

Package: sketchpad (via r-universe)

September 27, 2026

Title A Minimal Drawing System For Generative Art

Version 0.0.0.9000

Description A lightweight drawing system built on S7 classes and grid graphics that can be used for generative art.

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 8.0.0

URL <https://github.com/djnavarro/sketchpad>,
<https://sketchpad.djnavarro.net/>

BugReports <https://github.com/djnavarro/sketchpad/issues>

Imports ambient, grDevices, grid, purrr, rlang, S7, stats, withr

Suggests covr, testthat (>= 3.0.0), tibble, waepowifestre

Config/testthat/edition 3

Remotes djnavarro/waepowifestre

Collate 'utils.R' 'fill.R' 'palette.R' 'color.R' 'noise_field.R'
'noise_bridge.R' 'trans.R' 'style.R' 'xy.R' 'drawable.R'
'shape_bezier.R' 'curve_bezier.R' 'curve_line.R'
'curve_spiral.R' 'curve_scribble.R' 'shape_raw.R' 'curve_raw.R'
'points_raw.R' 'shape_combine.R' 'shape_circle.R'
'shape_rectangle.R' 'shape_polygon.R' 'shape_star.R'
'shape_ellipse.R' 'shape_wedge.R' 'curve_arc.R' 'shape_blob.R'
'shape_ribbon.R' 'shape_twist.R' 'curve_twist.R'
'shape_stroke.R' 'shape_strokepath.R' 'canvas.R' 'sketch.R'
'format.R' 'vectorize.R' 'effects.R' 'effect_tremor.R'
'effect_bristle.R' 'effect_swatch.R' 'draw.R' 'group.R'
'effect_grain.R' 'convert.R' 'save.R' 'sketchpad-package.R'

Depends R (>= 3.6)

Repository <https://djnavarro.r-universe.dev>

Date/Publication 2026-08-23 09:05:54 UTC

RemoteUrl <https://github.com/djnavarro/sketchpad>

RemoteRef HEAD

RemoteSha a399dce446427f733491fc791396d0d2cb39a106

Contents

canvas	4
color_alpha	5
color_greyscale	6
color_invert	7
color_jitter	7
color_lighten	8
color_mix	9
color_rotate_hue	10
color_saturate	11
curve_arc	11
curve_bezier	13
curve_line	15
curve_raw	16
curve_scribble	18
curve_spiral	20
curve_twist	22
draw	24
drawable	25
effect_bristle	27
effect_grain	29
effect_swatch	31
effect_tremor	32
fill	34
fill_charcoal	35
fill_checker	36
fill_crosshatch	38
fill_flow	40
fill_gradient	42
fill_halftone	44
fill_hatch	45
fill_image	47
fill_marble	49
fill_noise	51
fill_none	53
fill_scatter	54
fill_scribble	56
fill_solid	59
fill_stipple	60
fill_stripe	62
fill_vignette	64
group	65

noise_bridge	67
noise_field	68
noise_sample	69
palette_cosine	70
palette_gradient	71
palette_harmony	72
palette_manual	73
palette_tonal	74
palette_walk	75
points_raw	76
save_png	77
shape_bezier	79
shape_blob	80
shape_circle	82
shape_combine	84
shape_ellipse	85
shape_polygon	87
shape_raw	88
shape_rectangle	90
shape_ribbon	92
shape_star	94
shape_stroke	96
shape_strokepath	98
shape_twist	100
shape_wedge	103
sketch	105
style	106
trans	109
trans_affine	110
trans_chain	110
trans_fn	111
trans_identity	113
trans_reflect	113
trans_rotate	114
trans_scale	115
trans_shear	117
trans_translate	118
trans_warp	119
xy	120

 canvas

Canvas settings for a sketch

Description

canvas bundles the settings `draw()` applies to a `sketch` as a whole, before drawing any of its shapes: a background fill and an optional fixed `xlim/ylim` frame. It plays the same role for sketch that `style()` plays for a single `drawable` – a small, reusable settings object, validated independently and stored as a property (sketch’s own `canvas`) rather than a flat list of arguments.

Usage

```
canvas(background = fill_none(), xlim = NULL, ylim = NULL, clip = FALSE)
```

Arguments

background	Background fill, drawn beneath every shape in the sketch. Either a plain colour string, or the output of a <code>fill_*</code> () helper – see <code>style()</code> ’s <code>fill</code> argument for the full family (a bare colour string/ <code>GridPattern</code> is coerced into a <code>fill</code> object automatically). Default <code>fill_none()</code> (no background drawn; the page’s own background shows through, matching <code>draw()</code> ’s behavior before <code>canvas</code> existed).
xlim, ylim	Fixed axis limits, each a numeric vector of length 2, or <code>NULL</code> (the default) to compute them from the sketch’s own shapes at draw time, as <code>draw()</code> already did before <code>canvas</code> existed. An explicit <code>xlim/ylim</code> passed to <code>draw()</code> itself always takes precedence over these.
clip	Whether to clip content to <code>xlim/ylim</code> . Must be a single logical. Default <code>FALSE</code> – see Details.

Details

`xlim/ylim` only fix the coordinate *scale* `draw()` maps onto the page – they do not, by themselves, clip content that falls outside that range; a shape wider than its sketch’s `canvas` still renders in full, spilling past the frame, exactly as passing `xlim/ylim` to `draw()` directly already behaves. This is a deliberate default: shapes built from a `noise_field/noise_bridge` (e.g. `shape_blob()`, `shape_twist()`) can have somewhat unpredictable extents, and silently cropping part of one away is a worse failure mode than a visibly overflowing shape, which is an obvious cue to adjust the sketch’s parameters. Set `clip = TRUE` to opt into hard clipping at `xlim/ylim` instead – most useful once background is also set to something other than `fill_none()`, since an opaque background otherwise has visible content bleeding past its own edge onto the bare page. `clip` has no visible effect when `xlim/ylim` are both left `NULL`, since `draw()` then computes them from the sketch’s own shapes, which by construction never exceed that range.

See Also

Other core structure: `draw()`, `drawable()`, `group()`, `sketch()`, `style()`, `xy()`

Examples

```

canvas()
canvas(background = "grey90")

# a background is drawn beneath every shape in the sketch
draw(sketch(canvas = canvas(background = "grey90")) + shape_blob(radius = 1))

# xlim/ylim alone only fix the coordinate scale -- a shape wider than the
# frame still overflows it
overflowing <- sketch(canvas = canvas(
  background = "white", xlim = c(-1, 1), ylim = c(-1, 1)
)) + shape_circle(radius = 1.4)
draw(overflowing)

# clip = TRUE hard-clips content at xlim/ylim instead
overflowing@canvas@clip <- TRUE
draw(overflowing)

```

color_alpha	<i>Set a colour vector's alpha (opacity)</i>
-------------	--

Description

Multiplies each colour's own alpha channel (1, i.e. fully opaque, for a colour with none already set) by alpha, via `grDevices::adjustcolor()` – the same mechanism `style()`'s own `color_alpha`/`fill_alpha` properties use internally at `draw()` time (see `?style`). Since an ordinary opaque colour's own alpha is 1, this has the effect of setting the result's alpha to alpha directly for any color with no alpha of its own already.

Usage

```
color_alpha(color, alpha = 0.5)
```

Arguments

color	A character vector of colours.
alpha	A numeric vector, each in $[0, 1]$, recycled against color.

Value

A character vector of hex colour strings (including an alpha channel), the same length as the longer of color/alpha.

See Also

Other color helpers: `color_greyscale()`, `color_invert()`, `color_jitter()`, `color_lighten()`, `color_mix()`, `color_rotate_hue()`, `color_saturate()`

Examples

```
color_alpha("black", alpha = 0.5)
color_alpha(palette_manual(index = 1), alpha = c(1, 0.75, 0.5, 0.25, 0.1))
```

color_greyscale	<i>Convert a colour vector to greyscale</i>
-----------------	---

Description

A thin wrapper around [color_desaturate\(\)](#) with amount = 1 – every colour’s HSL saturation is dropped to 0, leaving only its own lightness. This is a plain HSL desaturation, not a perceptual (luminance-weighted) greyscale conversion – two colours with very different perceived brightness but the same HSL lightness (e.g. a saturated yellow and a saturated blue) convert to the same grey.

Usage

```
color_greyscale(color)
```

Arguments

color	A character vector of colours.
-------	--------------------------------

Value

A character vector of hex colour strings, the same length as color.

See Also

Other color helpers: [color_alpha\(\)](#), [color_invert\(\)](#), [color_jitter\(\)](#), [color_lighten\(\)](#), [color_mix\(\)](#), [color_rotate_hue\(\)](#), [color_saturate\(\)](#)

Examples

```
color_greyscale(palette_manual(index = 1))
draw(effect_swatch(color = color_greyscale(palette_manual(index = 1))))
```

color_invert	<i>Invert a colour vector</i>
--------------	-------------------------------

Description

Inverts each colour's RGB channels (1 - channel), the classic "photo negative" transform. Unlike `color_rotate_hue(degrees = 180)`, which keeps HSL saturation/lightness fixed and only flips hue, RGB inversion also flips lightness (e.g. white inverts to black, not to another light colour).

Usage

```
color_invert(color)
```

Arguments

`color` A character vector of colours.

Value

A character vector of hex colour strings, the same length as `color`.

See Also

Other color helpers: [color_alpha\(\)](#), [color_greyscale\(\)](#), [color_jitter\(\)](#), [color_lighten\(\)](#), [color_mix\(\)](#), [color_rotate_hue\(\)](#), [color_saturate\(\)](#)

Examples

```
color_invert("white")
color_invert(palette_manual(index = 1))
draw(effect_swatch(color = color_invert(palette_manual(index = 1))))
```

color_jitter	<i>Randomly perturb a colour vector</i>
--------------	---

Description

Nudges each colour's HSL hue/saturation/lightness by independent Gaussian noise – hue by amount * 360 degrees (wrapped, %% 360), saturation/lightness by amount (reflected back into [0, 1] at the boundary via the same triangle-wave technique [palette_walk\(\)](#) uses for its own random steps) – giving each colour an independent, organic variation rather than [palette_walk\(\)](#)'s own *correlated* sequence of colours.

Usage

```
color_jitter(color, amount = 0.05, seed = 1L)
```

Arguments

color	A character vector of colours.
amount	A single non-negative number: the standard deviation of the saturation/lightness noise (and amount * 360, of the hue noise, in degrees).
seed	A single whole number seeding the noise, so the same seed always produces the same perturbation.

Value

A character vector of hex colour strings, the same length as color.

See Also

Other color helpers: [color_alpha\(\)](#), [color_greyscale\(\)](#), [color_invert\(\)](#), [color_lighten\(\)](#), [color_mix\(\)](#), [color_rotate_hue\(\)](#), [color_saturate\(\)](#)

Examples

```
color_jitter(palette_manual(index = 1), amount = 0.05, seed = 8241)
draw(effect_swatch(color = color_jitter(rep("steelblue", 8), amount = 0.1, seed = 8241)))
```

color_lighten	<i>Lighten or darken a colour vector</i>
---------------	--

Description

Shifts each colour's HSL lightness by amount, clamped to [0, 1]. `color_lighten()` and `color_darken()` shift in opposite directions by the same amount – there is no single signed-amount function, since separate verbs read more clearly at a call site.

Usage

```
color_lighten(color, amount = 0.1)
```

```
color_darken(color, amount = 0.1)
```

Arguments

color	A character vector of colours.
amount	A numeric vector, each in [0, 1], recycled against color: the (clamped) HSL lightness shift.

Value

A character vector of hex colour strings, the same length as the longer of color/amount.

See Also

Other color helpers: [color_alpha\(\)](#), [color_greyscale\(\)](#), [color_invert\(\)](#), [color_jitter\(\)](#), [color_mix\(\)](#), [color_rotate_hue\(\)](#), [color_saturate\(\)](#)

Examples

```
color_lighten("steelblue", amount = 0.2)
color_darken(palette_manual(index = 1), amount = 0.15)
draw(effect_swatch(color = color_lighten("steelblue", amount = seq(0, 0.4, 0.1))))
```

color_mix

*Blend two colour vectors***Description**

Mixes color1 and color2 channel-by-channel in linear RGB space, amount of the way from color1 to color2 (amount = 0 returns color1 unchanged, amount = 1 returns color2). Unlike [palette_gradient\(\)](#) (which interpolates *through* a sequence of colours to produce a longer palette), color_mix() blends two same-length (or recyclable) vectors pointwise – useful for tinting an existing palette toward a shared accent colour. As with [palette_gradient\(\)](#)'s own space argument, space = "Lab" blends perceptually more evenly than "rgb", particularly for color1/ color2 pairs far apart in hue (a plain RGB blend of two saturated, near-complementary colours often passes through a dull, muddy midpoint that a Lab blend avoids) – but unlike "rgb", which always stays in gamut by construction, a "Lab" blend's midpoint can occasionally land just outside the representable sRGB gamut; this is clipped back into [0, 1] per channel rather than erroring.

Usage

```
color_mix(color1, color2, amount = 0.5, space = c("rgb", "Lab"))
```

Arguments

color1, color2	Character vectors of colours, recycled against each other and against amount.
amount	A numeric vector, each in [0, 1], recycled against color1/color2: the proportion of color2 in the blend.
space	The colour space the blend happens in: "rgb" (the default) linearly interpolates raw RGB channels; "Lab" interpolates in CIE Lab space instead, via grDevices::convertColor() .

Value

A character vector of hex colour strings, the same length as the longest of color1/color2/amount.

See Also

Other color helpers: [color_alpha\(\)](#), [color_greyscale\(\)](#), [color_invert\(\)](#), [color_jitter\(\)](#), [color_lighten\(\)](#), [color_rotate_hue\(\)](#), [color_saturate\(\)](#)

Examples

```
color_mix("steelblue", "tomato", amount = 0.5)
color_mix(palette_manual(index = 1), "white", amount = 0.3)
color_mix("tomato", "seagreen", amount = 0.5, space = "Lab")
draw(effect_swatch(color = color_mix("steelblue", "tomato", amount = seq(0, 1, 0.2))))
```

color_rotate_hue	<i>Rotate a colour vector's hue</i>
------------------	-------------------------------------

Description

Adds degrees to each colour's HSL hue, wrapping around the colour wheel (%% 360). Generalizes the fixed hue offsets [palette_harmony\(\)](#) uses internally into a plain, arbitrary-angle operation on an existing colour vector.

Usage

```
color_rotate_hue(color, degrees = 30)
```

Arguments

color	A character vector of colours.
degrees	A numeric vector (degrees), recycled against color: the hue rotation, positive or negative, of any magnitude.

Value

A character vector of hex colour strings, the same length as the longer of color/degrees.

See Also

Other color helpers: [color_alpha\(\)](#), [color_greyscale\(\)](#), [color_invert\(\)](#), [color_jitter\(\)](#), [color_lighten\(\)](#), [color_mix\(\)](#), [color_saturate\(\)](#)

Examples

```
color_rotate_hue("tomato", degrees = 180)
color_rotate_hue(palette_manual(index = 1), degrees = c(0, 30, 60, 90, 120))
draw(effect_swatch(color = color_rotate_hue("tomato", degrees = seq(0, 300, 60))))
```

color_saturate	<i>Saturate or desaturate a colour vector</i>
----------------	---

Description

Shifts each colour's HSL saturation by amount, clamped to [0, 1]. `color_saturate()` and `color_desaturate()` shift in opposite directions by the same amount, mirroring `color_lighten()`/`color_darken()`'s own paired-verb shape.

Usage

```
color_saturate(color, amount = 0.1)

color_desaturate(color, amount = 0.1)
```

Arguments

color	A character vector of colours.
amount	A numeric vector, each in [0, 1], recycled against color: the (clamped) HSL saturation shift.

Value

A character vector of hex colour strings, the same length as the longer of color/amount.

See Also

Other color helpers: `color_alpha()`, `color_greyscale()`, `color_invert()`, `color_jitter()`, `color_lighten()`, `color_mix()`, `color_rotate_hue()`

Examples

```
color_saturate("slategray", amount = 0.3)
color_desaturate(palette_manual(index = 1), amount = 0.5)
draw(effect_swatch(color = color_desaturate(palette_manual(index = 1), amount = 0.5)))
```

curve_arc	<i>An open arc</i>
-----------	--------------------

Description

`curve_arc` is `shape_wedge()`'s arc alone, with no centroid vertex: an open path of n points on the circle centred at (x, y) with the given radius, sweeping from angle start to end (radians).

`curve_arcs()` is a vectorized version of `curve_arc()`: each argument may be a vector, recycled against the others. The result is a `sketch` containing one `curve_arc()` per recycled row, rather than a single drawable.

Usage

```

curve_arc(
  x = 0,
  y = 0,
  radius = 1,
  start = 0,
  end = pi/2,
  n = 100L,
  trans = trans_identity(),
  ...
)

curve_arcs(
  x = 0,
  y = 0,
  radius = 1,
  start = 0,
  end = pi/2,
  n = 100L,
  trans = trans_identity(),
  ...
)

```

Arguments

<code>x, y</code>	Centroid coordinates. Default 0.
<code>radius</code>	Radius. Must be non-negative. Default 1.
<code>start, end</code>	Start/end angle of the arc, in radians. Default 0/ pi / 2.
<code>n</code>	Number of points used to approximate the arc. Must be at least 2. Default 100L.
<code>trans</code>	A trans object applied to the shape's computed points. Default trans_identity() (no transform).
<code>...</code>	Arguments passed to style() .

Details

Shares its point computation and argument validation with `shape_wedge()` via two internal helpers factored into `R/shape_wedge.R` (`arc_points()`, `validate_arc_args()`), differing only in which `drawable(geometry = ...)` they construct from and the missing centroid vertex.

`style@fill` has no effect for `curve_arc()` – see [drawable](#)'s geometry documentation for why "path" geometries have no interior to fill. Passing `fill` via `...` is still accepted (it's simply ignored at draw time), since `style()` is shared across every geometry.

Recycling uses `purrr::pmap()`'s own vctrs-based rules: any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error.

Value

A [drawable](#).

For `curve_arcs()`, a [sketch](#).

See Also

Other 1D curves: [curve_bezier\(\)](#), [curve_line\(\)](#), [curve_raw\(\)](#), [curve_scribble\(\)](#), [curve_spiral\(\)](#), [curve_twist\(\)](#)

Examples

```
draw(curve_arc(start = 0, end = 3 * pi / 2))

# unlike shape_wedge(), there's no centroid vertex or fill -- just the
# arc itself
draw(curve_arc(start = pi / 4, end = pi, linewidth = 4, lineend = "round"))

# a start greater than end sweeps clockwise instead of counterclockwise
draw(curve_arc(start = pi, end = 0))

draw(curve_arcs(start = 0, end = seq(pi / 2, 2 * pi, length.out = 3)))

# concentric arcs of growing radius, all sweeping the same angle range
draw(curve_arcs(radius = seq(0.2, 1, length.out = 5), start = 0, end = pi))
```

curve_bezier

An open Bezier curve

Description

`curve_bezier` is a [drawable](#) whose path follows a Bezier curve defined by an arbitrary number of control points (x, y) , using the same Bernstein-polynomial machinery as [shape_bezier\(\)](#). Where `shape_bezier()` always closes back to its first control point (a consequence of every "polygon"-geometry `drawable` being rendered as a closed `grid::polygonGrob()`), `curve_bezier()` sets `geometry = "path"` and is rendered as an open `grid::polylineGrob()` instead, stopping at its last control point rather than looping back.

`curve_beziers()` is a vectorized version of [curve_bezier\(\)](#). Since x/y are themselves numeric vectors of control points for a single curve, `curve_beziers()` takes them as a `list()` of numeric vectors instead – one vector of control points per curve.

Usage

```
curve_bezier(x, y, n = 100L, trans = trans_identity(), ...)
```

```
curve_beziers(x, y, n = 100L, trans = trans_identity(), ...)
```

Arguments

<code>x, y</code>	For <code>curve_bezier()</code> , numeric vectors of control point coordinates, the same length, with at least two control points. For <code>curve_beziers()</code> , a <code>list()</code> of such vectors instead – one vector of control points per curve.
<code>n</code>	Number of points used to sample the curve. Default 100L.
<code>trans</code>	A trans object applied to the curve's computed points. Default trans_identity() (no transform).
<code>...</code>	Arguments passed to style() .

Details

`style@fill` has no effect for `curve_bezier()` – see [drawable](#)'s geometry documentation for why "path" geometries have no interior to fill. Passing `fill` via `...` is still accepted (it's simply ignored at draw time), since `style()` is shared across every geometry.

Every other argument may be a plain vector, recycled against `x/y` via `purrr::pmap()`'s own vctrs-based rules (any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error). The result is a [sketch](#) containing one `curve_bezier()` per list element/recycled row, rather than a single `drawable`.

Value

A [drawable](#).

For `curve_beziers()`, a [sketch](#).

See Also

Other 1D curves: [curve_arc\(\)](#), [curve_line\(\)](#), [curve_raw\(\)](#), [curve_scribble\(\)](#), [curve_spiral\(\)](#), [curve_twist\(\)](#)

Examples

```
draw(curve_bezier(x = c(0, 0.5, 1), y = c(0, 1, 0)))

# unlike shape_bezier(), the curve stops at its last control point
# rather than closing back to the first
draw(curve_bezier(
  x = c(0, 0.25, 0.75, 1), y = c(0, 1, -1, 0), linewidth = 3
))

# a ribbon can be built around any curve_*( ) path with shape_strokepath()
draw(shape_strokepath(
  curve_bezier(x = c(0, 0.5, 1), y = c(0, 1, 0)),
  width = 0.15
))

draw(curve_beziers(
  x = list(c(0, 0.5, 1), c(2, 2.5, 3)),
  y = list(c(0, 1, 0), c(0, 1, 0))
))
```

curve_line	<i>An open polyline</i>
------------	-------------------------

Description

curve_line is a [drawable](#) whose path connects an arbitrary number of control points (x, y) with straight segments, in order. With two control points this is a single line segment; with more, an open polyline.

curve_lines() is a vectorized version of [curve_line\(\)](#). Since x/y are themselves numeric vectors of control points for a single polyline, curve_lines() takes them as a `list()` of numeric vectors instead – one vector of control points per polyline.

