4 Effects

Shadows can be added to *shapes* or *collections*, and shapes can be given rough paper textures:

```
x = [
    ag.circle(c=(100, 150), r=50, stroke="#FFDDDD"),
    ag.circle(c=(150, 100), r=50, stroke="#DDDDFF"),
]
ag.set_style(x, "stroke-width", 10)
ag.add_shadows(x, stdev=20, darkness=0.5)

y = [[
    ag.circle(c=(300, 250), r=50, fill="#FFDDDD"),
    ag.circle(c=(250, 300), r=50, fill="#DDDDFF"),
]]
ag.add_paper_texture(y)

ag.add_shadows(y, stdev=20, darkness=0.5)
```



CHAPTER

TWO

API

2.1 color.py

Functions for working with colors.

class algoraphics.color.**Color**(*hue=None*, *sat=None*, *li=None*, *RGB=None*)
Object to represent a color or distribution of colors.

Define using either an hsl tuple, separate hue/saturation/lightness arguments, or an RGB tuple. Any component can be a Param object.

Parameters

- **hsl** – The hsl specification, where each component is between 0 and 1.
- **hue** (Optional[float]) – The hue specification from 0 to 1. sat and li must also be provided.
- **sat** (Optional[float]) – The saturation specification from 0 to 1.
- **li** (Optional[float]) – The lightness specification from 0 to 1.
- **RGB** (Optional[Tuple[int, int, int]]) – The red/green/blue components, each ranging from 0 to 255.

hsl(*t=0*)
Get the color's hsl specification.

Returns the specification at time t if the color is parameterized.

Return type Tuple[float, float, float]

rgb(*t=0*)
Get the color's rgb specification.

Returns the specification at time t if the color is parameterized.

Return type Tuple[float, float, float]

state(*t=0*)
Get the color's hex specification.

Returns one fixed specification if the color is parameterized.

Return type str

algoraphics.color.**average_color**(*colors*)
Find average of list of colors.

This finds the arithmetic mean in RGB color space, since averaging hues has unexpected results with black, white, and gray.

Parameters **colors** (Sequence[*Color*]) – A list of Color objects.

Return type *Color*

Returns The average color.

algoraphics.color.**make_color**(*x*)
Convert to a color object if a tuple (assumed to be hsl) is provided.

Return type *Color*

2.2 geom.py

General functions involving points in 2D space.

algoraphics.geom.**angle_between**(*p1*, *p2*, *p3*)
Get the angle (in radians) between segment p2->p1 and p2->p3.

The angle can be negative.

Parameters

- **p1** (Tuple[float, float]) – The first endpoint.
- **p2** (Tuple[float, float]) – The point where the angle is calculated.
- **p3** (Tuple[float, float]) – The other endpoint.

Return type float

algoraphics.geom.**deg**(*rad*)
Convert radians to degrees.

Return type float

algoraphics.geom.**direction_to**(*p1*, *p2*)
Get the direction of p2 from p1 in degrees.

Return type float

algoraphics.geom.**distance**(*p1*, *p2*)
Get the distance between two points.

Return type float

algoraphics.geom.**endpoint**(*start*, *angle*, *distance*)

Parameters

- **start** (Tuple[float, float]) – Starting point.
- **angle** (float) – Direction from starting point in radians.
- **distance** (float) – Distance from starting point.

Return type Tuple[float, float]

Returns A point distance from start in the direction angle.

algoraphics.geom.**get_nearest**(*points*, *point*, *index=False*)
Find the nearest point in a list to a target point.

Parameters

- **points** (Sequence[Tuple[float, float]]) – A list of points.

- **point** (Tuple[float, float]) – The target point.
- **index** (bool) – Whether to return the point or its index in the list.

Return type Union[Tuple[float, float], int]

Returns If index is False, returns point, otherwise returns index of point in list.

algoraphics.geom.**horizontal_range**(*points*)
Get the magnitude of the horizontal range of a list of points.

Parameters **points** (Sequence[Tuple[float, float]]) – A list of points.

Return type float

algoraphics.geom.**interpolate**(*points*, *spacing*)
Insert interpolated points.

Insert equally-spaced, linearly interpolated points into list such that consecutive points are no more than 'spacing' distance apart.

Parameters

- **points** (Sequence[Tuple[float, float]]) – A list of points.
- **spacing** (float) – Maximum distance between adjacent points.

algoraphics.geom.**is_clockwise**(*points*)
Determine the derection of a sequence of points around a polygon.

Finds whether a set of polygon points goes in a clockwise or counterclockwise direction. If edges cross, it gives the more prominent direction.

Parameters **points** (Sequence[Tuple[float, float]]) – A list of polygon vertices.

Return type bool

algoraphics.geom.**jitter_points**(*points*, *r*)
Add noise to the locations of points.

Distance and direction of movement are both uniformly random, so the 2D probability density is circular with higher concentration toward the center.

Parameters

- **points** (Sequence[Tuple[float, float]]) – A list of points.
- **r** (float) – The maximum distance points will move.

algoraphics.geom.**jittered_points**(*points*, *r*)
Get noisy copy of points.

Like jitter_points but returns jittered points, not affecting the original list.

Parameters

- **points** (Sequence[Tuple[float, float]]) – A list of points.
- **r** (float) – The maximum distance points will move.

Return type Sequence[Tuple[float, float]]

Returns A list of points.

algoraphics.geom.**line_to_polygon**(*points*, *width*)
Convert a sequence of points to a thin outline.

Imagining the points were connected with a stroke with positive width, the outline of the stroke is returned.

Parameters

- **points** (Sequence[Tuple[float, float]]) – A list of line points.
- **width** (float) – Width of the stroke to be outlined.

Return type Sequence[Tuple[float, float]]

Returns A list of points.

algoraphics.geom.**midpoint**(*p1*, *p2*)

Get the midpoint between two points.

Parameters

- **p1** (Tuple[float, float]) – A point.
- **p2** (Tuple[float, float]) – Another point.

Return type Tuple[float, float]

algoraphics.geom.**move_toward**(*start*, *target*, *distance*)

Parameters

- **start** (Tuple[float, float]) – Starting point.
- **target** (Tuple[float, float]) – Point to indicate direction from starting point to move.
- **distance** (float) – Distance from starting point to returned point.

Return type Tuple[float, float]

Returns A point distance from start in the direction of target.

algoraphics.geom.**points_on_arc**(*center*, *radius*, *theta_start*, *theta_end*, *spacing*)

Generate points along an arc.

Parameters

- **center** (Tuple[float, float]) – The center of the arc.
- **radius** (float) – The radius of the arc.
- **theta_start** (float) – The starting position in degrees.
- **theta_end** (float) – The ending position in degrees.
- **spacing** (float) – The approximate distance between adjacent points.