Usage

```
curve_line(x, y, trans = trans_identity(), ...)
```

```
curve_lines(x, y, trans = trans_identity(), ...)
```

Arguments

x, y	For curve_line(), numeric vectors of control point coordinates, the same length, with at least two control points. For curve_lines(), a <code>list()</code> of such vectors instead – one vector of control points per polyline.
trans	A trans object applied to the curve's computed points. Default trans_identity() (no transform).
...	Arguments passed to style() .

Details

Unlike [curve_bezier\(\)/shape_bezier\(\)](#), the control points are not smoothed or resampled – points is exactly (x, y), so there is no `n` argument.

`style@fill` has no effect for curve_line() – see [drawable](#)'s geometry documentation for why "path" geometries have no interior to fill. Passing fill via ... is still accepted (it's simply ignored at draw time), since `style()` is shared across every geometry.

Every other argument may be a plain vector, recycled against x/y via `purrr::pmap()`'s own vctrs-based rules (any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error). The result is a [sketch](#) containing one curve_line() per list element/recycled row, rather than a single drawable.

Value

A [drawable](#).

For curve_lines(), a [sketch](#).

See Also

Other 1D curves: [curve_arc\(\)](#), [curve_bezier\(\)](#), [curve_raw\(\)](#), [curve_scribble\(\)](#), [curve_spiral\(\)](#), [curve_twist\(\)](#)

Examples

```
draw(curve_line(x = c(0, 1, 1, 2), y = c(0, 1, 0, 1)))

# a thick, square-capped line reads very differently from a thin,
# round-capped one
draw(curve_line(
  x = c(0, 1, 2), y = c(0, 1, 0), linewidth = 12, lineend = "square"
))

# layer several jittered copies for a hand-drawn look
draw(effect_tremor(
  curve_line(x = c(0, 1, 1, 2), y = c(0, 1, 0, 1)),
  layers = 5L
))

draw(curve_lines(
  x = list(c(0, 1, 1, 2), c(2, 3, 3, 4)),
  y = list(c(0, 1, 0, 1), c(0, 1, 0, 1))
))

# a simple hatched grid of crossing lines
draw(curve_lines(
  x = list(c(0, 5), c(0, 5), c(0, 5)),
  y = list(c(0, 5), c(1, 4), c(2, 3))
))
```

curve_raw

An open path defined directly by its vertices

Description

curve_raw is [shape_raw](#)'s "path"-geometry analog: the user supplies x/y coordinates directly, connected by straight segments in the order given, with no smoothing, resampling, or implicit closing edge. Unlike [curve_line\(\)](#) (which requires at least two control points, since a single-point "line" isn't meaningful), curve_raw places no minimum on length(x), matching [shape_raw](#)'s own leniency – it exists primarily as a [convert\(\)](#) target for "freezing" any "path"-geometry drawable's computed points, the same role [shape_raw](#) plays for "polygon"-geometry drawables.

curve_raws() is a vectorized version of [curve_raw\(\)](#). Since x/y are themselves numeric vectors of vertex coordinates for a single path, curve_raws() takes them as a [list\(\)](#) of numeric vectors instead – one vector of vertices per path.

Usage

```
curve_raw(x, y, id = NULL, trans = trans_identity(), ...)
```

```
curve_raws(x, y, id = NULL, trans = trans_identity(), ...)
```

Arguments

<code>x, y</code>	For <code>curve_raw()</code> , a numeric vector of x/y coordinates. For <code>curve_raws()</code> , a <code>list()</code> of such vectors instead – one vector of vertices per path.
<code>id</code>	For <code>curve_raw()</code> , an integer vector the same length as <code>x/y</code> , or <code>NULL</code> (the default, a single sub-path). For <code>curve_raws()</code> , a <code>list()</code> of such vectors (or <code>NULL</code> s) instead, one per path.
<code>trans</code>	A trans object applied to the curve's computed points. Default <code>trans_identity()</code> (no transform).
<code>...</code>	Arguments passed to style() .

Details

`style@fill` has no effect for `curve_raw()` – see [drawable](#)'s geometry documentation for why "path" geometries have no interior to fill. Passing `fill` via `...` is still accepted (it's simply ignored at draw time), since `style()` is shared across every geometry.

`id` optionally groups `x/y` into multiple sub-paths (see [xy](#)'s own `id`), letting a single `curve_raw()` render as several disjoint strokes sharing one [style](#).

Every other argument may be a plain vector, recycled against `x/y` via `purrr::pmap()`'s own `vctrs`-based rules (any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error). The result is a [sketch](#) containing one `curve_raw()` per list element/recycled row, rather than a single `drawable`.

Value

A [drawable](#).

For `curve_raws()`, a [sketch](#).

See Also

Other 1D curves: [curve_arc\(\)](#), [curve_bezier\(\)](#), [curve_line\(\)](#), [curve_scribble\(\)](#), [curve_spiral\(\)](#), [curve_twist\(\)](#)

Examples

```
draw(curve_raw(x = c(0, 1, 2), y = c(0, 1, 0)))

# useful for "freezing" a wandering path's own computed points
frozen <- S7::convert(
  curve_twist(
    x = 0, y = 0, xend = 1, yend = 0,
    path_distortion = noise_bridge(seed = 99)
  ),
```

```

    curve_raw
  )
  draw(frozen)

  # id groups x/y into several disjoint strokes sharing one style
  draw(curve_raw(
    x = c(0, 1, 2, 3), y = c(0, 1, 0, 1),
    id = c(1, 1, 2, 2)
  ))

  draw(curve_raws(
    x = list(c(0, 1, 2), c(2, 3, 4)),
    y = list(c(0, 1, 0), c(0, 1, 0))
  ))

```

curve_scribble	<i>A wandering scribble curve</i>
----------------	-----------------------------------

Description

curve_scribble is a [drawable](#) whose path is a single random wandering line – a random finite sum of sine harmonics – built from the same internal scribble_lines() generator that fill_scribble() uses to texture a tile’s interior, but scaled here into an arbitrary bounding box on the sketch’s own coordinate plane rather than tiled inside a fill pattern. Where fill_scribble() scatters several such lines across a repeating tile as a texture, curve_scribble() draws exactly one as a standalone open curve.

curve_scribbles() is a vectorized version of curve_scribble(): each argument may be a vector, recycled against the others via purrr::pmap()’s own vctrs-based rules (any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error).

Usage

```

curve_scribble(
  x = 0,
  y = 0,
  width = 1,
  height = 1,
  direction = "horizontal",
  n_harmonics = 3L,
  amplitude = 0.35,
  n = 200L,
  seed = 1L,
  trans = trans_identity(),
  ...
)

curve_scribbles(

```

```

    x = 0,
    y = 0,
    width = 1,
    height = 1,
    direction = "horizontal",
    n_harmonics = 3L,
    amplitude = 0.35,
    n = 200L,
    seed = 1L,
    trans = trans_identity(),
    ...
)

```

Arguments

x, y	Coordinates of the curve's origin. Default 0.
width, height	Extent of the curve's bounding box along/across its direction of travel. Must be positive. Default 1.
direction	Either "horizontal" (the curve runs left-right) or "vertical" (top-bottom). Default "horizontal".
n_harmonics	Number of sine harmonics summed to build the curve. Must be a positive integer. Default 3L.
amplitude	Maximum total wiggle amplitude, as a fraction of height (for direction = "horizontal") or width (for "vertical"), split across n_harmonics. Must be a non-negative number. Default 0.35.
n	Number of points sampled along the curve. Must be a positive integer of at least 2L. Default 200L.
seed	Integer seed for the random harmonics. Default 1L.
trans	A trans object applied to the curve's computed points. Default trans_identity() (no transform).
...	Arguments passed to style() .

Details

The underlying line is generated in (along, across) form, along running from 0 to 1 and across wandering around a random baseline near 0.5 (see [scribble_lines\(\)](#)'s own details for why this particular construction was chosen – periodicity, needed for tiling, is irrelevant here). x/y place the curve's origin, and width/ height scale it: for direction = "horizontal" (the default), along maps to x + along * width and across to y + across * height, so the curve runs left-to-right; for direction = "vertical", the mapping swaps (along maps to y, across to x), so the curve runs bottom-to-top instead.

`style@fill` has no effect for `curve_scribble()` – see [drawable](#)'s geometry documentation for why "path" geometries have no interior to fill. Passing fill via ... is still accepted (it's simply ignored at draw time), since `style()` is shared across every geometry.

Varying seed per curve is usually what makes several scribbles look distinct from each other. The result is a [sketch](#) containing one `curve_scribble()` per recycled row, rather than a single `drawable`.

Value

A [drawable](#).

For `curve_scribbles()`, a [sketch](#).

See Also

Other 1D curves: [curve_arc\(\)](#), [curve_bezier\(\)](#), [curve_line\(\)](#), [curve_raw\(\)](#), [curve_spiral\(\)](#), [curve_twist\(\)](#)

Examples

```
draw(curve_scribble(width = 2, height = 0.5, seed = 5591))

# more harmonics and higher amplitude give a more agitated wiggle
draw(curve_scribble(
  width = 2, height = 0.5, n_harmonics = 8, amplitude = 0.6, seed = 5591
))

# direction = "vertical" runs the curve bottom-to-top instead
draw(curve_scribble(
  width = 0.5, height = 2, direction = "vertical", seed = 5591
))

draw(curve_scribbles(x = 1:3, width = 0.8, height = 0.5, seed = 1:3))

# stacked scribbles, like several strands of hatching
draw(curve_scribbles(y = 1:6, width = 3, height = 0.6, seed = 1:6))
```

curve_spiral

An open spiral

Description

`curve_spiral` is a [drawable](#) whose path winds around a centroid (`x`, `y`), sweeping through turns full revolutions while its radius interpolates linearly from `radius_start` to `radius_end`. With `radius_start = radius_end` this traces a circle repeated turns times (visually indistinguishable from a single circle, since the path retraces itself); the usual case has `radius_start != radius_end`, giving an Archimedean-style spiral that grows or shrinks outward.

`curve_spirals()` is a vectorized version of [curve_spiral\(\)](#): each argument may be a vector, recycled against the others via `purrr::pmap()`'s own vctrs-based rules (any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error). The result is a [sketch](#) containing one `curve_spiral()` per recycled row, rather than a single `drawable`.

Usage

```

curve_spiral(
  x = 0,
  y = 0,
  radius_start = 0,
  radius_end = 1,
  turns = 3,
  n = 200L,
  trans = trans_identity(),
  ...
)

curve_spirals(
  x = 0,
  y = 0,
  radius_start = 0,
  radius_end = 1,
  turns = 3,
  n = 200L,
  trans = trans_identity(),
  ...
)

```

Arguments

<code>x, y</code>	Centroid coordinates. Default 0.
<code>radius_start, radius_end</code>	Radius at the start/end of the path. Must be non-negative. Default 0/1.
<code>turns</code>	Number of full revolutions. Must be positive. Default 3.
<code>n</code>	Number of points used to approximate the spiral. Default 200L.
<code>trans</code>	A trans object applied to the curve's computed points. Default <code>trans_identity()</code> (no transform).
<code>...</code>	Arguments passed to style() .

Details

`style@fill` has no effect for `curve_spiral()` – see [drawable](#)'s geometry documentation for why "path" geometries have no interior to fill. Passing `fill` via `...` is still accepted (it's simply ignored at draw time), since `style()` is shared across every geometry.

Value

A [drawable](#).

For `curve_spirals()`, a [sketch](#).

See Also

Other 1D curves: [curve_arc\(\)](#), [curve_bezier\(\)](#), [curve_line\(\)](#), [curve_raw\(\)](#), [curve_scribble\(\)](#), [curve_twist\(\)](#)

Examples

```
draw(curve_spiral(radius_start = 0, radius_end = 1, turns = 4))

# radius_start > radius_end spirals inward instead of outward
draw(curve_spiral(
  radius_start = 1, radius_end = 0.1, turns = 5, linewidth = 2
))

# equal start/end radii retrace a circle -- rarely useful on its own,
# but shows turns has no effect on shape when radius doesn't change
draw(curve_spiral(radius_start = 1, radius_end = 1, turns = 3))

draw(curve_spirals(x = c(0, 3, 6), turns = c(2, 3, 4)))

# nested spirals sharing a centroid, growing radius each time
draw(curve_spirals(
  radius_start = seq(0.1, 0.5, length.out = 4),
  radius_end = seq(0.6, 1, length.out = 4)
))
```

curve_twist

An open, wandering path following a random walk

Description

curve_twist is [shape_twist\(\)](#)'s path alone, with no ribbon width: an open [curve_line\(\)](#)-like polyline between (x, y) and (xend, yend), displaced away from a straight line by a [noise_bridge](#), giving a wandering, twisted appearance.

curve_twists() is a vectorized version of [curve_twist\(\)](#): each argument may be a vector, recycled against the others via purrr::pmap()'s own vctrs-based rules. The result is a [sketch](#) containing one curve_twist() per recycled row, rather than a single drawable.

Usage

```
curve_twist(
  x = 0,
  y = 0,
  xend = 1,
  yend = 1,
  scale = 0.2,
  n = 100L,
  path_distortion = noise_bridge(),
```

```

    trans = trans_identity(),
    ...
)

curve_twists(
  x = 0,
  y = 0,
  xend = 1,
  yend = 1,
  scale = 0.2,
  n = 100L,
  path_distortion = noise_bridge(),
  trans = trans_identity(),
  ...
)

```

Arguments

<code>x, y</code>	Start point. Default 0.
<code>xend, yend</code>	End point. Default 1.
<code>scale</code>	Amplitude of the Brownian-bridge displacement (internally scaled by 0.1, matching shape_twist() 's own scaling of its path). Must be non-negative. Default 0.2.
<code>n</code>	Number of points used along the path. Default 100L.
<code>path_distortion</code>	A noise_bridge controlling the path's Brownian bridge. Default noise_bridge() .
<code>trans</code>	A trans object applied to the curve's computed points. Default trans_identity() (no transform).
<code>...</code>	Arguments passed to style() .

Details

Where [shape_twist\(\)](#) also modulates a filled ribbon's width along this same kind of path (via a separate distortion [noise_field](#)), [curve_twist\(\)](#) has no width or fill at all – just the displaced backbone itself, rendered as an open `grid::polylineGrob()` (`geometry = "path"`). Shares its path computation with [shape_twist\(\)](#) via the internal `twisted_path_points()` helper (`R/shape_twist.R`) rather than duplicating it.

`style@fill` has no effect for [curve_twist\(\)](#) – see [drawable](#)'s geometry documentation for why "path" geometries have no interior to fill. Passing `fill` via `...` is still accepted (it's simply ignored at draw time), since `style()` is shared across every geometry.

Any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error. A shared path_distortion [noise_bridge](#) is automatically recycled across every path; pass a `list()` of several different [noise_bridges](#) instead to vary it per path.

Value

A [drawable](#).

For [curve_twists\(\)](#), a [sketch](#).

See Also

Other 1D curves: [curve_arc\(\)](#), [curve_bezier\(\)](#), [curve_line\(\)](#), [curve_raw\(\)](#), [curve_scribble\(\)](#), [curve_spiral\(\)](#)

Examples

```
draw(curve_twist(
  x = 0, y = 0, xend = 1, yend = 0,
  path_distortion = noise_bridge(seed = 7734)
))

# a larger scale wanders further from the straight line between the
# endpoints
draw(curve_twist(
  x = 0, y = 0, xend = 1, yend = 0, scale = 0.6,
  path_distortion = noise_bridge(seed = 7734)
))

# curve_twist() is shape_twist()'s backbone alone, with no ribbon width
draw(shape_twist(
  x = 0, y = 0, xend = 1, yend = 0,
  path_distortion = noise_bridge(seed = 7734)
))

draw(curve_twists(x = 1:3, y = 0, xend = 2:4, yend = 1))

# a bundle of independently-wandering strands between the same endpoints
draw(curve_twists(
  x = 0, y = 0, xend = 3, yend = 0,
  path_distortion = list(
    noise_bridge(seed = 1),
    noise_bridge(seed = 2),
    noise_bridge(seed = 3)
  )
))
```

draw

Draw a drawable or sketch

Description

`draw()` is a generic function that renders a [drawable](#) or [sketch](#) object to the current graphics device using **grid**. Methods accept optional `xlim/ylim` arguments giving axis limits; if omitted, limits are computed from the object's points.

Usage

```
draw(object, ...)
```

Arguments

object	A drawable or sketch object.
...	Passed to methods, e.g. <code>xlim/ylim</code> .

See Also

Other core structure: [canvas\(\)](#), [drawable\(\)](#), [group\(\)](#), [sketch\(\)](#), [style\(\)](#), [xy\(\)](#)

Examples

```
draw(shape_circle(radius = 1))

s <- sketch() + shape_circle(radius = 1) + shape_blob(x = 2, radius = 0.5)
draw(s)

# an explicit xlim/ylim overrides both the sketch's own canvas and the
# range of its shapes' own points, useful for zooming in/out or padding
draw(shape_circle(radius = 1), xlim = c(-2, 2), ylim = c(-2, 2))

# a non-drawable object is ignored, with a warning, rather than erroring
draw("not a drawable")
```

drawable

Parent class for all drawable objects

Description

`drawable` enforces structure on its subclasses: every `drawable` must carry a [style](#), a geometry, and expose a computed `points` property, of class [xy](#). It is not intended to be instantiated directly; use one of its subclasses ([shape_raw](#), [shape_circle](#), [shape_blob](#), [shape_ribbon](#), [shape_twist](#), [curve_raw](#), [points_raw](#), ...) instead.

Usage

```
drawable(..., geometry = "polygon", trans = trans_identity(), pathlike = FALSE)
```

Arguments

...	Arguments passed to style() .
geometry	One of "polygon" (default), "path", or "points". Not exposed as a constructor argument by any concrete <code>drawable</code> – each <code>shape_*()</code> / <code>curve_*()</code> / <code>points_raw()</code> constructor fixes one value internally instead (see details).
trans	A trans/trans_warp/trans_fn/trans_chain object. See details.
pathlike	A single TRUE/FALSE (default FALSE). Not exposed as a constructor argument by any concrete <code>drawable</code> – each fixes its own value internally, the same convention geometry follows. See details.

Details

geometry tells `draw()` which `grid` grob a drawable's points map to, following a dimensional reading: "points" (0D, `grid::pointsGrob()`, e.g. `points_raw()`), "path" (1D, an open `grid::polylineGrob()`, e.g. `curve_line()/curve_raw()`), or "polygon" (2D, a closed `grid::polygonGrob()` – the default, and the only value any `shape_*()` constructor uses). `style@fill` is ignored for "points"/"path" geometries, since only a closed polygon has an interior to fill.

`trans` is a `trans` (an affine map: `trans_translate()`, `trans_rotate()`, `trans_scale()`, `trans_reflect()`, `trans_shear()`, `trans_affine()`), a `trans_warp` (a non-rigid, noise-based deformation), a `trans_fn` (a non-rigid deformation from an arbitrary caller-supplied displacement function), or a `trans_chain` combining several via `+`, applied to a drawable's computed points as the very last step – after any shape-specific geometry (and, for `shape_blob()/shape_ribbon()/shape_twist()`, any noise-based distortion) has already been computed. This means a drawable's own defining parameters (e.g. `shape_circle()`'s centroid/radius) are never mutated or flattened by a transform – only the final rendered coordinates are affected. Default `trans_identity()` (no transform).

`pathlike` marks whether `x/y` (where present) hold a genuine, caller-ordered, perturbable control-point path – as opposed to `x/y` meaning something else entirely (e.g. `shape_circle()`'s centroid, or one fixed endpoint of `shape_ribbon()`'s two-point segment). This is the distinction `effect_tremor()/effect_bristle()` need to decide whether jittering `x/y` produces a meaningful wobble; it's orthogonal to geometry – a pathlike drawable can have any geometry (a future `points_*()` constructor could reasonably be pathlike despite `geometry == "points"`). Currently TRUE for `shape_raw()`, `curve_raw()`, `curve_line()`, `shape_stroke()`, `shape_bezier()`, `curve_bezier()`, and `points_raw()`; FALSE (the default) for every other concrete drawable, including `shape_ribbon()/shape_twist()/curve_twist()` – these do have a conceptual backbone, but it's exposed via `x/y/xend/yend` (or additional named control-point pairs), not a plain `x/y` vector. Whether a pathlike subclass actually has `x/y` properties is not enforced by drawable's own validator – every subclass constructor first builds a scaffold `drawable()` instance (validated on its own, before any subclass property exists) and only merges in `x/y` afterward via `S7::new_object()`, so a cross-property check here would fire on that scaffold and reject every pathlike subclass unconditionally (see "Gotchas"). Setting `pathlike = TRUE` on a subclass with no `x/y` is therefore an author error caught only when an effect tries to read `object@x/object@y`, not at construction time.