Return type Sequence[Tuple[float, float]]

Returns A list of points.

algoraphics.geom.**points_on_line**(*start*, *end*, *spacing*)

Generate points along a line.

Parameters

- **start** (Tuple[float, float]) – The first point.
- **end** (Tuple[float, float]) – The last point.
- **spacing** (float) – The approximate (max) distance between adjacent points.

Return type Sequence[Tuple[float, float]]

Returns A list of points.

algoraphics.geom.**rad**(*deg*)
Convert degrees to radians.

Return type float

algoraphics.geom.**remove_close_points**(*points*, *spacing*)
Remove points that are closer than 'spacing'.

A point is removed if it follows the previous point too closely. Consecutive points cannot both be removed, so the list is scanned repeatedly until all consecutive points are at least 'spacing' apart.

Parameters

- **points** (Sequence[Tuple[float, float]]) – A list of points.
- **spacing** (float) – Minimum distance between adjacent points.

algoraphics.geom.**rotate_and_move**(*start*, *ref*, *angle*, *distance*)
Combine rotated_point and move_toward for convenience.

Parameters

- **start** (Tuple[float, float]) – Starting point.
- **ref** (Tuple[float, float]) – A reference point.
- **angle** (float) – The angle in radians to rotate ref around start.
- **distance** (float) – The distance to move from start.

Returns A point distance from start in the angle direction relative to ref.

algoraphics.geom.**rotate_points**(*points*, *pivot*, *angle*)
Rotate points around a reference point.

Parameters

- **points** (Sequence[Sequence[+T_co]]) – A list of points, which can be nested.
- **pivot** (Tuple[float, float]) – The center of rotation.
- **angle** (float) – The angle in radians by which to rotate.

algoraphics.geom.**rotated_point**(*point*, *pivot*, *angle*)
Get the new location of a point after rotating around a reference point.

Parameters

- **point** (Tuple[float, float]) – The starting location.
- **pivot** (Tuple[float, float]) – The center of rotation.
- **angle** (float) – The angle in radians by which to rotate.

Return type Tuple[float, float]

algoraphics.geom.**scale_points**(*points*, *cx*, *cy=None*)
Scale the coordinates of points.

Parameters

- **points** (Sequence[Sequence[+T_co]]) – A list of points, which can be nested.
- **cx** (float) – The horizontal scale factor.
- **cy** (Optional[float]) – The vertical scale factor. If omitted, y-coordinates will be scaled by cx.

algoraphics.geom.**scaled_point**(*point*, *cx*, *cy=None*)
Get the new location of point after scaling coordinates.

Provide either one scaling factor or cx and cy.

Parameters

- **cx** (float) – Either the scaling factor, or if cy is also provided, the horizontal scaling factor.
- **cy** (Optional[float]) – The vertical scaling factor.

Return type Tuple[float, float]

algoraphics.geom.**translate_points**(*points*, *dx*, *dy*)
Shift the location of points.

Parameters

- **points** (Sequence[Sequence[+T_co]]) – A list of points, which can be nested.
- **dx** (float) – Horizontal change.
- **dy** (float) – Vertical change.

algoraphics.geom.**translated_point**(*point*, *dx*, *dy*)
Get a translated point.

Parameters

- **point** (Tuple[float, float]) – The starting location.
- **dx** (float) – The horizontal translation.
- **dy** (float) – The vertical translation.

Return type Tuple[float, float]

2.3 main.py

General functions for creating graphics.

algoraphics.main.**add_margin**(*bounds*, *margin*)
Add margin to bounds.

A convenience function used when generating objects to avoid issues at the edges of the region or canvas.

Parameters

- **bounds** (Tuple[float, float, float, float]) – A tuple of min x, min y, max x, and max y.
- **margin** (float) – The width of the margin.

Return type Tuple[float, float, float, float]

Returns Bounds that include the margin on all sides.

algoraphics.main.**add_shadows**(*objects*, *stdev=10*, *darkness=0.5*)
Add shadows to objects.

Each element (nested or not) of the list is replaced with a group with shadow filter. So items that are shapes will have their own shadow, while an item that is a (nested) list of shapes will have one shadow for the composite object.

Parameters

- **objects** (Sequence[Union[Shape, *Group*, list]]) – A list of shapes (can be nested).
- **stdev** (float) – Standard deviation of the shadow gradient.
- **darkness** (float) – A number below one for lighter shadow, above one for darker.

algographics.main.**filtered**(*obj*, *fltr*)
Apply a filter to one or more shapes.

Parameters

- **obj** (Union[Shape, *Group*, list]) – A shape or (nested) list.
- **fltr** (dict) – A filter.

Return type dict

Returns A group with obj as members and filter applied to group.

algographics.main.**flatten**(*objects*)
Create a flattened list from a nested list.

Parameters **objects** (Any) – A nested list or a non-list.

Return type list

Returns The non-list elements within the input.

algographics.main.**region_background**(*region*, *color*)
Add background color to a clipped region.

Adds a filled rectangle to the beginning of the region's members.

Parameters

- **region** (dict) – A clipped group shape.
- **color** (*Color*) – A color to apply to the region.

algographics.main.**reorder_objects**(*objects*, *by='random'*, *w=None*, *h=None*)
Reorder objects in list.

Used to change order in which objects are drawn.

Parameters

- **objects** (Sequence[Union[Shape, *Group*, list]]) – A list whose items are shapes or lists.
- **by** (str) – 'random' to shuffle objects. 'out to in' to arrange objects closer to the center on top of (later in list) those further from the center. Distance is determined by furthest corner of bounding box so that smaller objects tend to be arranged on top of larger ones that surround them.
- **w** (Optional[float]) – Canvas width, used to get center when by='out to in'.
- **h** (Optional[float]) – Canvas height, used to get center when by='out to in'.

algographics.main.**shuffled**(*items*)
Create shuffled version of a list.

Parameters **items** (Sequence[+T_co]) – A list of items.

Return type list

Returns A new list with same objects as input but reordered.

algoraphics.main.**with_shadow**(*obj*, *stdev*, *darkness*)
Add shadow to an object.

Like add_shadows() but returns a group with a single shadow filter.

Parameters

- **obj** (Union[Shape, *Group*, list]) – A shape or list of objects (can be nested).
- **stdev** (float) – Standard deviation of shadow gradient.
- **darkness** (float) – A number below one for lighter shadow, above one for darker.

Return type dict

Returns A group with obj as members and a filter applied to the group.

2.4 param.py

Define parameter objects that incorporate randomness.

class algoraphics.param.**Dynamic**(*start=None*, *delta=None*, *ratio=None*, *min=None*, *max=None*)
Parameters whose values depend on the previous value.

Provide a Param or number for delta which will be added to the previously generated value to get the next one. Or, provide a ratio Param or number to multiply to each previous value.

Passing a randomized Param object to delta or ratio will result in a random walk. These Param objects can themselves have a delta/ratio argument, resulting in higher-order random walks.

Parameters

- **start** – The starting value, which can be obtained from a Param.
- **delta** – A value to add to the previous value to get the next.
- **ratio** – Similar to delta, but is multiplied by, rather than added to, the previous value to get the next.
- **min** – The smallest allowable value.
- **max** – The largest allowable value.

class algoraphics.param.**Exponential**(*mean=1*, *stdev=1*, *sigma=2*, *static=True*)
Parameters with Exponential distributions.

Parameters

- **mean** (float) – The distribution's mean.
- **stdev** (float) – The distribution's standard deviation.
- **sigma** (float) – How many standard deviations from the mean to clip values.

class algoraphics.param.**Normal**(*mean=0*, *stdev=1*, *static=True*)
Parameters with Gaussian (normal) distributions.

Parameters

- **mean** (float) – The distribution's mean.
- **stdev** (float) – The distribution's standard deviation.

class algoraphics.param.**Param**(*x*, *static=True*)
Objects to represent fixed or random parameters for shapes.

Create `Param` objects for fixed values, value lists, or arbitrary functions. For random distributions, use a specific class that inherits from `Param`.

Parameters **x** (`Union[str, float, list, Callable]`) – A value, list, or a function that takes no arguments and returns a value.

class algoraphics.param.**Uniform**(*min=0*, *max=1*, *static=True*)
Parameters with uniformly random distributions.

Parameters

- **min** (`float`) – The lower bound.
- **max** (`float`) – The upper bound.
- **static** (`bool`) – If set to false, the value will be recomputed for each frame.

algoraphics.param.**fixed_value**(*x*, *t=0*)
Get a fixed value even if a `Param` object is supplied.

Parameters

- **x** (`Union[float, str, Param]`) – An object.
- **t** (`int`) – The current timepoint.

Return type `Union[float, str]`

algoraphics.param.**make_param**(*x*)
Get a `Param` object even if something else is supplied.

Return type *Param*

2.5 point.py

Define objects that represent dynamic 2D points.

class algoraphics.point.**Move**(*ref*, *direction=None*, *distance=0*)

Parameters

- **ref** (`Union[Tuple[float, float], Point]`) – The reference or starting point.
- **direction** (`Optional[Param]`) – A param giving directions (in degrees) of generated points relative to ref. By default the direction is uniformly random.
- **distance** (*Param*) – A param giving distances of generated points relative to ref.

class algoraphics.point.**Point**(*point*)
A representation of a dynamic location in 2D space.

Parameters

- **ref** – The reference or starting point.
- **direction** – A param giving directions (in degrees) of generated points relative to ref. By default the direction is uniformly random.
- **distance** – A param giving distances of generated points relative to ref.

class algoraphics.point.**Rotation**(*start*, *pivot*, *angle*)

class algoraphics.point.**Scaling**(*start*, *cx*, *cy=None*)

class algoraphics.point.**Translation**(*start*, *move*)

algoraphics.point.**make_point**(*x*)
Get a Point object even if a tuple is supplied.

Return type *Point*

2.6 shapes.py

Create and manipulate shapes.

class algoraphics.shapes.**Circle**(*c*, *r*, ***style*)
A circle shape.

Parameters

- **c** (*Point*) – The circle's center.
- **r** (float) – The circle's radius.

class algoraphics.shapes.**Group**(*members=None*, *clip=None*, *filter=None*)
A group of shapes, usually with a clip.

Parameters

- **members** (Union[list, Shape, *Group*, None]) – The shapes in the group.
- **clip** (Union[list, Shape, *Group*, None]) – The shape/s determining where the members are visible.
- **filter** (Optional[dict]) – An SVG filter, e.g. shadow, to apply to the group.

class algoraphics.shapes.**Line**(*p1=None*, *p2=None*, *points=None*, ***style*)
A line or polyline shape.

Supply either p1 and p2 for a line or points for a polyline.

Parameters

- **p1** (Optional[*Point*]) – The starting point.
- **p2** (Optional[*Point*]) – The ending point.
- **points** (Optional[Sequence[*Point*]]) – If a list of points is provided, a polyline is created.

class algoraphics.shapes.**Polygon**(*points*, ***style*)
A polygon shape.

Parameters **points** (Sequence[*Point*]) – A list of polygon vertices.

class algoraphics.shapes.**Spline**(*points*, *smoothing=0.3*, *circular=False*, ***style*)
A spline shape.

Parameters

- **points** (Sequence[*Point*]) – A list of points.
- **smoothing** (float) – The distance to the control point relative to the distance to the adjacent point. Usually between zero and one.
- **circular** (bool) – If False, spline ends reasonably at the first and last points. If True, the ends of the spline will connect smoothly.

algoraphics.shapes.**bounding_box**(*shapes*)
Find the bounding box of a shape or shape collection.

Currently assumes t == 0.

Parameters **shapes** (Union[list, Shape, *Group*]) – One or more shapes.

Return type Tuple[float, float, float, float]

Returns The min x, max x, min y, and max y coordinates of the input.

algoraphics.shapes.**centroid**(*shape*)
Find the centroid of a shape.

Parameters **shape** (Shape) – A shape.

Return type Tuple[float, float]

Returns A point.

algoraphics.shapes.**coverage**(*obj*)
Create a shapely object.

Used to calculate area/coverage.

Parameters **obj** (Union[list, Shape, *Group*]) – One or more shapes.

Return type Union[Polygon, Point, GeometryCollection]

Returns A shapely object representing the union of coverage for all input shapes.

algoraphics.shapes.**keep_points_inside**(*points*, *boundary*)
Keep points that lie within a boundary.

Parameters

- **points** (Sequence[Tuple[float, float]]) – A list of points.
- **boundary** (Union[list, Shape, *Group*]) – One or more shapes giving the boundary.

algoraphics.shapes.**keep_shapes_inside**(*shapes*, *boundary*)
Remove shapes if they lie entirely outside the boundary.

Used to optimize SVG file without altering the appearance.

Parameters

- **shapes** (Sequence[Union[list, Shape, *Group*]]) – A list of shapes, which can be nested.
- **boundary** (Union[list, Shape, *Group*]) – One or more shapes giving the boundary.

algoraphics.shapes.**polygon_area**(*vertices*)
Find the area of a polygon.

Parameters **vertices** (Sequence[Tuple[float, float]]) – The vertex points.