See Also

Other core structure: `canvas()`, `draw()`, `group()`, `sketch()`, `style()`, `xy()`

Examples

```
circ <- shape_circle(radius = 1)
S7::S7_inherits(circ, drawable)

# geometry controls which grob draw() builds: a closed outline, an open
# stroke, or unconnected markers
draw(shape_circle(radius = 1, n = 8))
draw(curve_line(x = c(0, 1, 2, 3), y = c(0, 1, 0, 1), linewidth = 3))
draw(points_raw(x = runif(30), y = runif(30)))

# trans applies after a shape's own geometry/distortion is computed, so
# shape_rectangle()'s centroid/width/height stay fixed -- only the final
# rendered corners rotate
```

```

draw(shape_rectangle(
  width = 1.5,
  height = 0.5,
  trans = trans_rotate(pi / 6)
))

```

effect_bristle

A fanned bristle/dry-brush effect along a path

Description

effect_bristle() builds a [group](#) of n_bristles copies of a template drawable, fanned out perpendicular to its own backbone path.

Usage

```

effect_bristle(
  object,
  n_bristles = 9L,
  spread = 0.3,
  width_jitter = 0.3,
  fray = 0.15,
  jitter = 0.015,
  jitter_frequency = 1.2,
  n = 100L,
  seed = 1L
)

```

Arguments

object	A template pathlike drawable (@pathlike == TRUE, see drawable 's docs) with a width property – typically shape_stroke() . Every bristle is a copy of object with its own fanned x/y, scaled width, and (if object has an n property) resampled n; all other properties (style, distortion, trans, ...) carry over unchanged.
n_bristles	Number of bristles to fan out. Must be a positive integer. Default 9L.
spread	Total perpendicular distance the bristles fan across, centred on the backbone. Must be non-negative. Default 0.3.
width_jitter	Fractional random variation applied to each bristle's own object@width (e.g. 0.3 scales width by a factor drawn uniformly from [0.7, 1.3]). Must be in [0, 1). Default 0.3.
fray	Maximum fraction of the backbone's own length randomly trimmed from each bristle's start and end. Must be in [0, 0.5). Default 0.15.
jitter, jitter_frequency	Passed to effect_tremor() 's own arguments of the same name, controlling each bristle's independent wobble. Defaults 0.015/1.2.

n	Number of points used along the backbone when fanning bristles out (independent of any n property object itself has). Must be at least 2. Default 100L.
seed	Integer seed for the per-bristle randomization and wobble. Default 1L.

Details

Like `effect_tremor()`, no single drawable can express this by itself, since a dry-brush look comes from several independently-frayed, independently wobbling strands laid side by side, not one stroke. Each bristle is a copy of object (via `S7::set_props()`) that:

- is offset from the backbone by a fixed perpendicular distance (via the same per-point unit normal `shape_stroke()` itself uses, `stroke_normals()`), evenly spaced across spread;
- is trimmed to a random sub-range of the backbone's own length (fray controls how much), so bristles start/end raggedly rather than in one clean line – the same "frayed edge" a real brush's outer bristles show as it runs dry;
- has its own randomly-scaled width (via `width_jitter`, applied to `object@width`); and
- is independently wobbled via `effect_tremor()` (layers = 1L, since the fanning here already does the layering work `effect_tremor()` normally provides – one wobbling copy per bristle position, not several wobbling copies of one path).

Every other property of object – style, distortion, trans, ... – carries over to every bristle unchanged, since each is built with `S7::set_props()` rather than a fresh constructor call. `shape_stroke()`'s own taper-to-zero at both ends does double duty as each bristle's tip fade, needing no extra work here. Randomization (fray range, width scaling) is scoped per bristle with `withr::with_seed()`, the same reproducibility convention `fill_stipple()/fill_scatter()/fill_halftone()` already use, so it never leaks into the caller's global random state.

Value

A `group` containing `n_bristles` drawables.

See Also

Other effects: `effect_grain()`, `effect_swatch()`, `effect_tremor()`

Examples

```
t <- seq(0, 8, length.out = 200)
template <- shape_stroke(
  x = t, y = sin(t) * 1.2, width = 0.06,
  fill = "black", fill_alpha = 0.4, color = NA_character_
)

# before: a single plain stroke
draw(template)

# after: fanned into a dry-brush bristle effect
draw(effect_bristle(template, n_bristles = 11L, spread = 0.3))

# a wider spread and more fraying gives a rougher, more ragged brush
```

```
draw(effect_bristle(template, n_bristles = 15L, spread = 0.5, fray = 0.3))
```

effect_grain

A paper-grain/textured-ink rendering of a drawable's own outline

Description

effect_grain takes an existing polygon-geometry [drawable](#) (object, e.g. [shape_stroke\(\)](#), [shape_blob\(\)](#), [shape_ribbon\(\)](#)) and renders its own outline (object@points) not with a [style\(\)](#)@fill via the ordinary draw()/geometry_grob() path, but by compositing a rasterised paper-grain texture and masking it to the outline's exact polygon shape via [grid::as.mask\(\)](#).

Usage

```
effect_grain(
  object,
  grain = noise_field(frequency = 15, octaves = 2L, seed = 2L),
  resolution = 150L,
  color = "gray15",
  alpha = 1,
  background = NA_character_
)
```

Arguments

object	A polygon-geometry drawable (@geometry == "polygon") whose outline is textured – e.g. shape_stroke() , shape_blob() , shape_ribbon() . object@points is used directly; object@style plays no role (effect_grain() draws its own grain raster instead).
grain	A noise_field controlling the paper-grain texture, sampled directly at each raster pixel's own world position (not torus-periodic, unlike fill_noise() 's tiled field – see Details). Default noise_field(frequency = 15, octaves = 2L, seed = 2L) (finer/denser than noise_field()'s own default, for a paper-grain rather than cloudy look).
resolution	Raster resolution (pixels per edge of object's own bounding box). Must be a positive integer of at least 2L. Default 150L.
color	Grain colour. Default "gray15".
alpha	Maximum grain opacity, at the noise field's peak. Must be a number in (0, 1]. Default 1.
background	Colour revealed as grain fades out, or NA for true transparency (showing whatever is drawn behind). Default NA.

Details

This is the same masking technique `fill_vignette()` already uses for its own radial fade, but applied to object's own real (possibly concave, tapering-to-a-point) silhouette rather than a synthetic circle drawn purely to build the mask.

This is a different effect than filling object with `fill_charcoal()/fill_noise()` (already a good option for a shape's interior – see `fill_charcoal()`'s docs): those tile a periodic texture that repeats across the shape via `grid::pattern()`, sized relative to the target's own bounding box. `effect_grain()` instead samples its grain `noise_field` once, directly, across object's own world coordinates – a single non-repeating raster the size of the whole shape, with no tiling seam to manage, at the cost of needing its own `draw()` method rather than reusing `geometry_grob()`'s existing "polygon"/"path"/"points" branches (`effect_grain` is not a `drawable` subclass at all, since its rendering isn't expressible as a single points-based grob).

Grain is rendered as varying opacity of color, from fully transparent at the noise field's minimum to alpha at its maximum – exactly `fill_noise()`'s own opacity convention – optionally revealing a solid background colour underneath (as `fill_vignette()`'s own background argument does) rather than true transparency, which reads as a solid, mottled ink stroke instead of a sparse one.

Value

An `effect_grain` object.

See Also

Other effects: `effect_bristle()`, `effect_swatch()`, `effect_tremor()`

Examples

```
t <- seq(0, 8, length.out = 150)
template <- shape_stroke(x = t, y = sin(t), width = 0.4)

# before: a plain filled stroke
draw(template)

# after: the same outline rendered as textured grain instead of a
# solid style() fill
draw(effect_grain(template, color = "gray10", alpha = 0.9))

# a non-NA background reveals a solid colour underneath the grain,
# instead of true transparency
draw(effect_grain(
  template,
  color = "gray5",
  alpha = 0.85,
  background = "gray30"
))

# a coarser grain field (lower frequency) reads less like paper texture
# and more like a mottled brushstroke
draw(effect_grain(template, grain = noise_field(frequency = 3, seed = 2)))
```

effect_swatch	<i>A colour-swatch effect: repeat a template drawable, one per colour</i>
---------------	---

Description

effect_swatch() builds a [group](#) of length(color) copies of a template drawable, laid out left to right and each independently filled from color – a colour-preview effect for any palette_*() function’s own output (or any other character vector of colours).

Usage

```
effect_swatch(object = shape_rectangle(color = NA_character_), color, gap = 0)
```

Arguments

object	A template drawable , repeated once per colour. Its own bounding-box width (the range of object@points@x) sets the spacing between copies; every property besides style@fill carries over unchanged. Default shape_rectangle() with no visible border.
color	A character vector of colours to preview, in order (e.g. the output of another palette_*() function). Since object has a default and comes first, pass color by name unless also supplying object.
gap	A single non-negative number: the horizontal spacing between adjacent copies, in the same units as object’s own bounding-box width. Default 0 (contiguous swatches, no gap).

Details

Every property of object besides style@fill – its own size, style@color/linewidth, trans, ... – carries over unchanged to every copy, via the same [S7::set_props\(\)](#) contract [effect_tremor\(\)](#)/[effect_bristle\(\)](#) already use. Unlike those two, object need not be pathlike (see [drawable](#)’s own pathlike docs) – a [shape_rectangle\(\)](#)’s own x/y is a centroid, not a perturbable control-point path – so each copy is instead repositioned by composing a [trans_translate\(\)](#) onto object’s own trans, the one mechanism every drawable class supports uniformly regardless of what its own x/y means. This also means object can be *any* drawable, not just a rectangle – swatch a palette across circles, blobs, or stars by passing a different template.

Swatches render with hard edges rather than blending into one another, since that’s a more honest representation of a palette whose colours are meant to be used as-is (e.g. [palette_manual\(\)](#)/[palette_harmony\(\)](#)) – a smooth interpolation between swatches would show shades that aren’t actually part of the palette.

Value

A [group](#) of length(color) copies of object.

See Also

Other effects: [effect_bristle\(\)](#), [effect_grain\(\)](#), [effect_tremor\(\)](#)

Examples

```
draw(effect_swatch(color = palette_manual(index = 3)))
draw(effect_swatch(color = palette_harmony("steelblue", scheme = "triadic"), gap = 0.1))

# any drawable can serve as the template, not just a rectangle
draw(effect_swatch(shape_circle(radius = 0.5), palette_tonal("darkorchid", n = 6)))
```

effect_tremor

Layer jittered copies of a drawable for a hand-drawn look

Description

`effect_tremor()` builds a [group](#) of layers such copies: each is a copy of object (via [S7::set_props\(\)](#)) with its x/y displaced by smooth, seed-offset simplex noise sampled along the path's own normalized arc-length.

Usage

```
effect_tremor(
  object,
  layers = 4L,
  jitter = 0.05,
  jitter_frequency = 0.5,
  seed = 1L
)
```

Arguments

object	A pathlike drawable (<code>@pathlike == TRUE</code> , see drawable 's docs), e.g. curve_line() , shape_stroke() , shape_bezier() . Every other property is preserved unchanged across layers.
layers	Number of independently-jittered copies to layer. Must be a positive integer. Default 4L.
jitter	Maximum displacement amplitude applied to each layer's x/y. Must be non-negative. Default 0.05.
jitter_frequency	Frequency of the simplex noise driving the jitter, sampled along the path's normalized arc-length – lower values give a slower, smoother wobble; higher values a jumpier one. Must be non-negative. Default 0.5.
seed	Integer seed for the jitter noise. Default 1L.

Details

No single [drawable](#) can express a hand-drawn ink/pencil look on its own – what reads as hand-drawn is several independently wobbling copies of the same nominal path layered on top of each other, not one perfectly smooth line. This is the same ad hoc technique used, during development, to add a wobbling pencil edge on top of a [shape_stroke\(\)](#)'s tapered outline, or to layer a plain [curve_line\(\)](#) into a shakier-looking line by itself.

Noise is sampled at each control point's own position along arc-length (0 to 1), not at its raw x/y coordinates, so the jitter's shape is independent of the path's own scale or aspect ratio – a long, shallow path and a short, steep one with the same number of control points get comparably-shaped wobble. Each layer's noise is seed-offset from the last (seed + 2 * (i - 1) for the x displacement, one more for y, following the same seed-offset convention [trans_warp\(\)](#)/[twisted_path_points\(\)](#) already use for their own independent-axis noise), so layers wobble independently rather than moving in lockstep.

Because each layer is built with [S7::set_props\(\)](#) from object itself, every other property – style, width/distortion for a [shape_stroke\(\)](#), trans, ... – carries over unchanged; [effect_tremor\(\)](#) only varies x/y across layers. Vary [color_alpha](#) down and/or layers up (on object and via the [layers](#) argument, respectively) for a denser, more overlapping pencil/ink texture.

Value

A [group](#) containing layers drawables.

See Also

Other effects: [effect_bristle\(\)](#), [effect_grain\(\)](#), [effect_swatch\(\)](#)

Examples

```
template <- curve_line(x = c(0, 1, 2, 3), y = c(0, 1, 0, 1))
faded <- curve_line(x = c(0, 1, 2, 3), y = c(0, 1, 0, 1), color_alpha = 0.4)

# before: a single crisp line
draw(template)

# after: several jittered, faded copies read as hand-drawn
draw(effect_tremor(faded))

draw(effect_tremor(
  shape_stroke(
    x = c(0, 1, 2, 3), y = c(0, 1, 0, 1),
    width = 0.3, fill_alpha = 0.5, color = NA_character_
  ),
  layers = 3L, jitter = 0.03
))

# more layers and higher jitter give a denser, shakier scribble
draw(effect_tremor(faded, layers = 10L, jitter = 0.15))
```

fill	<i>A resolved or auto-resolving fill value</i>
------	--

Description

fill is the common representation stored in `style()`'s fill property and `canvas()`'s background property: a value (a plain colour string, or the `GridPattern` output of a `fill_*()` pattern/ gradient helper) plus an optional resolve function. `resolve`, when present, is called with the real target's own bounding-box aspect ratio at `draw()` time to rebuild value with the correct tile shape – see the internal `resolvable_fill()/resolve_fill()` helpers (`R/fill.R`), and `fill_hatch()`'s own aspect argument docs for the problem this solves. `resolve` is `NULL` whenever a helper's aspect was supplied explicitly (a fixed aspect, never automatically recomputed) or for a fill with no aspect-dependence at all (`fill_solid()`, `fill_none()`).

Usage

```
fill(value = "black", resolve = NULL)
```

Arguments

value	A plain colour string, or a <code>GridPattern</code> object (the output of <code>grid::pattern()</code> , as returned by every <code>fill_*()</code> helper besides <code>fill_solid()/fill_none()</code>). Default "black".
resolve	<code>NULL</code> , or a function of one argument (aspect) rebuilding value for a newly-known target aspect ratio. Default <code>NULL</code> .

Details

Not usually constructed directly – every `fill_*()` helper already returns one, and `style()/canvas()` coerce a bare colour string or `GridPattern` into one automatically (via the internal `as_fill()` helper) if passed directly.

See Also

Other fill helpers: `fill_charcoal()`, `fill_checker()`, `fill_crosshatch()`, `fill_flow()`, `fill_gradient()`, `fill_halftone()`, `fill_hatch()`, `fill_image()`, `fill_marble()`, `fill_noise()`, `fill_none()`, `fill_scatter()`, `fill_scribble()`, `fill_solid()`, `fill_stipple()`, `fill_stripe()`, `fill_vignette()`

fill_charcoal	<i>Charcoal/marker-style noise texture fill</i>
---------------	---

Description

fill_charcoal() is a [fill_noise\(\)](#) preset – same field, same rendering, just different defaults – tuned to read as hand-drawn charcoal or marker grain rather than [fill_noise\(\)](#)'s more general-purpose rasterized field: a lighter base tone, finer/denser tiling, and finer noise detail.

Usage

```
fill_charcoal(
  color = "gray15",
  spacing = 0.25,
  aspect = NULL,
  resolution = 32L,
  alpha = 1,
  frequency = 4,
  octaves = 3L,
  seed = 1L,
  extend = "repeat"
)
```

Arguments

color	One or more fill colours, as in fill_noise() . Default "gray15" (lighter than fill_noise() 's own "black" default, closer to a charcoal tone).
spacing	Baseline tile size, as a fraction of the target's bounding box. Must be a positive number. Default 0.25 (finer than fill_noise() 's own 0.5 default, for denser grain).
aspect	Width-to-height ratio of the target polygon's bounding box. Must be a positive number, or NULL (the default) to resolve it automatically from the real target's own bounding-box aspect ratio at draw() time – see the fill class. Passing a fixed number instead computes the pattern once, immediately, against that value only.
resolution	Raster resolution (pixels per tile edge). Must be a positive integer of at least 2L. Default 32L.
alpha	Opacity. For a single color, the maximum opacity at the noise field's peak; for two or more, a flat opacity applied uniformly. Must be a number in (0, 1]. Default 1.
frequency	Noise frequency, as in shape_blob() . Must be non-negative. Default 4 (finer detail than fill_noise() 's own default of 1).
octaves	Number of noise octaves, as in shape_blob() . Must be a positive integer. Default 3L (one more layer of detail than fill_noise() 's own default of 2L).
seed	Integer seed for the noise field, as in shape_blob() . Default 1L.
extend	Passed to grid::pattern() . Default "repeat".

Details

Found, while prototyping `shape_stroke()`'s interior texture, to be a substantially better fit than `fill_scribble()` for a curved stroke's body – `fill_scribble()`'s angle is a single fixed value for the whole tile (and only tiles exactly seamlessly at a multiple of 90 degrees; see its "Known limitation" docs section), so it can't track a curved path's own continuously-varying tangent, producing hatching that visibly cuts across the stroke at odd angles wherever the path bends, while `fill_noise()`'s directionless mottling has no orientation to clash with the curve.

Value

A pattern object as returned by `grid::pattern()`, suitable for use as the fill argument to `grid::gpar()`.

See Also

Other fill helpers: `fill()`, `fill_checker()`, `fill_crosshatch()`, `fill_flow()`, `fill_gradient()`, `fill_halftone()`, `fill_hatch()`, `fill_image()`, `fill_marble()`, `fill_noise()`, `fill_none()`, `fill_scatter()`, `fill_scribble()`, `fill_solid()`, `fill_stipple()`, `fill_stripe()`, `fill_vignette()`

Examples

```
draw(shape_stroke(
  x = c(0, 1, 2, 3), y = c(0, 1, 0, 1), width = 0.4,
  fill = fill_charcoal(), color = NA_character_
))

# a lighter tone and a curved backbone -- fill_charcoal()'s directionless
# mottling tracks a bend that fill_scribble()'s fixed direction wouldn't
t <- seq(0, 2 * pi, length.out = 200)
draw(shape_stroke(
  x = t, y = sin(t), width = 0.3,
  fill = fill_charcoal(color = "gray40"), color = NA_character_
))
```

fill_checker

Checkerboard pattern fill

Description

`fill_checker()` builds a `grid::pattern()` fill value that renders a checkerboard, generalized from two colours to an arbitrary palette.

Usage

```
fill_checker(
  color = c("black", "white"),
  spacing = 0.2,
  aspect = NULL,
  extend = "repeat"
)
```

Arguments

color	Two or more checker colours. Default <code>c("black", "white")</code> .
spacing	Baseline tile size, as a fraction of the target's bounding box. Must be a positive number. Default <code>0.2</code> .
aspect	Width-to-height ratio of the target polygon's bounding box. Must be a positive number, or <code>NULL</code> (the default) to resolve it automatically from the real target's own bounding-box aspect ratio at <code>draw()</code> time – see the <code>fill</code> class. Passing a fixed number instead computes the pattern once, immediately, against that value only.
extend	Passed to <code>grid::pattern()</code> . Default <code>"repeat"</code> .

Details

It's the cheapest member of the hatch family to build: a checkerboard square has no direction the way a hatch line does (compare `fill_hatch()`'s corner-to-corner diagonal, needed specifically to tile a *sloped* line seamlessly), so the tile content here is just a grid of plain quadrant rectangles – the same two-colour-grid special case `fill_crosshatch()` already falls back to when angle is a multiple of 90 degrees, pulled out into its own helper.

The tile is subdivided into an $n \times n$ grid, where $n = \text{length}(\text{color})$. The colour at grid cell `(row, col)` (0-indexed) is `color[((row + col) %% n) + 1]` – for the default two colours this reproduces the classic 2x2 checkerboard exactly; a longer color vector grows the grid rather than adding a separate density argument, since a checkerboard's cell size and colour count aren't independent concepts here.

Value

A pattern object as returned by `grid::pattern()`, suitable for use as the fill argument to `grid::gpar()`.

Known rendering risk with three or more colours

The default two-colour, four-rectangle tile has always rendered correctly, but color vectors of length 3 or more (a 3x3 or larger grid of rectangles) were found, at the default spacing, to trigger the same upstream grid/Cairo issue documented at `fill_stipple()`'s "Known rendering risk" section – several shapes inside a genuinely *repeated* `grid::pattern()` tile can render distorted (here, collapsing to a single solid colour instead of a grid), even though the same tile content renders correctly as a single, non-repeated tile (`spacing = 1`). **Visually check rendered output** before relying on more than two colors for anything beyond casual use.

As with the other `fill_*()` helpers, `grid::pattern()` tiles are sized as a fraction of the target polygon's own bounding box rather than a fixed physical square, so the checker squares would render as rectangles, not squares, on a non-square bounding box – the same tile-squaring technique `fill_stipple()` uses for its dots. `aspect` resolves this automatically by default (see `fill_hatch()`'s own aspect docs).