Return type float

Returns The area.

algoraphics.shapes.**rectangle**(*start=None*, *w=None*, *h=None*, *bounds=None*, ***style*)
Create a rectangular Polygon shape.

Provide either start + w + h or a bounds tuple.

Parameters

- **start** (Optional[Tuple[float, float]]) – Bottom left point of the rectangle (unless w or h is negative).
- **w** (Optional[float]) – Width of the rectangle.
- **h** (Optional[float]) – Height of the rectangle.
- **bounds** (Optional[Tuple[float, float, float, float]]) – The (x_min, y_min, x_max, y_max) of the rectangle.

Return type *Polygon*

Returns A polygon shape.

algoraphics.shapes.**remove_hidden**(*shapes*)

Remove shapes from (nested) list if they are entirely covered.

Used to optimize SVG file without altering appearance, e.g. when randomly placing objects to fill a region. Ignores opacity when determining overlap.

Parameters **shapes** (Sequence[Union[list, Shape, *Group*]]) – A list of shapes.

algoraphics.shapes.**rotate_shapes**(*shapes*, *angle*, *pivot=(0, 0)*)

Rotate one or more shapes around a point.

Parameters

- **shapes** (Union[list, Shape, *Group*]) – One or more shapes.
- **angle** (float) – The angle of rotation in degrees.
- **pivot** (Tuple[float, float]) – The rotation pivot point.

algoraphics.shapes.**rotated_bounding_box**(*shapes*, *angle*)

Find the rotated bounding box of a shape or shape collection.

Parameters

- **shapes** (Union[list, Shape, *Group*]) – One or more shapes.
- **angle** (float) – The orientation of the bounding box in degrees.

Return type Tuple[float, float, float, float]

Returns The min x, max x, min y, and max y coordinates in rotated space. Anything created using these coordinates must then be rotated by the same angle around the origin to be in the right place.

algoraphics.shapes.**sample_points_in_shape**(*shape*, *n*)

Sample random points inside a shape.

Parameters

- **shape** (dict) – A shape (currently works for polygons and splines).
- **n** (int) – Number of points to sample.

Return type List[Tuple[float, float]]

Returns The sampled points.

algoraphics.shapes.**scale_shapes**(*shapes*, *cx*, *cy=None*)

Scale one or more shapes.

Parameters

- **shapes** (Union[list, Shape, *Group*]) – One or more shapes.

- **cx** (float) – The horizontal scaling factor.
- **cy** (Optional[float]) – The vertical scaling factor. If missing, cx will be used.

algographics.shapes.**set_style**(*obj*, *attribute*, *value*)
Set style attribute of one or more shapes.

Parameters

- **obj** (Union[list, Shape, *Group*]) – A shape or (nested) list of shapes.
- **attribute** (str) – Name of the style attribute.
- **value** (Union[str, float, *Param*, *Color*]) – Either a single value, Color, or Param.

algographics.shapes.**set_styles**(*obj*, *attribute*, *value*)
Set style attribute of one or more shapes.

Unlike set_style, it creates a deep copy of the Param or Color for each shape so that there is variation.

Parameters

- **obj** (Union[list, Shape, *Group*]) – A shape or (nested) list of shapes.
- **attribute** (str) – Name of the style attribute.
- **value** (Union[*Param*, *Color*]) – A Color or Param.

algographics.shapes.**translate_shapes**(*shapes*, *dx*, *dy*)
Shift the location of one or more shapes.

Parameters

- **shapes** (Union[list, Shape, *Group*]) – One or more shapes.
- **dx** (float) – The horizontal shift.
- **dy** (float) – The vertical shift.

2.7 svg.py

write SVG files.

class algographics.svg.**Canvas**(*width*, *height*, *background='white'*)
A rectangular space to be filled with graphics.

Parameters

- **width** (float) – The canvas width.
- **height** (float) – The canvas height.
- **background** (*Color*) – The background color. If None, background will be transparent.

add(**object*)
Add one or more shapes or collections to the canvas.

clear()
Remove all objects from the canvas.

get_svg()
Get the SVG representation of the canvas as a string.

Return type str

gif (*file_name*, *fps*, *n_frames=None*, *seconds=None*)
Create a GIF image of a dynamic graphic.

Parameters

- **file_name** (str) – The file name to write to.
- **fps** (int) – Frames per second of the GIF.
- **n_frames** (Optional[int]) – Number of frames to generate.
- **seconds** (Optional[float]) – Specify length of the GIF in seconds instead of number of frames.

new (**object*)
Clear the canvas and then add one or more shapes or collections.

png (*file_name*, *force_RGBA=False*)
Write the canvas to a PNG file.

Parameters

- **file_name** (str) – The file name to write to.
- **force_RGBA** (bool) – Whether to write PNG in RGBA colorspace, even if it could be grayscale. This is for, e.g., moviepy which requires all frame images to be in the same colorspace.

svg (*file_name*, *optimize=True*)
Write the canvas to an SVG file.

Parameters

- **file_name** (str) – The file name to write to.
- **optimize** (bool) – Whether to optimize the SVG file using svgo.

algoraphics.svg.**gif** (*function*, *fps*, *file_name*, *n_frames=None*, *seconds=None*)
Create a GIF image from a frame-generating function.

By wrapping typical canvas drawing code in a function, multiple versions of the drawing, each with random variation, can be stitched together into an animated GIF.

Parameters

- **function** (Callable) – A function called with no arguments that returns a (filled) Canvas.
- **fps** (int) – Frames per second of the GIF.
- **file_name** (str) – The file name to write to.
- **n_frames** (Optional[int]) – Number of frames to generate.
- **seconds** (Optional[float]) – Specify length of the GIF in seconds instead of number of frames.

algoraphics.svg.**svg_string** (*objects*, *w*, *h*, *t=0*)
Create an SVG string for a collection of objects.

Parameters

- **objects** (Union[list, dict]) – A (nested) collection of objects. They are placed onto the canvas in order after flattening.
- **w** (float) – Width of the canvas.

- **h** (float) – Height of the canvas.
- **t** (int) – If objects are dynamic, the timepoint to render.

algoraphics.svg.**video**(*function*, *fps*, *file_name*, *n_frames=None*, *seconds=None*)

Create a GIF image from a frame-generating function.

By wrapping typical canvas drawing code in a function, multiple versions of the drawing, each with random variation, can be stitched together into an animated GIF.

Parameters

- **function** (Callable) – A function called with no arguments that returns a (filled) Canvas.
- **fps** (int) – Frames per second of the GIF.
- **file_name** (str) – The file name to write to.
- **n_frames** (Optional[int]) – Number of frames to generate.
- **seconds** (Optional[float]) – Specify length of the GIF in seconds instead of number of frames.

CHAPTER

THREE

EXTRAS API

3.1 fill.py

Fill regions with objects in various ways.

class algoraphics.extras.fill.**Doodle**(*function*, *footprint*)
A Doodle object is a generator of doodles.

Parameters

- **function** (Callable[[], Union[dict, list]]) – A function that takes no arguments and returns a shape or collection.
- **footprint** (ndarray) – A boolean 2D array whose cells indicate the shape (within a grid) occupied by the generated doodles (before being oriented). Row 0 should correspond to the top row of the doodle's footprint.

footprint(*orientation=0*)
Get the doodle's footprint in a given orientation.

Parameters **orientation** (int) – 0 to 7.

Return type ndarray

Returns The oriented footprint.

oriented(*orientation=0*)
Draw the doodle in a given orientation.

Parameters **orientation** (int) – 0 to 7.

Return type Union[dict, list]

Returns The oriented doodle.

algoraphics.extras.fill.**fill_region**(*outline*, *object_fun*, *min_coverage=1*, *max_tries=None*)
Fill a region by iteratively placing randomly generated objects.

Parameters

- **outline** (Union[dict, list]) – A shape or (nested) list of shapes that will become clip.
- **object_fun** (Callable[[Tuple[float, float, float, float]], Union[dict, list]]) – A function that takes bounds as input and returns a randomly generated object.
- **min_coverage** (float) – The minimum fraction of the region's area filled before stopping.

- **max_tries** (Optional[int]) – If not None, the number of objects to generate (including those discarded for not filling space) before giving up and returning the region as is.

Return type *Group*

Returns A group with clip.

algoraphics.extras.fill.**fill_spots**(*outline*, *spacing=10*)
Fill a region with randomly sized spots.

The spots are reminiscent of Ishihara color blindness tests. The spots are not completely non-overlapping, but overlaps are somewhat avoided by spacing out their centers.

Parameters

- **outline** (Union[dict, list]) – A region outline shape.
- **spacing** (float) – The approximate distance between the centers of neighboring spots.

Return type List[*Circle*]

Returns A list of Circle shapes.

algoraphics.extras.fill.**fill_wrapping_paper**(*outline*, *spacing*, *doodles*, *rotate=True*)
Fill a region with a tiling of non-overlapping doodles.

Parameters

- **outline** (Union[dict, list]) – A shape or (nested) list of shapes that will become the clip.
- **spacing** (float) – Height/width of each grid cell.
- **doodles** (Sequence[*Doodle*]) – A list of Doodle objects.
- **rotate** (bool) – Whether to place the grid in a random rotated orientation.

Return type *Group*

Returns A clipped group.

algoraphics.extras.fill.**grid_wrapping_paper**(*rows*, *cols*, *spacing*, *start*, *doodles*)
Create a tiling of non-overlapping doodles.

Parameters

- **rows** (int) – Number of rows to include.
- **cols** (int) – Number of columns to include.
- **spacing** (float) – Height/width of each grid cell.
- **start** (Tuple[float, float]) – Bottom left point of the grid.
- **doodles** (Sequence[*Doodle*]) – A list of Doodle objects.

Return type List[Union[dict, list]]

Returns A list of placed doodle collections.

3.2 grid.py

Functions for working with grids.

algoraphics.extras.grid.**grid_tree**(*rows*, *cols*)
Generate random spanning tree for a grid.

Tree connects adjacent elements of a grid. Used for pixels and for other grids.

Parameters

- **rows** (int) – Number of rows in the grid.
- **cols** (int) – Number of columns in the grid.

Return type ndarray

Returns A 2D binary array in sparse format with dimensions (num. grid cells x num. grid cells) indicating which cells are connected.

algoraphics.extras.grid.**grid_tree_dists**(*rows*, *cols*)
Generate an array of random spanning tree distances.

Each value is the distance to (0, 0) along a spanning tree. Map to a cyclical color gradient to create a billowing effect.

Parameters

- **rows** (int) – Number of rows in the grid.
- **cols** (int) – Number of columns in the grid.

Return type ndarray

Returns A 2D array of integers.

algoraphics.extras.grid.**grid_tree_edges**(*rows*, *cols*)
Generate edges of a random spanning tree for a grid.

Parameters

- **rows** (int) – Number of rows in the grid.
- **cols** (int) – Number of columns in the grid.

Return type Sequence[Tuple[Tuple[int, int], Tuple[int, int]]]

Returns A list of ((r1, c1), (r2, c2)) coordinate tuple pairs.

algoraphics.extras.grid.**grid_tree_neighbors**(*rows*, *cols*)
Generate a random spanning tree for a grid and return neighbor array.

Parameters

- **rows** (int) – Number of rows in grid.
- **cols** (int) – Number of cols in grid.

Return type ndarray

Returns A 3D boolean array (rows x cols x [d?, r?, u?, l?]). The third dimension is length 4 and indicates whether that cell shares an edge with the cell below, right, above, and left of it.

algoraphics.extras.grid.**hsv_array_to_rgb**(*hsv*)
Convert matrix of HSV values to rgb.

Parameters **hsv** (ndarray) – An array of HSV colors.

Return type ndarray

Returns An array of rgb colors.

algoraphics.extras.grid.**map_colors_to_array**(*values*, *colors*, *gradient_mode='rgb'*)
Map 2D array of values to a cyclical color gradient.

If values vary continuously in space, this produces a cyclical color gradient.

Parameters

- **values** (ndarray) – A 2D array of floats. Values should range from 0, inclusive, to len(colors) + 1, exclusive. Each value corresponds to a proportional mixture of the colors at the two indices it is between (with values higher than the last index cycling back to the first color).
- **colors** (Sequence[*Color*]) – A list of Color objects.
- **gradient_mode** (str) – Either 'rgb' or 'hsv' to indicate how the colors are interpolated.

Return type ndarray

Returns A 3D array of RGB values (RGB mode because this is used for PIL images).

algoraphics.extras.grid.**rgb_array_to_hsv**(*rgb*)
Convert matrix of rgb values to HSV.

Parameters **rgb** (ndarray) – An array of rgb colors.

Return type ndarray

Returns An array of HSV colors.

3.3 images.py

Generate graphics based on images.

algoraphics.extras.images.**fill_shapes_from_image**(*shapes*, *image*)
Fill shapes according to their corresponding image region.

Faster than region_color which samples points, but should only be used for regular shapes like tiles since it colors according to the centroid, which may not be inside the shape if it is irregular.

Parameters

- **shapes** – A list of shapes.
- **image** – A PIL image.

algoraphics.extras.images.**image_regions**(*image*, *n_segments=100*, *compactness=10*, *smoothness=0*, *simplify=1*, *expand=2*, *smoothing=0.2*)
Get spline shapes corresponding to image regions.