See Also

Other fill helpers: `fill()`, `fill_charcoal()`, `fill_crosshatch()`, `fill_flow()`, `fill_gradient()`, `fill_halftone()`, `fill_hatch()`, `fill_image()`, `fill_marble()`, `fill_noise()`, `fill_none()`, `fill_scatter()`, `fill_scribble()`, `fill_solid()`, `fill_stipple()`, `fill_stripe()`, `fill_vignette()`

Examples

```
draw(shape_circle(fill = fill_checker(color = c("black", "white"))))

# a coarser, differently-coloured checkerboard
draw(shape_circle(
  fill = fill_checker(color = c("steelblue", "white"), spacing = 0.4)
))

# three or more colours grow the grid rather than alternating just two;
# spacing = 1 avoids the tile-repetition rendering risk noted above
draw(shape_circle(
  fill = fill_checker(color = c("steelblue", "white", "tomato"), spacing = 1)
))
```

fill_crosshatch	<i>Crosshatch pattern fill</i>
-----------------	--------------------------------

Description

fill_crosshatch() builds a `grid::pattern()` fill value that renders two mirror-symmetric hatch lines, at angle and -angle, forming an "X" inside each tile.

Usage

```
fill_crosshatch(
  color = "black",
  angle = 45,
  spacing = 0.1,
  aspect = NULL,
  linewidth = 1,
  extend = "repeat"
)
```

Arguments

color	One or more line colours. A vector of length 2 colours the two lines independently; a single colour (the default) colours both the same, matching the original behaviour. Longer vectors are recycled to length 2. Default "black".
angle	Hatch angle in degrees, measured counterclockwise from the positive x-axis. Default 45.
spacing	Baseline tile size, as a fraction of the target's bounding box. Must be a positive number. Default 0.1.
aspect	Width-to-height ratio of the target polygon's bounding box. Must be a positive number, or NULL (the default) to resolve it automatically from the real target's own bounding-box aspect ratio at <code>draw()</code> time – see the <code>fill</code> class. Passing a fixed number instead computes the pattern once, immediately, against that value only.

linewidth	Line width. Must be a positive number. Default 1.
extend	Passed to <code>grid::pattern()</code> . Default "repeat".

Details

It shares `fill_hatch()`'s tile-shape technique: both lines are drawn as the two corner-to-corner diagonals of a single rectangular tile (rather than at an arbitrary baked-in slope), so both tile seamlessly under `extend = "repeat"`, and the tile's width/height ratio – not the diagonals' own coordinates – determines the rendered angle. See `fill_hatch()`'s details for why this matters.

Because both lines share one tile shape, they are only *perpendicular* when `angle = 45` (the classic crosshatch look); for other angles the two lines are symmetric about the horizontal axis but not at right angles to each other. Genuinely perpendicular hatching at an arbitrary angle would need two differently-shaped tiles layered as separate fills, which this function does not attempt.

At angle a multiple of 90 degrees, the two mirrored diagonals would coincide, so this case is handled separately by drawing a horizontal line and a vertical line instead (a simple grid).

Value

A pattern object as returned by `grid::pattern()`, suitable for use as the `fill` argument to `grid::gpar()`.

See Also

Other fill helpers: `fill()`, `fill_charcoal()`, `fill_checker()`, `fill_flow()`, `fill_gradient()`, `fill_halftone()`, `fill_hatch()`, `fill_image()`, `fill_marble()`, `fill_noise()`, `fill_none()`, `fill_scatter()`, `fill_scribble()`, `fill_solid()`, `fill_stipple()`, `fill_stripe()`, `fill_vignette()`

Examples

```
draw(shape_circle(fill = fill_crosshatch(angle = 30)))

# at a multiple of 90 degrees, the mirrored diagonals would coincide,
# so a plain horizontal/vertical grid is drawn instead
draw(shape_circle(fill = fill_crosshatch(angle = 0, spacing = 0.15)))

# only angle = 45 gives genuinely perpendicular lines; other angles are
# symmetric about the horizontal axis but not at right angles
draw(shape_circle(fill = fill_crosshatch(angle = 20, spacing = 0.12)))

# a two-colour vector colours the two lines independently
draw(shape_circle(
  fill = fill_crosshatch(color = c("steelblue", "tomato"), angle = 45)
))
```

fill_flow

*Domain-warped noise texture fill***Description**

fill_flow() is a swirlier variant of [fill_noise\(\)](#): before sampling the final noise field, the tile's own (theta_u, theta_v) angles (see [fill_noise\(\)](#) details for why angles, not raw (u, v)) are displaced by a second, independent torus-periodic noise field – the "fBm of fBm" domain-warping recipe popularized for flowing, curl-noise-like textures, adapted here so the warp field is itself torus-periodic rather than sampled directly, keeping the whole result seamless.

Usage

```
fill_flow(
    color = "black",
    spacing = 0.5,
    aspect = NULL,
    resolution = 32L,
    alpha = 1,
    warp = 2,
    warp_frequency = 1,
    warp_octaves = 1L,
    frequency = 1,
    octaves = 2L,
    seed = 1L,
    extend = "repeat"
)
```

Arguments

color	One or more fill colours. A single colour (the default) fades in opacity from fully transparent to alpha, exactly as before this argument accepted a vector. Two or more colours instead blend across a <code>grDevices::colorRamp()</code> built from them, driven by the noise value, with alpha applied as a flat opacity across the whole fill (see the internal <code>noise_to_pixels()</code> helper). Default "black".
spacing	Baseline tile size, as a fraction of the target's bounding box. Must be a positive number. Default 0.5.
aspect	Width-to-height ratio of the target polygon's bounding box. Must be a positive number, or NULL (the default) to resolve it automatically from the real target's own bounding-box aspect ratio at draw() time – see the fill class. Passing a fixed number instead computes the pattern once, immediately, against that value only.
resolution	Raster resolution (pixels per tile edge). Must be a positive integer of at least 2L. Default 32L.

alpha	Opacity. For a single color, the maximum opacity at the noise field's peak; for two or more, a flat opacity applied uniformly. Must be a number in $(0, 1]$. Default 1.
warp	Warp amplitude, in radians of angular displacement. Must be a non-negative number. Default 2.
warp_frequency	Frequency of the two warp-displacement noise fields. Must be non-negative. Default 1.
warp_octaves	Number of octaves for the warp-displacement noise fields. Must be a positive integer. Default 1L.
frequency	Noise frequency, as in shape_blob() . Must be non-negative. Default 1.
octaves	Number of noise octaves, as in shape_blob() . Must be a positive integer. Default 2L.
seed	Integer seed for the noise field, as in shape_blob() . Default 1L.
extend	Passed to grid::pattern() . Default "repeat".

Details

The warp field needs to be decorrelated from the final field and from itself along each axis, but `ambient::gen_simplex()`'s 4 input dimensions are already fully spent on the (θ_u, θ_v) torus trick, leaving no spare dimension to offset. `fill_flow()` decorrelates by seed instead: the u- and v-displacement fields are sampled at `seed + 104729L` and `seed + 200003L` respectively (arbitrary large primes, chosen only to make collisions with a user's own nearby seed choices unlikely), before the final field is sampled at seed itself.

Displacing a periodic field's own periodic coordinates by another periodic field preserves periodicity: shifting u from 0 to 1 still advances θ_u by exactly $2 * \pi$ (a full turn), and the warp added at each end is identical since it's sampled from a field that's periodic in u itself – so the warped angle wraps around exactly as the unwrapped one did, with no seam.

As with [fill_noise\(\)](#), a faint tile-boundary rasterization seam can still be visible on some devices despite the field being exactly periodic (see its details); larger warp values tend to make this more noticeable, for the same reason described at [fill_marble\(\)](#).

Value

A pattern object as returned by [grid::pattern\(\)](#), suitable for use as the fill argument to [grid::gpar\(\)](#).

See Also

Other fill helpers: [fill\(\)](#), [fill_charcoal\(\)](#), [fill_checker\(\)](#), [fill_crosshatch\(\)](#), [fill_gradient\(\)](#), [fill_halftone\(\)](#), [fill_hatch\(\)](#), [fill_image\(\)](#), [fill_marble\(\)](#), [fill_noise\(\)](#), [fill_none\(\)](#), [fill_scatter\(\)](#), [fill_scribble\(\)](#), [fill_solid\(\)](#), [fill_stipple\(\)](#), [fill_stripe\(\)](#), [fill_vignette\(\)](#)

Examples

```
draw(shape_circle(fill = fill_flow(warp = 3, seed = 9350L)))
```

```
# a larger warp gives a swirlier, more curl-noise-like look; warp = 0
# reduces to plain fill_noise()
```

```

draw(shape_circle(fill = fill_flow(warp = 0, seed = 9350L)))
draw(shape_circle(fill = fill_flow(warp = 6, seed = 9350L)))

# two or more colours blend across the field, as in fill_noise()
draw(shape_circle(
  fill = fill_flow(color = c("steelblue", "white", "tomato"), seed = 9350L)
))

```

fill_gradient

Gradient fill

Description

fill_gradient() builds a `grid::pattern()` fill value whose tile content is a single rectangle filled with a `grid::linearGradient()` or `grid::radialGradient()`, depending on type.

Usage

```

fill_gradient(
  color = c("white", "black"),
  type = c("linear", "radial"),
  angle = 45,
  stops = NULL,
  spacing = 1,
  aspect = NULL,
  extend = "pad"
)

```

Arguments

color	Two or more colours to interpolate between.
type	Either "linear" or "radial". Default "linear".
angle	Gradient direction in degrees, for type = "linear" only (ignored for "radial"). Default 45.
stops	Colour stop positions, as a numeric vector the same length as color, or NULL to space them evenly (the default used by <code>grid::linearGradient()</code> / <code>grid::radialGradient()</code>). Default NULL.
spacing	Tile size, as a fraction of the target's bounding box. Must be a positive number. Default 1 (one tile spans the whole shape).
aspect	Width-to-height ratio of the target polygon's bounding box. Must be a positive number, or NULL (the default) to resolve it automatically at <code>draw()</code> time – see <code>fill_hatch()</code> 's own aspect docs.
extend	Passed to the inner <code>grid::linearGradient()</code> / <code>grid::radialGradient()</code> , controlling what happens beyond the colour stops. Default "pad".

Details

Like the other `fill_*`() helpers, this needs the target's bounding-box aspect ratio (aspect, resolved automatically by default – see `fill_hatch()`'s own aspect docs) to render true rather than stretched – but the correction is applied differently here. Rather than adjusting the gradient's own coordinates (the way `fill_hatch()` adjusts a segment's direction), `fill_gradient()` corrects the *tile* itself to be physically square, exactly as `fill_stipple()` does for its dots: once the tile is square, a gradient specified inside it in plain "npc" needs no further correction to render at the requested angle, or as a true circle for `type = "radial"`.

This also means a gradient tile has none of `fill_hatch()`'s periodicity concerns: adjacent tiles are simply identical copies of the same square gradient, with nothing analogous to a hatch line's tile-edge dashing to avoid.

With the default `spacing = 1`, one tile spans (and, for a non-square bounding box, slightly overshoots) the target's entire bounding box, giving a single smooth gradient across the whole shape – the overshoot is invisibly clipped away by the target's own outline. Set `spacing < 1` for a repeating pattern of small gradient motifs instead.

Value

A pattern object as returned by `grid::pattern()`, suitable for use as the `fill` argument to `grid::gpar()`.

See Also

Other fill helpers: `fill()`, `fill_charcoal()`, `fill_checker()`, `fill_crosshatch()`, `fill_flow()`, `fill_halftone()`, `fill_hatch()`, `fill_image()`, `fill_marble()`, `fill_noise()`, `fill_none()`, `fill_scatter()`, `fill_scribble()`, `fill_solid()`, `fill_stipple()`, `fill_stripe()`, `fill_vignette()`

Examples

```
draw(shape_circle(fill = fill_gradient(c("white", "steelblue"))))
draw(shape_circle(fill = fill_gradient(c("yellow", "red"), type = "radial"))

# three or more colours interpolate in sequence; angle rotates a linear
# gradient's direction
draw(shape_circle(
  fill = fill_gradient(c("yellow", "orange", "red"), angle = 90)
))

# spacing < 1 repeats the gradient as a small tiled motif instead of one
# smooth sweep across the whole shape
draw(shape_circle(
  fill = fill_gradient(c("white", "steelblue"), spacing = 0.3)
))
```

fill_halftone	<i>Halftone dot pattern fill</i>
---------------	----------------------------------

Description

fill_halftone() is fill_stipple()'s other variant: instead of scattering dots of one fixed size, each dot's radius is drawn uniformly at random from radius, giving a mottled halftone-print look rather than a uniform stipple.

Usage

```
fill_halftone(
  color = "black",
  radius = c(0.05, 0.2),
  spacing = 0.3,
  aspect = NULL,
  n = 4L,
  seed = 1L,
  extend = "repeat"
)
```

Arguments

color	One or more dot colours. A vector shorter than n is recycled (in order, not randomly) across the scattered dots – a single colour (the default) colours every dot the same, matching the original behaviour. Default "black".
radius	Dot radius range, as a length-2 numeric vector giving the "npc"-fraction-of-tile minimum and maximum (a dot's actual radius is drawn uniformly from this range). Both values must be positive, and the first must be no larger than the second. Default c(0.05, 0.2).
spacing	Baseline tile size, as a fraction of the target's bounding box. Must be a positive number. Default 0.3.
aspect	Width-to-height ratio of the target polygon's bounding box. Must be a positive number, or NULL (the default) to resolve it automatically from the real target's own bounding-box aspect ratio at draw() time – see the fill class. Passing a fixed number instead computes the pattern once, immediately, against that value only.
n	Number of dots scattered per tile. Must be a positive integer. Default 4L.
seed	Integer seed for the dot positions. Default 1L.
extend	Passed to grid::pattern(). Default "repeat".

Details

Like fill_stipple() (and unlike fill_scatter()), it scatters plain grid::circleGrob()s, so it's immune to the *polygon*-specific rendering problems documented at fill_scatter() – but it

shares `fill_stipple()`'s own, separate "Known rendering risk with multiple dots" (repeated tiles containing several circleGrobs were, in testing, sometimes visibly distorted on this package's development R build; see that section for details). There is no known way to avoid this while still getting a genuine scattered-dot texture, so **check rendered output visually** here too.

As with `fill_stipple()`, `aspect` corrects for the target's own bounding-box aspect ratio automatically by default, keeping the dots circular rather than elliptical (see `fill_hatch()`'s own aspect docs). Dot centres are kept at least `max(radius)` from each tile edge, so even the largest possible dot isn't clipped away near a boundary.

Value

A pattern object as returned by `grid::pattern()`, suitable for use as the `fill` argument to `grid::gpar()`.

See Also

Other fill helpers: `fill()`, `fill_charcoal()`, `fill_checker()`, `fill_crosshatch()`, `fill_flow()`, `fill_gradient()`, `fill_hatch()`, `fill_image()`, `fill_marble()`, `fill_noise()`, `fill_none()`, `fill_scatter()`, `fill_scribble()`, `fill_solid()`, `fill_stipple()`, `fill_stripe()`, `fill_vignette()`

Examples

```
draw(shape_circle(
  fill = fill_halftone(radius = c(0.05, 0.15), seed = 3187L)
))

# a wider radius range gives more size contrast between dots
draw(shape_circle(
  fill = fill_halftone(radius = c(0.02, 0.22), n = 6L, seed = 3187L)
))

# a colour vector is recycled across the dots, as in fill_stipple()
draw(shape_circle(
  fill = fill_halftone(
    color = c("steelblue", "tomato"),
    n = 8L,
    seed = 3187L
  )
))
```

fill_hatch

Diagonal hatch pattern fill

Description

`fill_hatch()` builds a `grid::pattern()` fill value that renders a repeating diagonal hatch line. It's meant to be used as the `fill` argument to `grid::gpar()` (and eventually `style()`'s `fill` property), in place of a plain colour.

Usage

```
fill_hatch(
  color = "black",
  angle = 45,
  spacing = 0.1,
  aspect = NULL,
  linewidth = 1,
  extend = "repeat"
)
```

Arguments

color	Line colour. Default "black".
angle	Hatch angle in degrees, measured counterclockwise from the positive x-axis. Default 45.
spacing	Baseline tile size, as a fraction of the target's bounding box. Must be a positive number. Default 0.1.
aspect	Width-to-height ratio of the target polygon's bounding box. Must be a positive number, or NULL (the default) to resolve it automatically from the real target's own bounding-box aspect ratio at <code>draw()</code> time – see the <code>fill</code> class. Passing a fixed number instead computes the pattern once, immediately, against that value only.
linewidth	Line width. Must be a positive number. Default 1.
extend	Passed to <code>grid::pattern()</code> . Default "repeat".

Details

`grid::pattern()` tiles are sized as a *fraction of the target polygon's own bounding box*, not a fixed physical square, so a tile that looks square in that relative sense can be a stretched rectangle in absolute terms whenever the target's bounding box isn't square itself – which distorts any angle baked directly into the pattern content. This is corrected automatically: `aspect` defaults to `NULL`, resolved against the real target's own bounding-box aspect ratio (width / height) at `draw()` time (see the `fill` class); pass a fixed number instead to compute the pattern once, immediately, against that value only.

Internally, the hatch line is always drawn as a plain diagonal from one tile corner to the opposite corner (or the mirror image, for a negative-sloped angle) – never at an arbitrary slope baked into the segment's own coordinates. A corner-to-corner diagonal is the only slope that tiles seamlessly under `grid::pattern()`'s `extend = "repeat"`, which translates tile copies by whole tile-widths/heights only; any other local slope leaves a visible mismatch ("dashing") at every tile edge. The desired angle is instead achieved entirely by choosing the tile's width/height ratio. Exactly horizontal/vertical angles are handled as a special case, since a straight (non-diagonal) line tiles seamlessly at any tile aspect ratio.

Value

A pattern object as returned by `grid::pattern()`, suitable for use as the `fill` argument to `grid::gpar()`.

See Also

Other fill helpers: [fill\(\)](#), [fill_charcoal\(\)](#), [fill_checker\(\)](#), [fill_crosshatch\(\)](#), [fill_flow\(\)](#), [fill_gradient\(\)](#), [fill_halftone\(\)](#), [fill_image\(\)](#), [fill_marble\(\)](#), [fill_noise\(\)](#), [fill_none\(\)](#), [fill_scatter\(\)](#), [fill_scribble\(\)](#), [fill_solid\(\)](#), [fill_stipple\(\)](#), [fill_stripe\(\)](#), [fill_vignette\(\)](#)

Examples

```
draw(shape_circle(fill = fill_hatch(angle = 30, spacing = 0.15)))

# a steeper angle and finer spacing
draw(shape_circle(fill = fill_hatch(angle = 75, spacing = 0.06)))

# exactly horizontal/vertical are handled as a special case (no
# diagonal-tile trick needed)
draw(shape_circle(fill = fill_hatch(angle = 0, spacing = 0.1)))

# aspect corrects the rendered angle for a non-square bounding box --
# without it, a 45 degree hatch looks skewed on a wide rectangle
draw(shape_rectangle(width = 3, height = 1, fill = fill_hatch(angle = 45)))
draw(shape_rectangle(
  width = 3,
  height = 1,
  fill = fill_hatch(angle = 45, aspect = 3)
))
```

fill_image

User-supplied raster image fill

Description

`fill_image()` builds a [grid::pattern\(\)](#) fill value whose tile content is an arbitrary bitmap, rather than a texture generated by this package. It plugs any raster the caller already has in memory into the same `grid::pattern()` machinery [fill_noise\(\)](#) uses for its own generated raster – but with the raster supplied, not computed.

Usage

```
fill_image(
  image,
  preserve_aspect = TRUE,
  spacing = 1,
  aspect = NULL,
  interpolate = TRUE,
  extend = "repeat"
)
```

Arguments

image	A raster image: a "raster" object, or a matrix/array accepted by <code>grDevices::as.raster()</code> . Not a file path – read the file first with e.g. <code>png::readPNG()</code> , <code>jpeg::readJPEG()</code> , or <code>magick::image_read()</code> .
preserve_aspect	Whether to letterbox the image to preserve its own pixel aspect ratio (TRUE), or stretch it to exactly fill each tile (FALSE). Default TRUE.
spacing	Tile size, as a fraction of the target's bounding box. Must be a positive number. Default 1 (one tile spans the whole shape).
aspect	Width-to-height ratio of the target polygon's bounding box. Must be a positive number, or NULL (the default) to resolve it automatically from the real target's own bounding-box aspect ratio at <code>draw()</code> time – see the <code>fill</code> class. Passing a fixed number instead computes the pattern once, immediately, against that value only.
interpolate	Passed to <code>grid::rasterGrob()</code> . Default TRUE.
extend	Passed to <code>grid::pattern()</code> . Default "repeat".

Details

`fill_image()` doesn't read image files itself, to avoid adding an image I/O dependency (png/jpeg/magick, ...) to a package that otherwise has none. Load the file first with whichever of those packages is already on hand, then pass the result (or `grDevices::as.raster()` of it) as `image` – e.g. `fill_image(png::readPNG("logo.png"))` or `fill_image(magick::image_read("logo.png"))`. Anything `grDevices::as.raster()` accepts works: a "raster" object, a character matrix of colour strings, or a numeric array of 0-1 RGB/RGBA intensities.

Unlike the other `fill_*()` helpers, `fill_image()`'s content has its *own* pixel aspect ratio to account for, on top of the usual target-bounding-box correction every helper needs (aspect, defaulting to automatic resolution exactly as every other helper – see `fill_hatch()`'s own aspect docs – and corrected for exactly as `fill_noise()` does – keeping the tile physically square). With `preserve_aspect = TRUE` (the default), the image is letterboxed to fit that square tile without distorting it: whichever of width/height is smaller in the image's own pixel dimensions is shrunk to fit, leaving the tile's remaining margin transparent. Set `preserve_aspect = FALSE` to stretch the image to fill the tile exactly instead, matching `fill_noise()`'s own (always-stretched) behaviour.

Value

A pattern object as returned by `grid::pattern()`, suitable for use as the `fill` argument to `grid::gpar()`.