Parameters

- **image** – A PIL image.
- **n_segments** – Approximate number of desired segments.
- **compactness** – A higher value produces more compact, square-like segments. Try values along log-scale, e.g. 0.1, 1, 10, 100.
- **smoothness** – The width of gaussian smoothing applied before segmentation.
- **simplify** – Maximum distance from the edge of a simplified shape to its actual boundary when reducing the number of points, or None for no simplification.

- **expand** – Number of pixels to expand each segment in every direction to avoid gaps between adjacent shapes.
- **smoothing** – The degree of curvature in spline. Usually between zero and one.

Returns A list of spline shapes, generally in order from left to right and then bottom to top.

algographics.extras.images.**open_image**(*path*)
Load a PIL image from file.

Parameters **path** – Path to the image file.

Returns A PIL Image.

algographics.extras.images.**pad_array**(*pixels*, *margin=1*)
Create a new pixel array with added padding.

Adds additional rows and columns of zeros to four edges of matrix. Used to enable finding a contour around a segment at the edge of an image and to allow segments to be expanded to overlap each other.

Parameters

- **pixels** (ndarray) – A 2D array of pixels.
- **margin** (int) – The width of padding on each side.

Return type ndarray

Returns A 2D array with 2 * margin added to both dimensions.

algographics.extras.images.**region_color**(*outline*, *image*, *n_points=10*)
Find representative color for an image region.

Parameters

- **outline** – A shape corresponding to an image region to sample.
- **image** – A PIL image.
- **n_points** – Number of points to sample.

Returns The average color of the sampled image points.

algographics.extras.images.**resize_image**(*image*, *width*, *height*)
Downscale an image.

Scales according to whichever of width or height is not None. Only scales down.

Parameters

- **image** – A PIL image.
- **width** – The new width. If None, scales according to height.
- **height** – The new height. If None, scales according to width.

algographics.extras.images.**sample_colors**(*image*, *points*)
Sample colors from an image.

Parameters

- **image** – A PIL image.
- **points** – A list of image coordinates, or a single coordinate.

Returns A list of colors corresponding to points, or a single color if input is a single point.

3.4 mazes.py

Functions for creating maze-like patterns.

class algoraphics.extras.mazes.**Maze_Style**
Specifications for maze styles.

Built-in styles inherit from this class, and custom styles can be made to do so as well. It must implement the tip, turn, straight, T, and cross methods to describe how each component is drawn.

Each of these methods should generate forms for one cell of the maze. They must provide one or more pieces which are stitched together when the maze is assembled. For example, for a cell in which the maze turns right, the inner and outer edges of the curve are generated. Once this cell is encountered in the stitching process, the inner edge is added, then everything from the cell to the right is recursively added, and then the outer edge is added to get one continuous edge.

For all methods the edge enters below and draws the component counter-clockwise. During maze generation components are generated and then rotated as needed to accomodate any combination of neighbors that the cell should connect to.

T()
Generate a 'T'-shaped cell.

This should return a tuple of three lists of path commands corresponding to the right inner edge, top edge, and then left inner edge.

cross()
Generate a cell connecting in all directions.

This should return a tuple of four lists of path commands corresponding to the lower-right, upper-right, upper-left, and lower-left inner edges.

output(*points*)
Make the output shape/s from the generated points.

This takes the maze outline points and produces the intended shape/s, e.g. polygon or spline. Returns a polygon by default.

straight()
Generate a non-turning cell.

This should return a tuple of two lists of path commands corresponding to the right and then left edges.

tip()
Generate a dead-end cell.

This should return a list of path command dictionaries.

turn()
Generate a right turn cell.

This should return a tuple of two lists of path commands corresponding to the inner and then outer edges.

Left turns will be drawn with this and then rotated.

class algoraphics.extras.mazes.**Maze_Style_Jagged**(*min_w*, *max_w*)
Generate pieces for jagged maze.

Parameters

- **min_w** (float) – Minimum width of channel segment relative to cell width.
- **max_w** (float) – Maximum width of channel segment relative to cell width.

output(*points*)
Make the output shape/s from the generated points.

This takes the maze outline points and produces the intended shape/s, e.g. polygon or spline. Returns a polygon by default.

class algoraphics.extras.mazes.**Maze_Style_Pipes**(*rel_thickness*)
Generate pieces for curved pipes.

Parameters **rel_thickness** (float) – Channel width relative to cell width, from 0 to 1.

output(*points*)
Make the output shape/s from the generated points.

This takes the maze outline points and produces the intended shape/s, e.g. polygon or spline. Returns a polygon by default.

class algoraphics.extras.mazes.**Maze_Style_Round**(*rel_thickness*)
Generate pieces for very curvy pipes.

Parameters **rel_thickness** (float) – Channel width relative to cell width, from 0 to 1.

output(*points*)
Make the output shape/s from the generated points.

This takes the maze outline points and produces the intended shape/s, e.g. polygon or spline. Returns a polygon by default.

class algoraphics.extras.mazes.**Maze_Style_Straight**(*rel_thickness*)
Generate pieces for simple right-angle maze.

Parameters **rel_thickness** (float) – Channel width relative to cell width, from 0 to 1.

output(*points*)
Make the output shape/s from the generated points.

This takes the maze outline points and produces the intended shape/s, e.g. polygon or spline. Returns a polygon by default.

algoraphics.extras.mazes.**fill_maze**(*outline*, *spacing*, *style*, *rotation=None*)
Fill a region with a maze-like pattern.

Parameters

- **outline** (Union[list, Shape, *Group*]) – The shape/s that will become the clip.
- **spacing** (float) – The cell width of the grid.
- **style** (*Maze_Style*) – An object specifying how the maze path is to be drawn.
- **rotation** (Optional[float]) – The orientation of the grid in degrees.

Return type dict

Returns A group with clip.

algoraphics.extras.mazes.**maze**(*rows*, *cols*, *spacing*, *start*, *style*)
Generate a maze-like pattern spanning the specified grid.

Parameters

- **rows** (int) – Number of rows in the grid.
- **cols** (int) – Number of columns in the grid.
- **spacing** (float) – The cell width.

- **start** (Tuple[float, float]) – The bottom-left coordinate of the grid.
- **style** (*Maze_Style*) – An object specifying how the maze path is to be drawn.

Return type Union[list, Shape, *Group*]

Returns A shape (usually a spline or polygon) or collection.

3.5 ripples.py

Create space-filling ripple effects.

algoraphics.extras.ripples.**ripple_canvas**(*w*, *h*, *spacing*, *trans_probs=None*, *existing_pts=None*)

Fill the canvas with ripples.