See Also

Other fill helpers: `fill()`, `fill_charcoal()`, `fill_checker()`, `fill_crosshatch()`, `fill_flow()`, `fill_gradient()`, `fill_halftone()`, `fill_hatch()`, `fill_marble()`, `fill_noise()`, `fill_none()`, `fill_scatter()`, `fill_scribble()`, `fill_solid()`, `fill_stipple()`, `fill_stripe()`, `fill_vignette()`

Examples

```
img <- matrix(c("red", "white", "white", "blue"), nrow = 2)
draw(shape_circle(fill = fill_image(img, preserve_aspect = FALSE)))

# a non-square image, letterboxed (default) vs. stretched to fill the tile
wide_img <- matrix(c("red", "white", "blue"), nrow = 1)
draw(shape_rectangle(width = 2, height = 1, fill = fill_image(wide_img)))
draw(shape_rectangle(
  width = 2,
  height = 1,
  fill = fill_image(wide_img, preserve_aspect = FALSE)
))
```

fill_marble

Marbled, veined texture fill

Description

fill_marble() builds a `grid::pattern()` fill value resembling veined marble: a set of stripes parallel bands running around the tile's u axis, displaced by a `fill_noise()`-style torus-periodic turbulence field (via the shared internal `torus_grid()/torus_noise()` helpers) rather than left straight – the classic "sine of a coordinate plus turbulence" recipe for procedural marble.

Usage

```
fill_marble(
  color = c("white", "black"),
  spacing = 0.5,
  aspect = NULL,
  resolution = 32L,
  stripes = 3L,
  warp = 4,
  frequency = 1,
  octaves = 3L,
  seed = 1L,
  extend = "repeat"
)
```

Arguments

color	Two or more colours blended across each band, in order, via <code>grDevices::colorRamp()</code> . Default <code>c("white", "black")</code> .
spacing	Baseline tile size, as a fraction of the target's bounding box. Must be a positive number. Default <code>0.5</code> .

aspect	Width-to-height ratio of the target polygon's bounding box. Must be a positive number, or NULL (the default) to resolve it automatically from the real target's own bounding-box aspect ratio at <code>draw()</code> time – see the <code>fill</code> class. Passing a fixed number instead computes the pattern once, immediately, against that value only.
resolution	Raster resolution (pixels per tile edge). Must be a positive integer of at least 2L. Default 32L.
stripes	Number of bands running around the tile's u axis before turbulence displacement. Must be a positive integer. Default 3L.
warp	Turbulence amplitude, in radians of displacement along the band coordinate. Must be a non-negative number. Default 4.
frequency	Noise frequency, as in <code>shape_blob()</code> . Must be non-negative. Default 1.
octaves	Number of turbulence octaves, as in <code>shape_blob()</code> . Must be a positive integer. Default 3L.
seed	Integer seed for the noise field, as in <code>shape_blob()</code> . Default 1L.
extend	Passed to <code>grid::pattern()</code> . Default "repeat".

Details

Periodicity needs two things, both already true here: the undisplaced bands (`sin(theta_u * stripes)`) are periodic in u for any *integer* stripes, since `theta_u` itself advances by exactly $2 * \pi$ over one tile width; and the turbulence added on top is periodic in both u and v because it comes from `torus_noise()`, the same torus-sampling technique `fill_noise()` uses for its own field. Adding one periodic function to another (here, inside a further `sin()`) stays periodic, so the combined result still tiles with no seam.

Unlike `fill_noise()` (opacity of one colour), the banding is rendered as a blend across color via `grDevices::colorRamp()`, since a marble texture's visual interest is the veining pattern itself rather than a fade to transparency.

`fill_noise()`'s own faint tile-boundary rasterization seam (see its details) can be more noticeable here: `sin()` turns a small mismatch in the turbulence field into a visibly sharper edge in a band than the same mismatch would produce in a plain opacity fade.

Value

A pattern object as returned by `grid::pattern()`, suitable for use as the `fill` argument to `grid::gpar()`.

See Also

Other fill helpers: `fill()`, `fill_charcoal()`, `fill_checker()`, `fill_crosshatch()`, `fill_flow()`, `fill_gradient()`, `fill_halftone()`, `fill_hatch()`, `fill_image()`, `fill_noise()`, `fill_none()`, `fill_scatter()`, `fill_scribble()`, `fill_solid()`, `fill_stipple()`, `fill_stripe()`, `fill_vignette()`

Examples

```
draw(shape_circle(fill = fill_marble(stripes = 4L, seed = 1274L)))

# more stripes and stronger warp give busier, more tangled veining
```

```

draw(shape_circle(fill = fill_marble(stripes = 8L, warp = 8, seed = 1274L)))

# a different colour pair changes the veining's contrast entirely
draw(shape_circle(
  fill = fill_marble(
    color = c("black", "goldenrod"),
    stripes = 4L,
    seed = 1274L
  )
))

# three or more colours blend across the band in sequence
draw(shape_circle(
  fill = fill_marble(
    color = c("black", "goldenrod", "white"),
    stripes = 4L,
    seed = 1274L
  )
))

```

fill_noise

Simplex/fractal noise texture fill

Description

fill_noise() builds a `grid::pattern()` fill value from a rasterised simplex/fractal noise field, using the same noise machinery as `shape_blob()`'s wobbly outline (`ambient::fracture()` / `ambient::gen_simplex()` / `ambient::fbm()`), with matching frequency, octaves, and seed arguments), so a noise-filled shape and a noise-wobbled outline share one visual vocabulary.

Usage

```

fill_noise(
  color = "black",
  spacing = 0.5,
  aspect = NULL,
  resolution = 32L,
  alpha = 1,
  frequency = 1,
  octaves = 2L,
  seed = 1L,
  extend = "repeat"
)

```

Arguments

color	One or more fill colours. A single colour (the default) fades in opacity from fully transparent to alpha, exactly as before this argument accepted a vector. Two or
-------	---

	more colours instead blend across a <code>grDevices::colorRamp()</code> built from them, driven by the noise value, with alpha applied as a flat opacity across the whole fill (see the internal <code>noise_to_pixels()</code> helper). Default "black".
spacing	Baseline tile size, as a fraction of the target's bounding box. Must be a positive number. Default 0.5.
aspect	Width-to-height ratio of the target polygon's bounding box. Must be a positive number, or NULL (the default) to resolve it automatically from the real target's own bounding-box aspect ratio at <code>draw()</code> time – see the <code>fill</code> class. Passing a fixed number instead computes the pattern once, immediately, against that value only.
resolution	Raster resolution (pixels per tile edge). Must be a positive integer of at least 2L. Default 32L.
alpha	Opacity. For a single color, the maximum opacity at the noise field's peak; for two or more, a flat opacity applied uniformly. Must be a number in (0, 1]. Default 1.
frequency	Noise frequency, as in <code>shape_blob()</code> . Must be non-negative. Default 1.
octaves	Number of noise octaves, as in <code>shape_blob()</code> . Must be a positive integer. Default 2L.
seed	Integer seed for the noise field, as in <code>shape_blob()</code> . Default 1L.
extend	Passed to <code>grid::pattern()</code> . Default "repeat".

Details

Noise is rendered as varying opacity of a single color, from fully transparent at the noise field's minimum to alpha at its maximum – a mottled, cloud-like texture rather than a hard-edged one.

A raster tile has no baked-in direction the way a hatch line does, so `fill_hatch()`'s tile-edge "dashing" problem doesn't apply directly, but an *ordinary* noise field still isn't periodic, and `grid::pattern()`'s `extend = "repeat"` will visibly seam wherever one tile edge fails to match the next. `fill_noise()` avoids this by sampling the noise on a torus: each raster pixel's (u, v) tile coordinate is mapped onto a pair of circles (`ambient::gen_simplex()`'s 4 dimensions, x/y for u and z/t for v) rather than sampled directly, so the field is mathematically periodic in both directions and tiles with no visible seam, at the cost of the noise "wrapping around" within each tile rather than varying smoothly across a larger area.

In practice, a very faint seam can still be visible at tile boundaries on some devices, even though the underlying field is exactly periodic; this appears to be an artifact of how the graphics device samples a repeated raster tile (it persists regardless of `interpolate` and doesn't improve with higher resolution), not a flaw in the noise field itself, and is far subtler than the tile-edge mismatch `fill_hatch()` has to actively avoid.

Value

A pattern object as returned by `grid::pattern()`, suitable for use as the `fill` argument to `grid::gpar()`.

See Also

Other fill helpers: [fill\(\)](#), [fill_charcoal\(\)](#), [fill_checker\(\)](#), [fill_crosshatch\(\)](#), [fill_flow\(\)](#), [fill_gradient\(\)](#), [fill_halftone\(\)](#), [fill_hatch\(\)](#), [fill_image\(\)](#), [fill_marble\(\)](#), [fill_none\(\)](#), [fill_scatter\(\)](#), [fill_scribble\(\)](#), [fill_solid\(\)](#), [fill_stipple\(\)](#), [fill_stripe\(\)](#), [fill_vignette\(\)](#)

Examples

```
draw(shape_circle(fill = fill_noise(seed = 8843L)))

# higher frequency/octaves give finer-grained texture
draw(shape_circle(
  fill = fill_noise(frequency = 4, octaves = 4L, seed = 8843L)
))

# a lower alpha lets the shape's own outline/background show through more
draw(shape_circle(
  fill = fill_noise(color = "steelblue", alpha = 0.5, seed = 8843L)
))

# two or more colours blend across the field instead of fading to transparent
draw(shape_circle(
  fill = fill_noise(color = c("steelblue", "white", "tomato"), seed = 8843L)
))
```

fill_none

Unfilled (transparent) fill

Description

`fill_none()` leaves a [drawable](#)'s interior entirely unfilled, while still stroking its outline (per `style()`'s `color/linewidth`). It's a thin, self-documenting wrapper around `fill_solid(NA_character_)`: `NA` is already a valid colour to [grid::gpar\(\)](#) (rendered as fully transparent), but spelling that out as `fill_none()` reads more clearly at a call site than a bare `NA_character_`, and groups discoverably with the rest of the `fill_*`() family.

Usage

```
fill_none()
```

Details

Since every [drawable](#) is currently rendered as a closed [grid::polygonGrob\(\)](#) (see the "Deferred: open/stroked curve support" item in [.agents/PLAN.md](#)), `fill_none()` gives an unfilled *closed* outline – the edge connecting the last point back to the first is still drawn. It does not, by itself, produce an open/unstroked curve.

Value

A [fill](#) object wrapping NA_character_.

See Also

Other fill helpers: [fill\(\)](#), [fill_charcoal\(\)](#), [fill_checker\(\)](#), [fill_crosshatch\(\)](#), [fill_flow\(\)](#), [fill_gradient\(\)](#), [fill_halftone\(\)](#), [fill_hatch\(\)](#), [fill_image\(\)](#), [fill_marble\(\)](#), [fill_noise\(\)](#), [fill_scatter\(\)](#), [fill_scribble\(\)](#), [fill_solid\(\)](#), [fill_stipple\(\)](#), [fill_stripe\(\)](#), [fill_vignette\(\)](#)

Examples

```
draw(shape_circle(fill = fill_none(), linewidth = 2))

# still a closed outline: the edge from the last point back to the
# first is drawn even though the interior isn't filled
draw(shape_polygon(
  n = 5L,
  fill = fill_none(),
  color = "steelblue",
  linewidth = 3
))
```

fill_scatter

Scattered-shape pattern fill

Description

[fill_scatter\(\)](#) generalizes [fill_stipple\(\)](#): instead of a fixed dot, it scatters copies of an arbitrary small [drawable](#) – rendered with its own style (colour, fill, linewidth), which may itself be another [fill_*\(\)](#) pattern – at random positions inside each tile, using [withr::with_seed\(\)](#) for reproducibility exactly as [fill_stipple\(\)](#) does.

Usage

```
fill_scatter(
  unit = shape_circle(radius = 1),
  n = 6L,
  size = 0.2,
  color = NULL,
  spacing = 1,
  aspect = NULL,
  seed = 1L,
  extend = "repeat"
)
```

Arguments

unit	A small drawable to scatter copies of. Default <code>shape_circle(radius = 1)</code> .
n	Number of copies scattered per tile. Must be a positive integer. Default 6L.
size	unit's rescaled size, as a "npc" fraction of the tile. Must be a number strictly between 0 and 1. Default 0.2.
color	NULL, or one or more colours overriding <code>unit@style@color</code> for each stamp, recycled (in order, not randomly) across the n copies. NULL (the default) colours every stamp from unit's own style, as before this argument existed.
spacing	Tile size, as a fraction of the target's bounding box. Must be a positive number. Default 1 (one tile spans the whole shape, since <code>spacing < 1</code> risks the tiling distortion described above).
aspect	Width-to-height ratio of the target polygon's bounding box. Must be a positive number, or NULL (the default) to resolve it automatically at <code>draw()</code> time – see fill_hatch() 's own aspect docs.
seed	Integer seed for the scatter positions. Default 1L.
extend	Passed to grid::pattern() . Default "repeat".

Details

unit's own points are rescaled (preserving its own aspect ratio) to a bounding box of size `size` and re-centred at each scattered position; its absolute coordinates, position, and radius/width/etc. don't matter, only its shape.

This needed two corrections neither [fill_stipple\(\)](#)'s circles nor [fill_gradient\(\)](#)'s/[fill_checker\(\)](#)'s rectangles did:

- Every other `fill_*()` helper's tile-squaring correction ($\text{height} = \text{spacing} * \text{aspect}$, keeping the tile physically square) was, by itself, enough to keep circular/rectangular content correctly proportioned. Arbitrary polygon content does not get the same treatment: empirically, a [grid::polygonGrob\(\)](#) (or [grid::pathGrob\(\)](#)) used as pattern content renders as though it inherits the *target's own, uncorrected* bounding-box distortion directly, regardless of the tile-squaring correction applied around it – confirmed by testing a hand-built circular polygon side by side with an equivalent [grid::circleGrob\(\)](#) in the same corrected tile: the circle stayed circular, the polygon became an ellipse. So `fill_scatter()` applies a second, explicit correction directly to unit's own vertex x-coordinates (dividing by aspect) on top of the usual tile-squaring.
- Repeated (tiled) polygon content can render with visible clipping artifacts on Cairo devices – confirmed interactively: a single stamp, comfortably inside its tile's margins, rendered as a clean shape when the tile spans the whole target (`spacing = 1`, so `extend = "repeat"` is present but never actually exercised within the visible, clipped area) but as a "bitten" partial shape once `spacing < 1` made the device actually tile multiple copies. This matches [grid::pattern\(\)](#)'s own documented warning that "on Cairo devices, use of clipping in the pattern definition should be avoided because it is very likely to result in distortion of the pattern tile." Circles/rectangles/rasters didn't show this in the rest of the family, but arbitrary polygon geometry did. `spacing` therefore defaults to 1 here (one tile spans the whole shape, scattering all n copies across it at once) rather than the smaller, densely-tiled defaults used elsewhere;

setting `spacing < 1` is still possible for a repeating scattered motif, but may show this distortion. (Later testing on `fill_stipple()` found the same *actually-repeated tile with multiple shapes* combination distorts circleGrob content too, not just polygons – see its "Known rendering risk" section. A single tile with multiple shapes, as used by this function's default, was never observed to have the problem; only real repetition, `spacing < 1`, was.)

Value

A pattern object as returned by `grid::pattern()`, suitable for use as the fill argument to `grid::gpar()`.

See Also

Other fill helpers: `fill()`, `fill_charcoal()`, `fill_checker()`, `fill_crosshatch()`, `fill_flow()`, `fill_gradient()`, `fill_half-tone()`, `fill_hatch()`, `fill_image()`, `fill_marble()`, `fill_noise()`, `fill_none()`, `fill_scribble()`, `fill_solid()`, `fill_stipple()`, `fill_stripe()`, `fill_vignette()`

Examples

```
draw(shape_circle(
  fill = fill_scatter(unit = shape_circle(radius = 1), n = 8L, size = 0.15)
))

# any small drawable works as the scattered unit, e.g. a triangle
draw(shape_circle(
  fill = fill_scatter(
    unit = shape_polygon(n = 3, fill = "steelblue"),
    n = 10L,
    size = 0.2
  )
))

# a color vector overrides unit's own style colour, recycled per stamp
draw(shape_circle(
  fill = fill_scatter(
    color = c("steelblue", "tomato", "goldenrod"),
    n = 9L,
    size = 0.15
  )
))
```

fill_scribble

Wandering-line scribble texture fill

Description

`fill_scribble()` builds a `grid::pattern()` fill value from several randomly wandering lines, each a random finite sum of sine harmonics (see the internal `scribble_lines()` helper) rather than a smooth noise field or a scattered discrete motif – giving a loose, hand-drawn scribble texture built from genuinely continuous strokes.

Usage

```
fill_scribble(
  color = "black",
  angle = 0,
  n_lines = 5L,
  n_harmonics = 3L,
  amplitude = 0.35,
  resolution = 200L,
  linewidth = 1,
  spacing = 0.25,
  aspect = NULL,
  seed = 1L,
  extend = "repeat"
)
```

Arguments

color	One or more line colours. A vector shorter than <code>n_lines</code> is recycled (in order, not randomly) across the wandering lines – a single colour (the default) colours every line the same, matching the original behaviour. Default "black".
angle	Line angle in degrees, measured counterclockwise from horizontal (matching <code>fill_hatch()</code> 's own angle convention). 0/90/180/270 (the old direction = "horizontal"/"vertical") tile exactly seamlessly; any other angle rotates the tile content as an approximation only – see "Known limitation" below. Default 0.
n_lines	Number of wandering lines per tile. Must be a positive integer. Default 5L.
n_harmonics	Number of sine harmonics summed per line. Must be a positive integer. Default 3L.
amplitude	Maximum total wiggle amplitude, as a "npc" fraction of the tile, split across <code>n_harmonics</code> (so more harmonics each contribute proportionally less). Must be a non-negative number. Default 0.35.
resolution	Number of points sampled along each line. Must be a positive integer of at least 2L. Default 200L.
linewidth	Line width. Must be a positive number. Default 1.
spacing	Baseline tile size, as a fraction of the target's bounding box. Must be a positive number. Default 0.25.
aspect	Width-to-height ratio of the target polygon's bounding box. Must be a positive number, or NULL (the default) to resolve it automatically from the real target's own bounding-box aspect ratio at <code>draw()</code> time – see the <code>fill</code> class. Passing a fixed number instead computes the pattern once, immediately, against that value only.
seed	Integer seed for the random harmonics. Default 1L.
extend	Passed to <code>grid::pattern()</code> . Default "repeat".

Details

A wandering *open* line poses a tiling problem none of the other `fill_*()` helpers have: `fill_hatch()`'s diagonal and `fill_noise()`'s raster both tile by construction (a straight corner-to-corner line, or an already-periodic field), and `fill_stipple()/fill_scatter()/fill_halftone()` sidestep the problem entirely by keeping their scattered content margined well clear of the tile edge. A wandering line that's meant to look continuous *can't* stay clear of the edge – it has to run all the way to it, and pick up again at exactly the right place, in both position and slope, on the opposite edge, or the seam shows as a visible kink. `fill_scribble()` gets this for free by building each line as a random sum of sine harmonics at *integer* frequencies only: over one full period, such a sum always returns exactly to its starting value and slope (to floating-point precision), so consecutive tile copies join with no visible seam – confirmed visually with `extend = "repeat"` at small spacing, including repeated copies (unlike the polygon-in-a-genuinely-repeated-tile issue documented at `fill_scatter()`, open-line content showed no clipping or distortion in testing).

Value

A pattern object as returned by `grid::pattern()`, suitable for use as the `fill` argument to `grid::gpar()`.

Known limitation – arbitrary angle is only approximate

Every other angled helper (`fill_hatch()/fill_crosshatch()/fill_stripe()`) achieves an arbitrary angle by reshaping the *tile* itself (via `hatch_tile_dims()`) around content that's a plain corner-to-corner diagonal. That trick doesn't generalize here: reshaping the tile around a *wandering* line just anisotropically stretches its wiggle rather than rotating it, since the line's content isn't a bare diagonal the tile shape can reinterpret. A genuinely rotated wandering line would need the tile built as a rotated/sheared parallelogram with edge-matching worked out for a curve rather than a segment – no such technique exists in this package. At angle a multiple of 90 degrees, `fill_scribble()` instead swaps/reflects the along/across axes directly (no trigonometry involved), so tiling stays exactly seamless – this is what `angle = 0/90` (the old `direction = "horizontal"/"vertical"`) already relied on, now extended to 180/270 for free. Any other angle instead rotates the whole tile's content about its own centre, which is only an approximation: the underlying wandering line is periodic along its own local axis, not around an arbitrary rotated one, so a non-right-angle rotation can leave a visible seam where adjacent tile copies meet, and may leave small gaps near the tile corners that no line's wiggle reaches. Revisit if a real sketch needs a genuinely seamless arbitrary angle.

See Also

Other fill helpers: `fill()`, `fill_charcoal()`, `fill_checker()`, `fill_crosshatch()`, `fill_flow()`, `fill_gradient()`, `fill_halftone()`, `fill_hatch()`, `fill_image()`, `fill_marble()`, `fill_noise()`, `fill_none()`, `fill_scatter()`, `fill_solid()`, `fill_stipple()`, `fill_stripe()`, `fill_vignette()`

Examples

```
draw(shape_circle(fill = fill_scribble(n_lines = 4L, seed = 6602L)))

# angle = 90 runs the wandering lines top-to-bottom instead, tiling
# exactly as seamlessly as angle = 0
draw(shape_circle(
```

```

    fill = fill_scribble(n_lines = 4L, angle = 90, seed = 6602L)
  ))

  # any other angle rotates the tile content -- an approximation only,
  # so a seam can be visible at the tile edges
  draw(shape_circle(
    fill = fill_scribble(n_lines = 4L, angle = 30, seed = 6602L)
  ))

  # more harmonics and higher amplitude give a more agitated scribble
  draw(shape_circle(
    fill = fill_scribble(
      n_lines = 6L,
      n_harmonics = 6L,
      amplitude = 0.6,
      seed = 6602L
    )
  ))

  # a colour vector is recycled across the wandering lines
  draw(shape_circle(
    fill = fill_scribble(
      color = c("steelblue", "tomato"),
      n_lines = 4L,
      seed = 6602L
    )
  ))

```

fill_solid

Solid colour fill

Description

fill_solid() is the trivial member of the fill_*() family: a plain colour needs no [grid::pattern\(\)](#) machinery, since [grid::gpar\(\)](#)'s fill argument already accepts a colour string directly. It exists so that a plain colour can be requested with the same fill_*() naming as the pattern-based helpers (grouping them together in autocomplete), and so style()'s fill property has one uniform family of constructors to accept – including for its own default – once it's extended to take fill_*() outputs alongside a bare colour string.

Usage

```
fill_solid(color = "black")
```

Arguments

color Fill colour. Default "black".

Value

A [fill](#) object wrapping color.

See Also

Other fill helpers: [fill\(\)](#), [fill_charcoal\(\)](#), [fill_checker\(\)](#), [fill_crosshatch\(\)](#), [fill_flow\(\)](#), [fill_gradient\(\)](#), [fill_halftone\(\)](#), [fill_hatch\(\)](#), [fill_image\(\)](#), [fill_marble\(\)](#), [fill_noise\(\)](#), [fill_none\(\)](#), [fill_scatter\(\)](#), [fill_scribble\(\)](#), [fill_stipple\(\)](#), [fill_stripe\(\)](#), [fill_vignette\(\)](#)

Examples

```
fill_solid("steelblue")
draw(shape_circle(fill = fill_solid("tomato")))

# equivalent to passing the colour string directly, since style()'s own
# `fill` default is fill_solid("black")
draw(shape_circle(fill = "tomato"))
```

fill_stipple	<i>Stippled dot pattern fill</i>
--------------	----------------------------------

Description

[fill_stipple\(\)](#) builds a [grid::pattern\(\)](#) fill value that scatters a handful of dots at random positions inside each tile, using [withr::with_seed\(\)](#) so the same seed always reproduces the same scatter (the same convention used by [shape_blob\(\)](#), [shape_ribbon\(\)](#), and [shape_twist\(\)](#)'s noise fields).

Usage

```
fill_stipple(
  color = "black",
  radius = 0.15,
  spacing = 0.3,
  aspect = NULL,
  n = 4L,
  seed = 1L,
  extend = "repeat"
)
```

Arguments

color	One or more dot colours. A vector shorter than n is recycled (in order, not randomly) across the scattered dots – a single colour (the default) colours every dot the same, matching the original behaviour. Default "black".
radius	Dot radius, as a "npc" fraction of the tile. Must be a positive number. Default 0.15.

spacing	Baseline tile size, as a fraction of the target's bounding box. Must be a positive number. Default 0.3.
aspect	Width-to-height ratio of the target polygon's bounding box. Must be a positive number, or NULL (the default) to resolve it automatically from the real target's own bounding-box aspect ratio at <code>draw()</code> time – see the <code>fill</code> class. Passing a fixed number instead computes the pattern once, immediately, against that value only.
n	Number of dots scattered per tile. Must be a positive integer. Default 4L.
seed	Integer seed for the dot positions. Default 1L.
extend	Passed to <code>grid::pattern()</code> . Default "repeat".

Details

Unlike `fill_hatch()/fill_crosshatch()`, a dot has no direction, so there's no analogue of their tile-edge "dashing" problem here. There's still a circularity problem to correct for, though: `grid::pattern()` tiles are sized as a fraction of the target polygon's own bounding box, so a dot drawn with an npc-relative radius renders as an ellipse whenever that bounding box isn't square. `aspect` corrects for this automatically by default, keeping dots circular (see `fill_hatch()`'s own aspect docs).

Value

A pattern object as returned by `grid::pattern()`, suitable for use as the `fill` argument to `grid::gpar()`.

Known rendering risk with multiple dots

On this package's development R build (4.6.1, a very recent/development version), `grid::pattern()` tiles whose content is a *group* of several `grid::circleGrob()`s (i.e. $n > 1$) were found, in some cases, to render individual dots visibly distorted – clipped into crescents or otherwise not circular – even though each dot's own coordinates are correct and a single dot ($n = 1$) always renders correctly. This reproduced in a fresh R session (so it isn't specific to a long interactive session), across multiple n and radius values, with no clean rule found for exactly when it triggers; it appeared on both an interactive device and `ragg::agg_png()`. No fix or reliable workaround was found – this looks like an upstream `grid`/Cairo issue with multi-shape pattern tile content, not something specific to how this function builds its content. **Visually check rendered output** before relying on `fill_stipple()` (or `fill_scatter()/fill_halftone()`, which share this risk) for anything beyond casual use, especially on unfamiliar R/`grid`/graphics-device versions.

See Also

Other fill helpers: `fill()`, `fill_charcoal()`, `fill_checker()`, `fill_crosshatch()`, `fill_flow()`, `fill_gradient()`, `fill_halftone()`, `fill_hatch()`, `fill_image()`, `fill_marble()`, `fill_noise()`, `fill_none()`, `fill_scatter()`, `fill_scribble()`, `fill_solid()`, `fill_stripe()`, `fill_vignette()`

Examples

```
draw(shape_circle(fill = fill_stipple(n = 6L, seed = 2091L)))

# more, smaller dots per tile give a denser stipple
```

```

draw(shape_circle(
  fill = fill_stipple(n = 15L, radius = 0.06, spacing = 0.5, seed = 2091L)
))

# a colour vector is recycled across the dots
draw(shape_circle(
  fill = fill_stipple(color = c("steelblue", "tomato"), n = 8L, seed = 2091L)
))

```

fill_stripe

Striped pattern fill

Description

fill_stripe() builds a [grid::pattern\(\)](#) fill value that renders repeating solid bands, at a given angle, generalized from two colours to an arbitrary palette.

Usage

```

fill_stripe(
  color = c("black", "white"),
  angle = 45,
  spacing = 0.2,
  aspect = NULL,
  extend = "repeat"
)

```

Arguments

color	Two or more stripe colours, one equal-width band each (see details for biasing band widths). Default <code>c("black", "white")</code> .
angle	Hatch angle in degrees, measured counterclockwise from the positive x-axis. Default 45.
spacing	One full period through all of color, as a fraction of the target's bounding box. Must be a positive number. Default 0.2.
aspect	Width-to-height ratio of the target polygon's bounding box. Must be a positive number, or NULL (the default) to resolve it automatically from the real target's own bounding-box aspect ratio at draw() time – see the fill class. Passing a fixed number instead computes the pattern once, immediately, against that value only.
extend	Passed to the inner grid::linearGradient() , controlling what happens beyond the colour stops – <i>not</i> to the outer grid::pattern() call, which always uses <code>extend = "repeat"</code> (see details). Default "repeat".

Details

Unlike `fill_hatch()`, this doesn't use the corner-to-corner-diagonal tile-shape technique at all – because a *filled* band, unlike a thin hatch *line*, needs every point along a tile's edge to match its neighbour, not just the points where a thin line happens to cross, a single diagonal split of a tile turns out not to tile seamlessly at an arbitrary angle the way a thin hatch line does. Instead, `fill_stripe()` sidesteps `grid::pattern()`'s tile-copy repetition altogether: the stripe angle and period come from a `grid::linearGradient()` with hard colour stops (no smooth transition) and `extend = "repeat"`, which repeats *itself* continuously along its own axis – a fundamentally different (and for this purpose, simpler) mechanism than tiling a rasterised copy. `grid::pattern()` is still used around this gradient, but only once, as a single square (automatically aspect-corrected – see `fill_hatch()`'s own aspect docs) tile spanning the whole target shape, exactly as `fill_gradient()` does by default – not to create repetition, which the gradient already provides.

Each of the `n = length(color)` colours gets an equal-width band by default (1/n of the period); there's no separate argument for unequal bands – repeat a colour in `color` instead (e.g. `c("steelblue", "steelblue", "white")` gives a 2:1 ratio between the two colours), which reuses the same recycling mechanism rather than adding a second one.

Value

A pattern object as returned by `grid::pattern()`, suitable for use as the `fill` argument to `grid::gpar()`.

See Also

Other fill helpers: `fill()`, `fill_charcoal()`, `fill_checker()`, `fill_crosshatch()`, `fill_flow()`, `fill_gradient()`, `fill_halftone()`, `fill_hatch()`, `fill_image()`, `fill_marble()`, `fill_noise()`, `fill_none()`, `fill_scatter()`, `fill_scribble()`, `fill_solid()`, `fill_stipple()`, `fill_vignette()`

Examples

```
draw(shape_circle(fill = fill_stripe(angle = 30)))

# repeating a colour biases the band widths, rather than a separate argument
draw(shape_circle(
  fill = fill_stripe(color = c("steelblue", "steelblue", "white"))
))

# narrower spacing gives more, thinner stripes
draw(shape_circle(fill = fill_stripe(angle = 90, spacing = 0.08)))

# three or more colours repeat through the same period
draw(shape_circle(
  fill = fill_stripe(color = c("steelblue", "white", "tomato"), angle = 30)
))
```

fill_vignette

*Vignette fill***Description**

fill_vignette() builds a `grid::pattern()` fill value that fades colour out towards the edges of each tile, using `grid::as.mask()` – the one grid capability the rest of the fill_*() family doesn't touch. A color layer (solid for a single colour, or a `grid::radialGradient()` blend for two or more) is masked by a radial alpha mask (opaque at the tile centre, fully transparent at its edge), optionally revealing a solid background layer underneath rather than true transparency.

Usage

```
fill_vignette(
  color = "black",
  background = NA,
  spacing = 1,
  aspect = NULL,
  extend = "repeat"
)
```

Arguments

color	One or more fill colours, blended from the tile's centre outward. A single colour (the default) is a solid fade, exactly as before this argument accepted a vector; two or more blend via a <code>grid::radialGradient()</code> before the same alpha mask is applied. Default "black".
background	Fill colour revealed as color fades out, or NA for true transparency (showing whatever is drawn behind the target shape). Default NA.
spacing	Tile size, as a fraction of the target's bounding box. Must be a positive number. Default 1 (one tile spans the whole shape).
aspect	Width-to-height ratio of the target polygon's bounding box. Must be a positive number, or NULL (the default) to resolve it automatically at <code>draw()</code> time – see <code>fill_hatch()</code> 's own aspect docs.
extend	Passed to <code>grid::pattern()</code> . Default "repeat".

Details

As with `fill_gradient()`, the fade shape is kept circular by correcting the *tile* to be physically square via aspect (the target's bounding-box width-to-height ratio, resolved automatically by default – see `fill_hatch()`'s own aspect docs), the same technique `fill_stipple()` uses for its dots – once the tile is square, a mask specified inside it in plain "npc" needs no further correction.

A mask must always be built with `grid::as.mask()` and an explicit type = "alpha" here, rather than passed as a bare grob (which defaults to an alpha mask anyway) – during prototyping, a bare mask grob whose own fill was a `grid::radialGradient()` intermittently triggered an "Ignored luminance mask (not supported on this device)" warning on this session's device, even though

the rendered result was visually correct either way. Being explicit with `as.mask(..., type = "alpha")` avoided the warning entirely with an identical render, so that's what's used here; true `grid::as.mask()` luminance masks were found not to work at all in this nested tile context (silently ignored, regardless of explicitness), so `fill_vignette()` only offers the alpha variant.

Value

A pattern object as returned by `grid::pattern()`, suitable for use as the fill argument to `grid::gpar()`.

See Also

Other fill helpers: `fill()`, `fill_charcoal()`, `fill_checker()`, `fill_crosshatch()`, `fill_flow()`, `fill_gradient()`, `fill_halftone()`, `fill_hatch()`, `fill_image()`, `fill_marble()`, `fill_noise()`, `fill_none()`, `fill_scatter()`, `fill_scribble()`, `fill_solid()`, `fill_stipple()`, `fill_stripe()`

Examples

```
draw(shape_circle(fill = fill_vignette(color = "black")))

# a non-NA background reveals a solid colour underneath the fade,
# instead of true transparency
draw(shape_circle(
  fill = fill_vignette(color = "steelblue", background = "white")
))

# two or more colours blend radially before fading via the alpha mask
draw(shape_circle(fill = fill_vignette(color = c("goldenrod", "steelblue"))))
```

group

A nested collection of drawables sharing one transform/style

Description

group bundles several [drawable](#)/group objects (shapes) together with one shared trans and, optionally, one shared style override – the whole group can be rotated/scaled/moved as a unit, or restyled as a unit, without touching any member's own trans/style. This is different from [sketch](#), which represents the whole canvas of independently-styled/-positioned shapes; a group is meant to be one element *inside* a [sketch](#) (or inside another group), not a replacement for it.

Usage

```
group(shapes = list(), trans = trans_identity(), style = NULL)
```

Arguments

shapes	A list of drawable /group objects. Default <code>list()</code> .
trans	A trans/trans_warp/trans_fn/trans_chain , applied to every member as a unit. Default <code>trans_identity()</code> .
style	A style object overriding every descendant drawable's own style, or NULL (default) for no override.

Details

Groups are built up incrementally with `+`, the same way a [sketch](#) is: `group() + shape_circle() + shape_circle(x = 2)`. `group + <trans-like>` composes the transform onto the group's own `@trans`, applied to every member as a unit – it does not touch any member's own `@trans`. A group can itself be added to a [sketch](#) (or nested inside another group), and mixed freely with plain [drawables](#) in either container.

`style`, when set (default NULL, meaning no override), is meant to replace every descendant [drawable](#)'s own `@style` when the group is drawn – a nested group with no style of its own would inherit its ancestor's override; a nested group that sets its own style would keep that instead (the nearest override wins, it does not stack). This cascade is implemented by `draw()`'s own `group` method, not by `group` itself.

See Also

Other core structure: [canvas\(\)](#), [draw\(\)](#), [drawable\(\)](#), [sketch\(\)](#), [style\(\)](#), [xy\(\)](#)

Examples

```
# shape_square() (rather than shape_circle()) makes a rotation visually
# detectable -- a circle's outline is rotationally symmetric about its
# own centroid, so rotating one in place looks identical to the original
g <- group() + shape_square(side = 1) + shape_square(x = 2, side = 0.6)
draw(g)

# a trans applied to a group composes onto every member as a unit,
# without changing any member's own @trans -- both squares rotate about
# the group's own origin, sweeping the smaller one around the larger one
draw(g + trans_rotate(pi / 6))

# a style override, applied to every member when the group is drawn
draw(g + style(color = "tomato", fill = "grey90"))

# groups mix freely with plain drawables inside a sketch, and can nest;
# a nested group's own style override wins over an outer one, and an
# outer group's trans applies on top of everything inside it
inner <- group() + shape_square(x = 0.4, side = 0.3) + style(fill = "steelblue")
outer <- (group() + g + inner + style(color = "tomato")) + trans_translate(1, 0)
draw(sketch() + shape_square(x = -2) + outer)
```

noise_bridge

A smoothed Brownian bridge, as a path distortion

Description

noise_bridge bundles the settings needed to generate a smoothed Brownian bridge (via the internal smooth_bridge() helper): how many local-averaging smooth-ing passes to apply, and which seed to draw from.

Usage

```
noise_bridge(smooth = 3L, seed = 1L)
```

Arguments

smooth	Number of smoothing passes. Must be non-negative. Default 3L.
seed	Integer seed. Default 1L.

Details

Unlike [noise_field](#) (sampled at arbitrary (x, y) positions in the plane), a Brownian bridge has no spatial position to sample at – [noise_sample\(\)](#)'s method for noise_bridge instead takes a point count n and a scale, returning a length-n displacement vector. This is the distortion behind [shape_twist\(\)](#)'s wandering path, split out into its own class for the same reason [noise_field](#) was: so the distortion is a first-class, swappable object rather than bare constructor arguments, and so a future curve_twist() (an open, unfilled wandering path) can reuse it without duplicating shape_twist()'s path logic.

See Also

Other noise helpers: [noise_field\(\)](#), [noise_sample\(\)](#)

Examples

```
noise_bridge(smooth = 5L, seed = 4821)

# more smoothing passes give a gentler bridge; embedding it in
# shape_twist()'s path_distortion makes the effect easy to see
draw(shape_twist(
  x = 0, y = 0, xend = 1, yend = 0, width = 0.2,
  path_distortion = noise_bridge(smooth = 0L, seed = 7734)
))
draw(shape_twist(
  x = 0, y = 0, xend = 1, yend = 0, width = 0.2,
  path_distortion = noise_bridge(smooth = 20L, seed = 7734)
))
```

noise_field

A sampled scalar noise field

Description

noise_field bundles the settings needed to sample a scalar-valued noise field from **ambient** at arbitrary (x, y) positions: which base noise function and fractal combinator to use (`ambient::fracture()`'s noise/fractal arguments), how coarse or fine the field is (frequency), how many octaves of detail to layer (octaves), and which seed to draw from.

Usage

```
noise_field(
  noise = ambient::gen_simplex,
  fractal = ambient::fbm,
  frequency = 1,
  octaves = 2L,
  seed = 1L
)
```

Arguments

noise	A noise-generating function, passed to <code>ambient::fracture()</code> 's noise argument (e.g. <code>ambient::gen_simplex()</code> , <code>ambient::gen_perlin()</code> , <code>ambient::gen_worley()</code>). Default <code>ambient::gen_simplex()</code> .
fractal	A fractal combinator function, passed to <code>ambient::fracture()</code> 's fractal argument (e.g. <code>ambient::fbm()</code> , <code>ambient::billow()</code>). Default <code>ambient::fbm()</code> .
frequency	Noise frequency. Must be non-negative. Default 1.
octaves	Number of noise octaves. Must be a positive integer. Default 2L.
seed	Integer seed for the noise field. Default 1L.

Details

`noise_sample()` evaluates a noise_field at a set of positions and rescales the result into a target range – the operation shared by `shape_blob()`'s radius perturbation and `shape_ribbon()`/`shape_twist()`'s width modulation, factored out here so it isn't duplicated across those three constructors, and so the noise/fractal functions themselves are configurable rather than hardcoded to `ambient::gen_simplex()`/`ambient::fbm()`.

See Also

Other noise helpers: `noise_bridge()`, `noise_sample()`

Examples

```

noise_field(frequency = 2, octaves = 3, seed = 4821)

# a noise_field's effect is easiest to see through a drawable that
# samples it -- lower frequency gives broad, gentle undulation, higher
# frequency a bumpier, more textured one
draw(shape_blob(
  radius = 1,
  range = 0.4,
  distortion = noise_field(frequency = 0.5, seed = 4821)
))
draw(shape_blob(
  radius = 1,
  range = 0.4,
  distortion = noise_field(frequency = 6, seed = 4821)
))

# more octaves layer finer detail on top of the base frequency
draw(shape_blob(
  radius = 1,
  range = 0.4,
  distortion = noise_field(octaves = 6, seed = 4821)
))

```

noise_sample

*Sample a noise object***Description**

noise_sample() evaluates a noise object at a set of positions and returns the (typically rescaled) sampled values.

Usage

```
noise_sample(field, ...)
```

Arguments

field	A noise object, e.g. one built by noise_field() .
...	Passed to the method for field's class.

Details

It is an S7 generic dispatching on field; each concrete noise class implements its own method, since what "a position" means differs by class – a [noise_field](#) is sampled at arbitrary (x, y) coordinates in the plane, matching [ambient::fracture\(\)](#)'s own interface, while a [noise_bridge](#) instead samples by point count alone, with no (x, y) positions involved.

Value

A numeric vector.

See Also

Other noise helpers: `noise_bridge()`, `noise_field()`

Examples

```
noise_sample(noise_field(seed = 4821), x = 1:5, y = 1:5, to = c(0, 1))

# noise_bridge()'s method samples by point count instead of position
noise_sample(noise_bridge(seed = 4821), n = 5, scale = 1)

# sampled values can drive a drawable's own geometry, e.g. shape_blob()'s
# radius perturbation (see its `points` getter)
angle <- seq(0, 2 * pi, length.out = 12)
noise_sample(
  noise_field(seed = 4821),
  x = cos(angle),
  y = sin(angle),
  to = c(0.8, 1.2)
)
```

palette_cosine

Colour palette from a linear cosine formula

Description

Generates a smoothly varying colour palette using the linear cosine formula described by Iñigo Quilez for procedural palettes (see https://blog.djnavarro.net/posts/2025-09-14_cosine-palettes/):

$$f(t) = a + b \cos(2\pi(ct + d))$$

where $a = (0.5, 0.5, 0.5)$ is fixed and b, c, d are each an RGB triple sampled (with replacement) from base.

Usage

```
palette_cosine(n, base = NULL, seed = 1L)
```

Arguments

n	A single positive whole number: the number of colours to generate.
base	NULL (the default) or a character vector of candidate colours to sample b, c, and d from. NULL uses <code>grDevices::colors(distinct = TRUE)</code> .
seed	A single whole number seeding the random sampling of b, c, and d, so the same seed always produces the same palette.

Details

As in the source algorithm, resulting values are clamped only on the high end (values above 1 are capped at 1); values below 0 are folded back with `abs()` rather than clamped to 0. This is an intentional quirk inherited from the source algorithm, not a bug – it occasionally produces a colour "reflected" around black rather than a flat black.