The behavior of the ripples is determined by a first-order Markov chain in which events correspond to points along splines. The states are 'S', 'R', 'L', and 'X'. At 'S', the ripple begins in a random direction. At 'R', the ripple turns right sharply until encountering a ripple or other barrier, and then follows along it. Likewise with 'L' turning left. At 'X', the ripple moves straight forward +/- up to 60 degrees. Higher state-changing transition probabilities result in more erratic ripples.

Parameters

- **w** (float) – Width of the canvas.
- **h** (float) – Height of the canvas.
- **spacing** (float) – Distance between ripples.
- **trans_probs** (Optional[Dict[str, Dict[str, float]]]) – A dictionary of dictionaries containing Markov chain transition probabilities from one state (first key) to another (second key).
- **existing_pts** (Optional[Sequence[Tuple[float, float]]]) – An optional list of points that ripples will avoid.

Return type List[dict]

Returns The ripple splines.

3.6 structures.py

Create structures such as filaments and trees.

algoraphics.extras.structures.**blow_paint_area**(*points*, *spacing=20*, *length=40*, *len_dev=0.25*, *width=5*)

Draw a blow-paint effect around an area.

Creates 'fingers' of paint projecting from each edge, as if being blown along the page perpindicular to the edge.

Parameters

- **points** (Sequence[Tuple[float, float]]) – The vertices of the polygonal area.
- **spacing** (float) – Average distance between paint fingers.
- **length** (float) – Average length of the paint fingers.
- **len_dev** (float) – The standard deviation of finger lengths relative to length (so it should be less than 1).

- **width** (float) – Average thickness of each finger.

Return type dict

algoraphics.extras.structures.**blow_paint_line**(*points*, *line_width=10*, *spacing=20*, *length=20*, *len_dev=0.33*, *width=5*)

Draw a blow-paint effect connecting a sequence of points.

Creates 'fingers' of paint projecting from each edge, as if being blown along the page perpindicular to the edge (in both directions).

Parameters

- **points** (Sequence[Tuple[float, float]]) – The points to connect.
- **line_width** (float) – The thickness of the line (excluding the fingers).
- **spacing** (float) – Average distance between paint fingers.
- **length** (float) – Average length of the paint fingers.
- **len_dev** (float) – The standard deviation of finger lengths relative to length (so it should be less than 1).
- **width** (float) – Average thickness of each finger.

Return type dict

algoraphics.extras.structures.**blow_paint_spot**(*point*, *length=10*, *len_dev=0.7*, *width=3*)

Draw a paint splatter.

Creates 'fingers' of paint projecting from a point.

Parameters

- **point** (Tuple[float, float]) – The center of the splatter.
- **length** (float) – Average length of the paint fingers.
- **len_dev** (float) – The standard deviation of finger lengths relative to length (so it should be less than 1).
- **width** (float) – Average thickness of each finger.

Return type dict

algoraphics.extras.structures.**filament**(*backbone*, *width*)

Generate a meandering segmented filament.

Parameters

- **backbone** (Sequence[*Point*]) – A list of Points specifying the midpoint of the filament ends and segment boundaries.
- **width** (*Param*) – The width/s of the filament (at segment joining edges).

Return type List[*Polygon*]

Returns A list of Polygons (the segments from start to end).

algoraphics.extras.structures.**tentacle**(*backbone*, *width*)

Generate a filament that tapers to a point.

Parameters

- **backbone** (Sequence[*Point*]) – A list of Points specifying the midpoint of the filament ends and segment boundaries.
- **width** (float) – The width of the tentacle base.

Return type List[*Polygon*]

Returns A list of polygons (the segments from base to tip).

algoraphics.extras.structures.**tree**(*start*, *direction*, *branch_length*, *theta*, *p*, *delta_p=0*)

Generate a tree with randomly terminating branches.

Parameters

- **start** (*Point*) – The starting point.
- **direction** (*Param*) – The starting direction (in degrees).
- **branch_length** (*Param*) – Branch length.
- **theta** (*Param*) – The angle (in degrees) between sibling branches.
- **p** (float) – The probability that a given branch will split instead of terminating. Recommended to have a delta < 0 or ratio < 1 so that the tree is guaranteed to terminate.
- **delta_p** (float) – The decrease in p at each branching.

Return type List[dict]

Returns A list of line shapes.

3.7 text.py

Generate text in the form of shapes or SVG text.

algoraphics.extras.text.**char_points**(*char*, *start*, *h*, *spacing*)

Generate points along a character shape.

Parameters

- **char** (str) – A character.
- **start** (Tuple[float, float]) – The lower-left point of the character bounds.
- **h** (float) – The line height.
- **spacing** (float) – Distance between adjacent points.

Return type List[List[Tuple[float, float]]]

Returns A list of lists of points. Each list of points corresponds to a pen stroke (i.e. what can be drawn without lifting the pen).

algoraphics.extras.text.**text_points**(*text*, *height*, *pt_spacing*, *char_spacing=0.1*, *grouping='points'*)

Generate points that spell out text.

Text starts at (0, 0) and should be repositioned.

Parameters

- **text** (str) – The text.
- **height** (float) – Line height.
- **pt_spacing** (float) – Approximate distance between adjacent points.
- **char_spacing** (float) – Size of the largest space on either side of each character relative to line height. This space is shrunk for certain characters depending on shape.
- **grouping** (str) – 'points' to return list of points, 'strokes' to group by stroke.

Return type Union[List[Tuple[float, float]], List[List[Tuple[float, float]]]]

Returns Either a list of points or a list of lists of points in which each list of points corresponds to a pen stroke (i.e. what can be drawn without lifting the pin).

3.8 tiling.py

Fill a region or the canvas with tiles or webs.

algoraphics.extras.tiling.**delaunay_edges**(*points*)
Find edges of Delaunay regions for a set of points.

Parameters **points** (Sequence[Tuple[float, float]]) – A list of points.

Return type List[dict]

Returns A list of line shapes.

algoraphics.extras.tiling.**delaunay_regions**(*points*)
Find the Delaunay regions for a set of points.

Parameters **points** (Sequence[Tuple[float, float]]) – A list of points.

Return type List[dict]

Returns A list of triangular polygon shapes. Items do not correspond to input points because points on the periphery do not have finite regions.

algoraphics.extras.tiling.**fill_nested_triangles**(*outline*, *min_level*, *max_level*, *color=None*, *color2=None*)
Fill region with nested triangle pattern.