Value

A character vector of `n` hex colour strings.

See Also

Other palette helpers: `palette_gradient()`, `palette_harmony()`, `palette_manual()`, `palette_tonal()`, `palette_walk()`

Examples

```
palette_cosine(n = 16, seed = 11)
draw(effect_swatch(color = palette_cosine(n = 16, seed = 11)))
```

palette_gradient	<i>Colour palette interpolated between user-supplied colours</i>
------------------	--

Description

Linearly interpolates `n` colours through the sequence of `color`, via `grDevices::colorRampPalette()`. Unlike `palette_manual()` (which interpolates a single curated 5-colour palette), `color` here is supplied directly by the caller – any number of colours, in any order.

Usage

```
palette_gradient(color, n, space = c("Lab", "rgb"))
```

Arguments

<code>color</code>	A character vector of at least two colours to interpolate through, in order.
<code>n</code>	A single positive whole number: the number of colours to return.
<code>space</code>	The colour space interpolation happens in, forwarded to <code>grDevices::colorRampPalette()</code> 's own space argument: "Lab" (the default) blends perceptually more evenly than "rgb", particularly for color pairs far apart in hue.

Value

A character vector of `n` hex colour strings.

See Also

Other palette helpers: [palette_cosine\(\)](#), [palette_harmony\(\)](#), [palette_manual\(\)](#), [palette_tonal\(\)](#), [palette_walk\(\)](#)

Examples

```
palette_gradient(c("midnightblue", "orange"), n = 8)
palette_gradient(c("black", "tomato", "white"), n = 12)
draw(effect_swatch(color = palette_gradient(c("midnightblue", "orange"), n = 8)))
```

palette_harmony	<i>Colour palette from hue-rotation colour-wheel harmonies</i>
-----------------	--

Description

Generates a palette by rotating base's own hue around the colour wheel, at fixed saturation and value, following standard colour-theory harmonies.

Usage

```
palette_harmony(
  base,
  scheme = c("complementary", "analogous", "triadic", "tetradic", "split_complementary"),
  n = 5L,
  spread = 30
)
```

Arguments

base	A single colour, used as the palette's starting hue (saturation and value are held fixed and shared by every returned colour).
scheme	One of "complementary" (base plus its hue rotated 180 degrees; always 2 colours), "triadic" (base plus hues rotated 120/240 degrees; always 3 colours), "tetradic" (base plus hues rotated 90/180/270 degrees; always 4 colours), "split_complementary" (base plus two hues either side of its complement, offset by spread; always 3 colours), or "analogous" (n hues centred on base, evenly spaced spread degrees apart).
n	A single positive whole number: the number of colours to return. Only used when scheme = "analogous" – every other scheme returns its own fixed colour count regardless of n.
spread	A single number (degrees): for scheme = "analogous", the spacing between adjacent hues; for scheme = "split_complementary", the offset of each returned hue from the exact complement. Unused (and silently ignored) for every other scheme.

Value

A character vector of hex colour strings.

See Also

Other palette helpers: [palette_cosine\(\)](#), [palette_gradient\(\)](#), [palette_manual\(\)](#), [palette_tonal\(\)](#), [palette_walk\(\)](#)

Examples

```
palette_harmony("steelblue", scheme = "triadic")
palette_harmony("steelblue", scheme = "analogous", n = 5, spread = 20)
draw(effect_swatch(color = palette_harmony("steelblue", scheme = "triadic")))
```

palette_manual	<i>Colour palette drawn from a curated collection</i>
----------------	---

Description

Selects one palette from a vendored, deduplicated copy of the manually curated palettes at <https://github.com/djnavarro/palettes>, each originally a 5-colour hex palette. If `n` differs from the palette's own native length, colours are linearly interpolated to `n` colours via [grDevices::colorRampPalette\(\)](#).

Usage

```
palette_manual(n = NULL, index = 1L)
```

Arguments

<code>n</code>	NULL (the default) or a single positive whole number: the number of colours to return. NULL returns the selected palette's own colours unchanged; otherwise the palette is interpolated to <code>n</code> colours.
<code>index</code>	A single positive whole number selecting which stored palette to use. An out-of-range index errors with the valid range for the currently vendored data.

Value

A character vector of `n` hex colour strings (or the selected palette's own colours, if `n` is NULL).

See Also

Other palette helpers: [palette_cosine\(\)](#), [palette_gradient\(\)](#), [palette_harmony\(\)](#), [palette_tonal\(\)](#), [palette_walk\(\)](#)

Examples

```
palette_manual(index = 1)
palette_manual(n = 12, index = 1)
draw(effect_swatch(color = palette_manual(index = 1)))
```

palette_tonal

*Colour palette from a monochromatic tint/shade ramp***Description**

Generates *n* colours sharing base's own hue and saturation, swept linearly through HSL lightness from `lightness[1]` to `lightness[2]` – a monochromatic ramp from near-black, through base's own hue, to near-white (or whatever narrower lightness range is supplied).

Usage

```
palette_tonal(base, n = 5L, lightness = c(0.1, 0.9))
```

Arguments

<code>base</code>	A single colour, used as the palette's shared hue and saturation.
<code>n</code>	A single positive whole number: the number of colours to return.
<code>lightness</code>	A length-2 numeric vector, each between 0 and 1: the HSL lightness of the first and last returned colours. Default <code>c(0.1, 0.9)</code> avoids sweeping all the way to pure black/white.

Value

A character vector of *n* hex colour strings.

See Also

Other palette helpers: [palette_cosine\(\)](#), [palette_gradient\(\)](#), [palette_harmony\(\)](#), [palette_manual\(\)](#), [palette_walk\(\)](#)

Examples

```
palette_tonal("darkorchid", n = 6)
palette_tonal("darkorchid", n = 6, lightness = c(0.3, 0.7))
draw(effect_swatch(color = palette_tonal("darkorchid", n = 6)))
```

palette_walk*Colour palette from a bounded random walk through colour space*

Description

Generates *n* colours by taking successive random steps through RGB space, starting at *base*, reflecting back into range at the $[0, 1]$ boundary of each channel rather than clamping – so a channel that would overshoot bounces back instead of flattening out at the boundary for the rest of the walk. This gives a sequence of related but organically drifting colours, in contrast to [palette_harmony\(\)](#)'s fixed geometric relationships or [palette_cosine\(\)](#)'s periodic formula.

Usage

```
palette_walk(n, base = NULL, step = 0.08, seed = 1L)
```

Arguments

<i>n</i>	A single positive whole number: the number of colours to generate.
<i>base</i>	NULL (the default) or a single colour to start the walk at. NULL starts from a uniform-random colour instead.
<i>step</i>	A single positive number controlling the standard deviation of each channel's random step between successive colours – larger values wander further/faster.
<i>seed</i>	A single whole number seeding the random walk, so the same seed always produces the same palette.

Value

A character vector of *n* hex colour strings.

See Also

Other palette helpers: [palette_cosine\(\)](#), [palette_gradient\(\)](#), [palette_harmony\(\)](#), [palette_manual\(\)](#), [palette_tonal\(\)](#)

Examples

```
palette_walk(n = 10, base = "seagreen", seed = 4187)
palette_walk(n = 10, step = 0.2, seed = 4187)
draw(effect_swatch(color = palette_walk(n = 10, base = "seagreen", seed = 4187)))
```

points_raw

A scatter of points defined directly by their coordinates

Description

points_raw is [shape_raw](#)'s "points"-geometry analog, and the first concrete constructor to use geometry = "points" (previously reserved on the dimensional reading "points"(0D)/"path"(1D)/"polygon"(2D), but with no constructor exposing it – see [.agents/PLAN.md](#)). The user supplies x/y coordinates directly, rendered as unconnected markers rather than a connected outline or path.

points_raws() is a vectorized version of [points_raw\(\)](#). Since x/y are themselves numeric vectors of point coordinates for a single scatter, points_raws() takes them as a list() of numeric vectors instead – one vector per scatter – which is most useful for giving several distinct scatters different style arguments (e.g. a different color each).

Usage

```
points_raw(x, y, id = NULL, trans = trans_identity(), ...)
```

```
points_raws(x, y, id = NULL, trans = trans_identity(), ...)
```

Arguments

x, y	For points_raw(), a numeric vector of x/y coordinates. For points_raws(), a list() of such vectors instead – one vector per scatter.
id	For points_raw(), an integer vector the same length as x/y, or NULL (the default). For points_raws(), a list() of such vectors (or NULLs) instead, one per scatter. Inert either way – see Details.
trans	A trans object applied to the computed points. Default trans_identity() (no transform).
...	Arguments passed to style() .

Details

style@fill and every line-related style property (linewidth, linetype, linejoin, lineend, linemitre) have no effect for points_raw() – see [drawable](#)'s geometry documentation and geometry_grob()'s internal dispatch (R/draw.R) for why a "points" geometry has no line to stroke and no interior to fill. Only style@color is used, as the marker colour.

id (see [xy](#)'s own id) is accepted and stored for consistency with [shape_raw\(\)](#)/[curve_raw\(\)](#) and to round-trip through [convert\(\)](#), but is otherwise inert here – grid::pointsGrob() has no sub-path concept, consistent with fill already being inert for "points" geometry.

Every other argument may be a plain vector, recycled against x/y via purrr::pmap()'s own vctrs-based rules (any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error). The result is a [sketch](#) containing one points_raw() per list element/recycled row, rather than a single drawable.

Value

A [drawable](#).

For `points_raws()`, a [sketch](#).

Examples

```
draw(points_raw(
  x = seq(0, 1, length.out = 20),
  y = sin(seq(0, 2 * pi, length.out = 20)) / 2 + 0.5,
  color = "steelblue"
))

# a random scatter, and the same points converted from a polygon's
# own outline (only style-related properties survive the round trip)
draw(points_raw(x = runif(200), y = runif(200)))
draw(S7::convert(
  shape_blob(radius = 1, distortion = noise_field(seed = 42)),
  points_raw
))

# points_raw() is pathlike, so effect_tremor() can wobble the scatter
draw(effect_tremor(
  points_raw(x = seq(0, 1, length.out = 30), y = rep(0.5, 30)),
  layers = 6L, jitter = 0.08
))

draw(points_raws(
  x = list(seq(0, 1, length.out = 10), seq(0, 1, length.out = 10)),
  y = list(rep(0.25, 10), rep(0.75, 10)),
  color = c("steelblue", "darkred")
))
```

 save_png

Save a drawable or sketch to an image file

Description

`save_png()`, `save_svg()`, and `save_pdf()` render a [drawable](#) or [sketch](#) straight to a raster (PNG) or vector (SVG/PDF) file, without the caller needing to manage a [grDevices](#) device by hand. Each is a thin wrapper: open the appropriate device, call `draw()`, and always close the device afterward (via `on.exit()`, so a device is never left open even if `draw()` itself errors).

Usage

```
save_png(
  object,
  filename,
```

```

    width = 7,
    height = 7,
    dpi = 300,
    bg = "white",
    units = "in",
    ...
)

save_svg(object, filename, width = 7, height = 7, bg = "white", ...)

save_pdf(object, filename, width = 7, height = 7, bg = "white", ...)

```

Arguments

object	A drawable or sketch object.
filename	A single string, the path to write to.
width, height	Image dimensions, in units for <code>save_png()</code> (default inches) or always in inches for <code>save_svg()/save_pdf()</code> , which have no <code>units</code> argument of their own. Default 7.
dpi	Resolution in dots per inch. Only meaningful for <code>save_png()</code> (a raster format); ignored by <code>save_svg()/save_pdf()</code> , which are drawn at vector resolution. Default 300.
bg	Background colour passed to the underlying device, e.g. "white" (the default) or "transparent". This is the device's own page colour, independent of any canvas() background a sketch itself already carries – the two compose, so a transparent bg here still shows an opaque <code>canvas()</code> background underneath, and vice versa a sketch with no <code>canvas()</code> background shows bg through any of its own shapes that don't fully cover the page.
units	One of "in" (the default), "cm", "mm", or "px", forwarded to grDevices::png() 's own <code>units</code> argument – only meaningful for <code>save_png()</code> ; <code>save_svg()/save_pdf()</code> have no <code>units</code> argument, since grDevices::svg()/grDevices::pdf() always take width/height in inches.
...	Passed on to draw() , e.g. <code>xlim/ylim</code> .

Value

filename, invisibly.

Examples

```

file <- tempfile(fileext = ".png")
save_png(shape_circle(radius = 1), file)

file <- tempfile(fileext = ".svg")
save_svg(shape_circle(radius = 1), file)

file <- tempfile(fileext = ".pdf")
save_pdf(shape_circle(radius = 1), file)

```

shape_bezier	<i>A closed Bezier curve</i>
--------------	------------------------------

Description

shape_bezier is a [drawable](#) whose outline follows a Bezier curve defined by an arbitrary number of control points (x, y). With two control points this is a straight line; with four, a cubic Bezier of the kind used to build ribbons and other flowing shapes.

shape_beziers() is a vectorized version of [shape_bezier\(\)](#). Since x/y are themselves numeric vectors of control points for a single curve, shape_beziers() takes them as a `list()` of numeric vectors instead – one vector of control points per shape – rather than a bare vector (which shape_circles()-style constructors use for a plain per-shape scalar).

Usage

```
shape_bezier(x, y, n = 100L, trans = trans_identity(), ...)
```

```
shape_beziers(x, y, n = 100L, trans = trans_identity(), ...)
```

Arguments

x, y	For shape_bezier(), numeric vectors of control point coordinates, the same length, with at least two control points. For shape_beziers(), a <code>list()</code> of such vectors instead – one vector of control points per shape.
n	Number of points used to sample the curve. Default 100L.
trans	A trans object applied to the shape's computed points. Default trans_identity() (no transform).
...	Arguments passed to style() .

Details

Since [draw\(\)](#) renders every "polygon"-geometry drawable's points as a closed polygon, the curve is implicitly closed from its last control point back to its first – for an open Bezier curve/path instead, see [curve_bezier\(\)](#).

Every other argument may be a plain vector, recycled against x/y via `purrr::pmap()`'s own vctrs-based rules (any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error). The result is a [sketch](#) containing one [shape_bezier\(\)](#) per list element/recycled row, rather than a single drawable.

Value

A [drawable](#).

For shape_beziers(), a [sketch](#).

See Also

Other 2D shapes: [shape_blob\(\)](#), [shape_circle\(\)](#), [shape_combine\(\)](#), [shape_ellipse\(\)](#), [shape_polygon\(\)](#), [shape_raw\(\)](#), [shape_rectangle\(\)](#), [shape_ribbon\(\)](#), [shape_star\(\)](#), [shape_stroke\(\)](#), [shape_strokepath\(\)](#), [shape_twist\(\)](#), [shape_wedge\(\)](#)

Examples

```
draw(shape_bezier(x = c(0, 0.5, 1, 0.5), y = c(0, 1, 0, -1)))

# two control points collapse the curve to a straight-edged polygon
draw(shape_bezier(x = c(0, 1), y = c(0, 1), fill = "steelblue"))

# more control points give the curve more inflections
draw(shape_bezier(
  x = c(0, 0.3, 0.6, 1, 0.6, 0.3), y = c(0, 1, -1, 0, 1, -1),
  fill = fill_hatch(angle = 60, spacing = 0.08)
))

draw(shape_beziers(
  x = list(c(0, 0.5, 1, 0.5), c(2, 2.5, 3, 2.5)),
  y = list(c(0, 1, 0, -1), c(0, 1, 0, -1))
))
```

 shape_blob

An irregular, "blobby" circle

Description

shape_blob is a [drawable](#) similar to [shape_circle](#), except that its radius varies smoothly around the circumference according to Perlin/simplex noise generated with **ambient**.

shape_blobs() is a vectorized version of [shape_blob\(\)](#): each argument may be a vector, recycled against the others via `purrr::pmap()`'s own `vctrs`-based rules. The result is a [sketch](#) containing one [shape_blob\(\)](#) per recycled row, rather than a single [drawable](#).

Usage

```
shape_blob(
  x = 0,
  y = 0,
  radius = 1,
  range = 0.2,
  n = 100L,
  distortion = noise_field(),
  trans = trans_identity(),
  ...
)
```

```

shape_blobs(
    x = 0,
    y = 0,
    radius = 1,
    range = 0.2,
    n = 100L,
    distortion = noise_field(),
    trans = trans_identity(),
    ...
)

```

Arguments

<code>x, y</code>	Centroid coordinates. Default 0.
<code>radius</code>	Mean radius. Must be non-negative. Default 1.
<code>range</code>	Amplitude of the radius distortion. Must be non-negative. Default 0.2.
<code>n</code>	Number of points used to approximate the outline. Default 100L.
<code>distortion</code>	A noise_field controlling the radius distortion. Default <code>noise_field()</code> .
<code>trans</code>	A trans object applied to the shape's computed points. Default <code>trans_identity()</code> (no transform).
<code>...</code>	Arguments passed to style() .

Details

Any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error. A shared distortion [noise_field](#) is automatically recycled across every blob; pass a `list()` of several different `noise_fields` instead to vary it per blob.

Value

A [drawable](#).

For `shape_blobs()`, a [sketch](#).

See Also

Other 2D shapes: [shape_bezier\(\)](#), [shape_circle\(\)](#), [shape_combine\(\)](#), [shape_ellipse\(\)](#), [shape_polygon\(\)](#), [shape_raw\(\)](#), [shape_rectangle\(\)](#), [shape_ribbon\(\)](#), [shape_star\(\)](#), [shape_stroke\(\)](#), [shape_strokepath\(\)](#), [shape_twist\(\)](#), [shape_wedge\(\)](#)

Examples

```

draw(shape_blob(
  radius = 1,
  range = 0.3,
  distortion = noise_field(seed = 4821)
))

```

a larger range or lower frequency distortion gives a more irregular

```

# outline; a higher frequency gives a bumpier, lumpier one
draw(shape_blob(
  radius = 1,
  range = 0.6,
  distortion = noise_field(seed = 4821)
))
draw(shape_blob(
  radius = 1, range = 0.3,
  distortion = noise_field(frequency = 5, seed = 4821)
))

draw(shape_blob(
  radius = 1,
  range = 0.3,
  fill = fill_charcoal(),
  color = NA_character_
))

draw(shape_blobs(x = 1:3, radius = c(0.5, 1, 1.5), range = 0.2))

# a shared distortion recycles across every blob; a list() of several
# different noise_field()s varies it per blob instead
draw(shape_blobs(
  x = 1:3, radius = 0.8, range = 0.3,
  distortion = list(
    noise_field(seed = 1),
    noise_field(seed = 2),
    noise_field(seed = 3)
  )
))

```

shape_circle

A circle

Description

shape_circle is a [drawable](#) defined by a centroid and a radius; its vertices are computed as evenly spaced points around the circumference.

shape_circles() is a vectorized version of [shape_circle\(\)](#): each argument may be a vector, recycled against the others. The result is a [sketch](#) containing one shape_circle() per recycled row, rather than a single drawable.

Usage

```
shape_circle(x = 0, y = 0, radius = 1, n = 100L, trans = trans_identity(), ...)
```

```
shape_circles(
  x = 0,
```

```

    y = 0,
    radius = 1,
    n = 100L,
    trans = trans_identity(),
    ...
  )

```

Arguments

x, y	Centroid coordinates. Default 0.
radius	Radius. Must be non-negative. Default 1.
n	Number of points used to approximate the circle. Default 100L.
trans	A trans object applied to the shape's computed points. Default trans_identity() (no transform).
...	Arguments passed to style() .

Details

Recycling uses `purrr::pmap()`'s own vctrs-based rules: any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error.

Value

A [drawable](#).

For `shape_circles()`, a [sketch](#).

See Also

Other 2D shapes: [shape_bezier\(\)](#), [shape_blob\(\)](#), [shape_combine\(\)](#), [shape_ellipse\(\)](#), [shape_polygon\(\)](#), [shape_raw\(\)](#), [shape_rectangle\(\)](#), [shape_ribbon\(\)](#), [shape_star\(\)](#), [shape_stroke\(\)](#), [shape_strokepath\(\)](#), [shape_twist\(\)](#), [shape_wedge\(\)](#)

Examples

```

draw(shape_circle(radius = 1))
draw(shape_circle(x = 1, y = 1, radius = 0.5, n = 6, color = "darkred"))
draw(shape_circle(radius = 1, fill = "steelblue", color = NA_character_))
draw(shape_circle(radius = 1, fill = fill_hatch(angle = 30)))

draw(shape_circles(x = 1:3, radius = c(0.5, 1, 1.5)))

# each argument recycles independently, so colour can vary alongside position
draw(shape_circles(
  x = cos(seq(0, 2 * pi, length.out = 9))[-9],
  y = sin(seq(0, 2 * pi, length.out = 9))[-9],
  radius = 0.3,
  fill = rep(c("tomato", "steelblue"), length.out = 8)
))

```

shape_combine

Combine several drawables into one multi-sub-path shape

Description

shape_combine() merges the computed points of several polygon-geometry drawables into a single [shape_raw](#), one sub-path per input (see [xy](#)'s own id). This is the ergonomic entry point for the two motivating multi-sub-path use cases – a shape with a hole (one input nested inside another, under [style\(\)](#)'s default rule = "evenodd"), or several disjoint shapes sharing one [style](#) – without computing id by hand the way [shape_raw\(\)](#)'s own id argument requires.

Usage

```
shape_combine(..., style = NULL)
```

Arguments

...	Two or more polygon-geometry drawable objects (@geometry == "polygon", e.g. shape_circle() , shape_blob()).
style	A style object for the combined shape, or NULL (the default) to reuse the first input's own style. Every input's own style besides the one that's kept is discarded – a drawable has exactly one style; several independently-styled shapes should use a sketch instead.