Parameters

- **outline** (Union[list, Shape, *Group*]) – A region outline shape or collection.
- **min_level** (int) – The level of the largest triangles (0 is bounding triangle).
- **max_level** (int) – The level of the smallest triangles.
- **color1** – The color/s for half of the triangles.
- **color2** (Optional[*Color*]) – The color for the opposing half of the triangles. This half of triangles will all be one color because it is the background.

Return type dict

Returns A group with the outline as clip.

algoraphics.extras.tiling.**nested_triangles**(*tip*, *height*, *min_level*, *max_level*)
Generate nested equilateral triangles.

Parameters

- **tip** (Tuple[float, float]) – The tip of the bounding triangle.
- **height** (float) – The height of the bounding triangle (negative for upside-down triangle).
- **min_level** (int) – The level of the largest triangles (0 is the bounding triangle).
- **max_level** (int) – The level of the smallest triangles.

Return type List[dict]

Returns A list of triangle polygon shapes.

algoraphics.extras.tiling.**tile_canvas**(*w*, *h*, *shape='polygon'*, *edges=False*, *tile_size=500*, *regularity=10*)

Fill canvas with (uncolored) tiles.

Parameters

- **w** (float) – Width of the canvas.
- **h** (float) – Height of the canvas.
- **tile_shape** – Either 'polygon' for Voronoi tiling or 'triangle' for Delaunay tiling.
- **edges** (bool) – Whether to return the edges around tiles as lines instead of the tiles themselves.
- **tile_size** (float) – The approximate area of each tile.
- **regularity** (int) – A value of one or higher, passed to spaced_points.

Return type List[dict]

Returns A list of polygon or line shapes.

algoraphics.extras.tiling.**tile_region**(*outline*, *shape='polygon'*, *edges=False*, *tile_size=500*, *regularity=10*)

Fill region with (uncolored) tiles or tile edges.

Parameters

- **outline** (Union[list, Shape, *Group*]) – The shape/s that will become the clip.
- **shape** (str) – Either 'polygon' for Voronoi tiling or 'triangle' for Delaunay tiling.
- **edges** (bool) – Whether to return the edges around tiles as lines instead of the tiles themselves.
- **tile_size** (float) – The approximate area of each tile.
- **regularity** (int) – A value of one or higher, passed to spaced_points().

Return type dict

Returns A group with clip.

algoraphics.extras.tiling.**voronoi_edges**(*points*)

Find the edges of Voronoi regions for a set of points.

Parameters **points** (Sequence[Tuple[float, float]]) – A list of points.

Return type List[dict]

Returns A list of line shapes.

algoraphics.extras.tiling.**voronoi_regions**(*points*)

Find Voronoi regions for a set of points.

Parameters **points** (Sequence[Tuple[float, float]]) – A list of points.

Return type List[dict]

Returns A list of polygon shapes. Items do not correspond to input points because points on the periphery do not have finite regions.

3.9 utils.py

Functions used in the extras library.

class algographics.extras.utils.**Rtree**(*points=None*)
An object to efficiently query a field of points.

Parameters **points** (Optional[Sequence[Tuple[float, float]]]) – Starting points.

add_point(*point*)
Add a point to the collection.

Parameters **point** (Tuple[float, float]) – The new point.

add_points(*points*)
Add points to the collection.

Parameters **points** (Sequence[Tuple[float, float]]) – The new points.

nearest(*point*, *n=1*, *index=False*)
Get the nearest point or points to a query point.

Parameters

- **point** (Tuple[float, float]) – A query point.
- **n** (int) – Number of nearest points to return.
- **index** (bool) – Whether to return the nearets points' indices instead of the points themselves.

Return type Union[Sequence[Tuple[float, float]], Tuple[float, float]]

Returns If n is 1, the nearest point, otherwise a list of nearest points.

algographics.extras.utils.**contrasting_lightness**(*color*, *light_diff*)
Get color with contrasting lightness to reference color.

Color is lighter if original lightness is < 0.5 and darker otherwise. Used to create color pairs for a mixture of light and dark colors.

Parameters

- **color** (*Color*) – A color.
- **light_diff** (float) – Magnitude of difference in lightness, between 0 and 1.

Return type *Color*

Returns The contrasting color.

algographics.extras.utils.**spaced_points**(*n*, *bounds*, *n_cand=10*)
Generate random but evenly-spaced points.

Uses Mitchell's best-candidate algorithm.

Parameters

- **n** (int) – Number of points to generate.
- **bounds** (Tuple[float, float, float, float]) – A bounds tuple.
- **n_cand** (int) – Number of candidate points to generate for each output point. Higher numbers result in higher regularity.

Return type List[Tuple[float, float]]

Returns The generated points.

algoraphics.extras.utils.**wobble**(*shapes*, *dev=2*)

Add a little messiness to perfect lines and curves.

Convert straight lines and curves into slightly wavy splines. Currently it converts the shape into a shape with fixed parameters, since it has to do point interpolation. The new shape's parameters can later be made dynamic.

Parameters

- **obj** – A list of one or more shapes (can be nested).
- **dev** (float) – The (approximate) maximum distance a part of an edge will move.

CHAPTER

FOUR

GLOSSARY

bounds A tuple giving the minimum x, minimum y, maximum x, and maximum y coordinate of a *shape*, *collection*, or general rectangular space.

collection A shape or a list containing *shapes*, which can be nested in lists.

doodle An abstraction of a small drawing. It can vary each time it is drawn. A fill function can draw it in different orientations and locations to fill a *region*.

image A PIL Image object specifying a grid of pixels. These are usually used as templates, but can also be drawn using a `raster` *shape*.

margin The space added in all four directions to the *bounds* of some area to improve visual continuity. For example, a tiling or other filling pattern should extend beyond the canvas or *region* edges to avoid edge artifacts or gaps.

outline A *shape* or *collection* specifying an area in which some function will draw.

parameter A specification for some attribute of a drawing. A Param (1D) or Place (2D) object, or one whose type inherits from them, can represent the attribute so that the attribute can vary within the drawing, or so that multiple *shapes* can be drawn without having to explicitly generate random values for each one.

point A tuple containing the x and y coordinates of a point in 2D space. The units are pixels by default, but can be defined with floats.

region A drawing occupying a specified area. It is generally represented as a group *shape* clipped (i.e., contains the `clip` attribute) by an *outline*. It is often created using either a *bounds* specification or by segmenting an *image*.

segment An area of an *image* that is contiguous and visually distinct and can form the basis of a *region*.

shape A visual object that is specified by a single SVG element. It is represented by a dictionary with a `type` attribute specifying the type of shape, and other attributes defining its *parameters*.

style A visual attribute of a *shape* that is stored in a dictionary in the shape's `style` attribute. It usually corresponds to an SVG attribute, and any SVG attribute can be specified.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

R

S

T

U

V

W