Details

Each input's own already-computed points (i.e. trans/noise-based distortion already applied, the same way [convert\(\)](#) "bakes in" a drawable's transform) becomes one or more sub-paths in the combined output – an input that already has multiple sub-paths of its own (e.g. the output of an earlier shape_combine() call) keeps them all, renumbered to stay distinct from every other input's own sub-paths. Whether nested sub-paths render as a hole or a second solid region is purely a function of their geometric nesting and style@rule (see [style\(\)](#)'s own docs) – not of argument order, and not of anything shape_combine() itself decides.

Value

A [shape_raw](#).

See Also

Other 2D shapes: [shape_bezier\(\)](#), [shape_blob\(\)](#), [shape_circle\(\)](#), [shape_ellipse\(\)](#), [shape_polygon\(\)](#), [shape_raw\(\)](#), [shape_rectangle\(\)](#), [shape_ribbon\(\)](#), [shape_star\(\)](#), [shape_stroke\(\)](#), [shape_strokepath\(\)](#), [shape_twist\(\)](#), [shape_wedge\(\)](#)

Examples

```
# a ring with a hole: a smaller circle nested inside a larger one
draw(shape_combine(
  shape_circle(radius = 2),
  shape_circle(radius = 1),
  style = style(fill = "steelblue")
))

# several disjoint blobs sharing one style
draw(shape_combine(
  shape_blob(x = 0, distortion = noise_field(seed = 1L)),
  shape_blob(x = 3, distortion = noise_field(seed = 2L)),
  shape_blob(x = 6, distortion = noise_field(seed = 3L)),
  style = style(fill = "goldenrod")
))

# combining a hole with a disjoint extra shape in one call
draw(shape_combine(
  shape_circle(x = 0, radius = 2),
  shape_circle(x = 0, radius = 1),
  shape_circle(x = 4, radius = 0.5)
))
```

shape_ellipse

An ellipse

Description

shape_ellipse is a [drawable](#) defined by a centroid and independent x/y radii; its vertices are computed as evenly spaced points around the circumference, generalizing [shape_circle\(\)](#) (which is a shape_ellipse with equal radii in both directions).

shape_ellipses() is a vectorized version of [shape_ellipse\(\)](#): each argument may be a vector, recycled against the others. The result is a [sketch](#) containing one shape_ellipse() per recycled row, rather than a single drawable.

Usage

```
shape_ellipse(
  x = 0,
  y = 0,
  x_radius = 1,
  y_radius = 1,
  n = 100L,
  trans = trans_identity(),
  ...
)
```

```

shape_ellipses(
  x = 0,
  y = 0,
  x_radius = 1,
  y_radius = 1,
  n = 100L,
  trans = trans_identity(),
  ...
)

```

Arguments

<code>x, y</code>	Centroid coordinates. Default 0.
<code>x_radius, y_radius</code>	Radii along the x and y axes. Must be non-negative. Default 1.
<code>n</code>	Number of points used to approximate the ellipse. Default 100L.
<code>trans</code>	A trans object applied to the shape's computed points. Default trans_identity() (no transform).
<code>...</code>	Arguments passed to style() .

Details

Recycling uses `purrr::pmap()`'s own vctrs-based rules: any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error.

Value

A [drawable](#).

For `shape_ellipses()`, a [sketch](#).

See Also

Other 2D shapes: [shape_bezier\(\)](#), [shape_blob\(\)](#), [shape_circle\(\)](#), [shape_combine\(\)](#), [shape_polygon\(\)](#), [shape_raw\(\)](#), [shape_rectangle\(\)](#), [shape_ribbon\(\)](#), [shape_star\(\)](#), [shape_stroke\(\)](#), [shape_strokepath\(\)](#), [shape_twist\(\)](#), [shape_wedge\(\)](#)

Examples

```

draw(shape_ellipse(x_radius = 2, y_radius = 1))
draw(shape_ellipse(
  x = 1,
  y = 1,
  x_radius = 0.5,
  y_radius = 1,
  n = 6,
  color = "darkred"
))
draw(shape_ellipse(
  x_radius = 2,

```

```

    y_radius = 1,
    fill = fill_gradient(angle = 90)
  ))

  # rotating an ellipse (rather than swapping its radii) tilts its axes
  draw(shape_ellipse(
    x_radius = 2,
    y_radius = 0.7,
    trans = trans_rotate(pi / 6)
  ))

  draw(shape_ellipses(x = 1:3, x_radius = c(0.5, 1, 1.5), y_radius = 0.5))
  draw(shape_ellipses(
    x = 0, y = 0, x_radius = 1, y_radius = 0.3,
    trans = purrr::map(seq(0, pi, length.out = 6), trans_rotate)
  ))

```

shape_polygon

A regular polygon

Description

shape_polygon is a [drawable](#) defined by a centroid and a radius; its vertices are n evenly spaced points around the circumference, giving a regular n -gon (e.g. $n = 3$ a triangle, $n = 6$ a hexagon).

shape_polygons() is a vectorized version of [shape_polygon\(\)](#): each argument may be a vector, recycled against the others. The result is a [sketch](#) containing one shape_polygon() per recycled row, rather than a single drawable.

Usage

```
shape_polygon(x = 0, y = 0, radius = 1, n = 6L, trans = trans_identity(), ...)
```

```
shape_polygons(x = 0, y = 0, radius = 1, n = 6L, trans = trans_identity(), ...)
```

Arguments

x, y	Centroid coordinates. Default 0.
radius	Radius. Must be non-negative. Default 1.
n	Number of sides (vertices). Must be at least 3. Default 6L.
trans	A trans object applied to the shape's computed points. Default trans_identity() (no transform).
...	Arguments passed to style() .

Details

Unlike [shape_circle\(\)/shape_blob\(\)](#), [points](#) does not repeat its first vertex at the end – there are exactly n distinct vertices, since `grid::polygonGrob()` closes the path itself and a polygon has no approximation error to hide.

Recycling uses `purrr::pmap()`'s own vctrs-based rules: any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error.

Value

A [drawable](#).

For `shape_polygons()`, a [sketch](#).

See Also

Other 2D shapes: [shape_bezier\(\)](#), [shape_blob\(\)](#), [shape_circle\(\)](#), [shape_combine\(\)](#), [shape_ellipse\(\)](#), [shape_raw\(\)](#), [shape_rectangle\(\)](#), [shape_ribbon\(\)](#), [shape_star\(\)](#), [shape_stroke\(\)](#), [shape_strokepath\(\)](#), [shape_twist\(\)](#), [shape_wedge\(\)](#)

Examples

```
draw(shape_polygon(radius = 1, n = 3))
draw(shape_polygon(x = 1, y = 1, radius = 0.5, n = 8, color = "darkred"))
draw(shape_polygon(n = 5, fill = fill_stripe(color = c("white", "grey30"))))

# rotating a low-n polygon reorients its vertices
draw(shape_polygon(n = 4, trans = trans_rotate(pi / 4)))

draw(shape_polygons(x = 1:3, n = c(3L, 4L, 6L)))
draw(shape_polygons(
  x = 0, y = 0, radius = seq(3, 0.5, length.out = 6), n = 6,
  fill = rep(c("grey20", "grey90"), length.out = 6)
))
```

shape_raw

A polygon defined directly by its vertices

Description

`shape_raw` is the simplest [drawable](#): the user supplies x and y coordinates directly, and `points` is computed trivially from them. It is most often produced by [convert\(\)](#)ing a more complex drawable (e.g. a [shape_blob](#) or [shape_twist](#)) down to its raw vertices.

`shape_raws()` is a vectorized version of [shape_raw\(\)](#). Since x/y are themselves numeric vectors of vertex coordinates for a single polygon, `shape_raws()` takes them as a `list()` of numeric vectors instead – one vector of vertices per shape.

Usage

```
shape_raw(x, y, id = NULL, trans = trans_identity(), ...)
```

```
shape_raws(x, y, id = NULL, trans = trans_identity(), ...)
```

Arguments

<code>x, y</code>	For <code>shape_raw()</code> , a numeric vector of x/y coordinates. For <code>shape_raws()</code> , a <code>list()</code> of such vectors instead – one vector of vertices per shape.
<code>id</code>	For <code>shape_raw()</code> , an integer vector the same length as <code>x/y</code> , or <code>NULL</code> (the default, a single sub-path). For <code>shape_raws()</code> , a <code>list()</code> of such vectors (or <code>NULL</code> s) instead, one per shape.
<code>trans</code>	A trans object applied to the shape's computed points. Default <code>trans_identity()</code> (no transform).
<code>...</code>	Arguments passed to <code>style()</code> .

Details

`id` optionally groups `x/y` into multiple sub-paths (see [xy](#)'s own `id`) – several disjoint polygons sharing one [style](#), or, combined with `style()`'s rule, a shape with a hole. `shape_combine()` is a more convenient way to build this from several already-built drawables' own points, rather than computing `id` by hand.

Every other argument may be a plain vector, recycled against `x/y` via `purrr::pmap()`'s own `vctrs`-based rules (any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error). The result is a [sketch](#) containing one `shape_raw()` per list element/recycled row, rather than a single drawable.

Value

A [drawable](#).

For `shape_raws()`, a [sketch](#).

See Also

Other 2D shapes: [shape_bezier\(\)](#), [shape_blob\(\)](#), [shape_circle\(\)](#), [shape_combine\(\)](#), [shape_ellipse\(\)](#), [shape_polygon\(\)](#), [shape_rectangle\(\)](#), [shape_ribbon\(\)](#), [shape_star\(\)](#), [shape_stroke\(\)](#), [shape_strokepath\(\)](#), [shape_twist\(\)](#), [shape_wedge\(\)](#)

Examples

```
draw(shape_raw(x = c(0, 1, 1, 0), y = c(0, 0, 1, 1)))

# any polygon can be "frozen" to a shape_raw by converting it
frozen <- S7::convert(
  shape_blob(radius = 1, distortion = noise_field(seed = 5150)),
  shape_raw
)
draw(frozen)
```

```

draw(shape_raw(
  x = c(0, 1, 0.5), y = c(0, 0, 1),
  fill = "goldenrod", color = "black", linewidth = 2
))

# id groups x/y into sub-paths -- here, a ring with a hole (a smaller
# sub-path nested inside a larger one, under style()'s default
# rule = "evenodd")
draw(shape_raw(
  x = c(0, 0, 4, 4, 1, 1, 3, 3),
  y = c(0, 4, 4, 0, 1, 3, 3, 1),
  id = c(1, 1, 1, 1, 2, 2, 2, 2),
  fill = "steelblue"
))

draw(shape_raws(
  x = list(c(0, 1, 1, 0), c(2, 3, 3, 2)),
  y = list(c(0, 0, 1, 1), c(0, 0, 1, 1))
))

```

shape_rectangle	<i>A rectangle</i>
-----------------	--------------------

Description

shape_rectangle is a [drawable](#) defined by a centroid and a width/height, rendered as an axis-aligned rectangle.

shape_rectangles()/shape_squares() are vectorized versions of [shape_rectangle\(\)](#)/[shape_square\(\)](#): each argument may be a vector, recycled against the others. The result is a [sketch](#) containing one shape_rectangle()/shape_square() per recycled row, rather than a single drawable.

Usage

```

shape_rectangle(
  x = 0,
  y = 0,
  width = 1,
  height = 1,
  trans = trans_identity(),
  ...
)

shape_square(x = 0, y = 0, side = 1, trans = trans_identity(), ...)

shape_rectangles(
  x = 0,

```

```

    y = 0,
    width = 1,
    height = 1,
    trans = trans_identity(),
    ...
)

shape_squares(x = 0, y = 0, side = 1, trans = trans_identity(), ...)
```

Arguments

x, y	Centroid coordinates. Default 0.
width, height	Rectangle dimensions. Must be non-negative. Default 1.
trans	A trans object applied to the shape's computed points. Default trans_identity() (no transform).
...	Arguments passed to style() .
side	Side length, for shape_square() . Must be non-negative. Default 1.

Details

[shape_square\(\)](#) is a thin convenience wrapper around [shape_rectangle\(\)](#) for the common case of `width == height`, taking a single `side` argument instead – it returns a [shape_rectangle](#) object directly rather than being its own class.

Recycling uses `purrr::pmap()`'s own vctrs-based rules: any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error.

Value

A [drawable](#).

For [shape_rectangles\(\)](#)/[shape_squares\(\)](#), a [sketch](#).

See Also

Other 2D shapes: [shape_bezier\(\)](#), [shape_blob\(\)](#), [shape_circle\(\)](#), [shape_combine\(\)](#), [shape_ellipse\(\)](#), [shape_polygon\(\)](#), [shape_raw\(\)](#), [shape_ribbon\(\)](#), [shape_star\(\)](#), [shape_stroke\(\)](#), [shape_strokepath\(\)](#), [shape_twist\(\)](#), [shape_wedge\(\)](#)

Examples

```

draw(shape_rectangle(width = 2, height = 1))
draw(shape_square(x = 1, y = 1, side = 0.5, color = "darkred"))
draw(shape_rectangle(width = 2, height = 1, fill = fill_checker()))

# trans_rotate() tilts a rectangle about its own centroid by default
draw(shape_rectangle(width = 2, height = 0.6, trans = trans_rotate(pi / 8)))

draw(shape_rectangles(x = 1:3, width = c(0.5, 1, 1.5), height = 0.5))
draw(shape_squares(x = 1:3, side = c(0.5, 1, 1.5)))
```

```
# a checkerboard of squares, alternating fill colour
grid_xy <- expand.grid(x = 1:5, y = 1:3)
draw(shape_squares(
  x = grid_xy$x, y = grid_xy$y, side = 0.9,
  fill = c("grey20", "grey90")[(grid_xy$x + grid_xy$y) %% 2 + 1]
))
```

shape_ribbon

A tapered ribbon between two points

Description

shape_ribbon is a [drawable](#) polygon that follows a straight line between (x, y) and (xend, yend), with a width that tapers at both ends and varies along its length according to simplex noise.

shape_ribbons() is a vectorized version of [shape_ribbon\(\)](#): each argument may be a vector, recycled against the others via purrr::pmap()'s own vctrs-based rules. The result is a [sketch](#) containing one shape_ribbon() per recycled row, rather than a single drawable.

Usage

```
shape_ribbon(
  x = 0,
  y = 0,
  xend = 1,
  yend = 1,
  width = 0.2,
  n = 100L,
  distortion = noise_field(),
  trans = trans_identity(),
  ...
)
```

```
shape_ribbons(
  x = 0,
  y = 0,
  xend = 1,
  yend = 1,
  width = 0.2,
  n = 100L,
  distortion = noise_field(),
  trans = trans_identity(),
  ...
)
```

Arguments

<code>x, y</code>	Start point. Default 0.
<code>xend, yend</code>	End point. Default 1.
<code>width</code>	Maximum width. Must be non-negative. Default 0.2.
<code>n</code>	Number of points used along the path. Default 100L.
<code>distortion</code>	A noise_field controlling the width modulation. Default <code>noise_field()</code> .
<code>trans</code>	A trans object applied to the shape's computed points. Default <code>trans_identity()</code> (no transform).
<code>...</code>	Arguments passed to <code>style()</code> .

Details

Any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error. A shared distortion [noise_field](#) is automatically recycled across every ribbon; pass a `list()` of several different `noise_fields` instead to vary it per ribbon.

Value

A [drawable](#).

For `shape_ribbons()`, a [sketch](#).

See Also

Other 2D shapes: [shape_bezier\(\)](#), [shape_blob\(\)](#), [shape_circle\(\)](#), [shape_combine\(\)](#), [shape_ellipse\(\)](#), [shape_polygon\(\)](#), [shape_raw\(\)](#), [shape_rectangle\(\)](#), [shape_star\(\)](#), [shape_stroke\(\)](#), [shape_strokepath\(\)](#), [shape_twist\(\)](#), [shape_wedge\(\)](#)

Examples

```
draw(shape_ribbon(x = 0, y = 0, xend = 1, yend = 1, width = 0.3))

# a higher-frequency distortion modulates width more rapidly along the
# ribbon's length, giving a more textured/organic edge
draw(shape_ribbon(
  x = 0, y = 0, xend = 1, yend = 1, width = 0.3,
  distortion = noise_field(frequency = 8, seed = 99)
))

draw(shape_ribbon(
  x = 0, y = 0, xend = 2, yend = 0, width = 0.4,
  fill = fill_gradient(color = c("steelblue", "white"))
))

draw(shape_ribbons(x = 1:3, y = 0, xend = 2:4, yend = 1, width = 0.3))

# a fan of ribbons radiating from the origin
angle <- seq(0, 2 * pi, length.out = 13)[-13]
draw(shape_ribbons(
```

```

    x = 0,
    y = 0,
    xend = cos(angle),
    yend = sin(angle),
    width = 0.1
  ))

```

 shape_star

A star

Description

shape_star is a [drawable](#) defined by a centroid and two radii: its outline alternates n outer vertices (at outer_radius) with n inner vertices (at inner_radius), evenly spaced by angle, giving the familiar n -pointed star shape (e.g. $n = 5$ a five-pointed star, $n = 6$ a Star-of-David-like hexagram outline).

shape_stars() is a vectorized version of [shape_star\(\)](#): each argument may be a vector, recycled against the others. The result is a [sketch](#) containing one shape_star() per recycled row, rather than a single drawable.

Usage

```

shape_star(
  x = 0,
  y = 0,
  outer_radius = 1,
  inner_radius = 0.5,
  n = 5L,
  trans = trans_identity(),
  ...
)

```

```

shape_stars(
  x = 0,
  y = 0,
  outer_radius = 1,
  inner_radius = 0.5,
  n = 5L,
  trans = trans_identity(),
  ...
)

```

Arguments

x, y	Centroid coordinates. Default 0.
outer_radius	Outer radius (star tips). Must be non-negative. Default 1.

inner_radius	Inner radius (between tips). Must be non-negative and no greater than outer_radius. Default 0.5.
n	Number of star points. Must be at least 2. Default 5L.
trans	A trans object applied to the shape's computed points. Default trans_identity() (no transform).
...	Arguments passed to style() .

Details

Like [shape_polygon\(\)](#), points does not repeat its first vertex at the end – there are exactly $2 * n$ distinct vertices, since `grid::polygonGrob()` closes the path itself. The first vertex is an outer one at angle 0; use [trans_rotate\(\)](#) to reorient the star, the same way [shape_polygon\(\)](#)'s own low-n vertices are reoriented.

`inner_radius = outer_radius` degenerates the star into a regular $2 * n$ -gon (no points), and `inner_radius = 0` collapses every inner vertex onto the centroid, giving a sharp n-pointed "asterisk" outline.

Recycling uses `purrr::pmap()`'s own vctrs-based rules: any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error.

Value

A [drawable](#).

For `shape_stars()`, a [sketch](#).

See Also

Other 2D shapes: [shape_bezier\(\)](#), [shape_blob\(\)](#), [shape_circle\(\)](#), [shape_combine\(\)](#), [shape_ellipse\(\)](#), [shape_polygon\(\)](#), [shape_raw\(\)](#), [shape_rectangle\(\)](#), [shape_ribbon\(\)](#), [shape_stroke\(\)](#), [shape_strokepath\(\)](#), [shape_twist\(\)](#), [shape_wedge\(\)](#)

Examples

```
draw(shape_star())
draw(shape_star(x = 1, y = 1, outer_radius = 0.5, n = 8, color = "darkred"))
draw(shape_star(n = 6, fill = "goldenrod"))

# a sharper star: inner_radius closer to 0
draw(shape_star(inner_radius = 0.2, fill = "steelblue"))

# rotating a star reorients its tips
draw(shape_star(n = 5, trans = trans_rotate(pi / 2)))

draw(shape_stars(x = 1:3, n = c(4L, 5L, 6L)))
draw(shape_stars(
  x = 0, y = 0, outer_radius = seq(3, 0.5, length.out = 6), n = 5,
  fill = rep(c("grey20", "grey90"), length.out = 6)
))
```

shape_stroke

A tapered, pressure-modulated stroke along an arbitrary path

Description

shape_stroke is a [drawable](#) polygon that follows an arbitrary open path through (x, y) control points, offset into a ribbon whose width tapers to zero at both ends and varies along its length according to a [noise_field](#) – intended as a "pressure" curve, giving the outline an ink/brush-stroke look rather than a constant-width line.

shape_strokes() is a vectorized version of [shape_stroke\(\)](#). Since x/y are themselves numeric vectors of control points for a single stroke, shape_strokes() takes them as a `list()` of numeric vectors instead – one vector of control points per stroke.

Usage

```
shape_stroke(
  x,
  y,
  width = 0.2,
  n = 100L,
  distortion = noise_field(),
  trans = trans_identity(),
  ...
)

shape_strokes(
  x,
  y,
  width = 0.2,
  n = 100L,
  distortion = noise_field(),
  trans = trans_identity(),
  ...
)
```

Arguments

x, y	For shape_stroke(), numeric vectors of control point coordinates, the same length, with at least two control points. For shape_strokes(), a <code>list()</code> of such vectors instead – one vector of control points per stroke.
width	Maximum width. Must be non-negative. Default 0.2.
n	Number of points used along the resampled path. Must be at least 2. Default 100L.
distortion	A noise_field controlling the width ("pressure") modulation. Default <code>noise_field()</code> .
trans	A trans object applied to the shape's computed points. Default <code>trans_identity()</code> (no transform).

... Arguments passed to `style()`.

Details

The path is resampled to n evenly arc-length-spaced points (unlike `curve_line()`'s exact control-point vertices). This generalizes `shape_ribbon()` from a single straight segment to any path, at the cost of computing a true per-point unit normal (via the internal `stroke_normals()` helper) rather than `shape_ribbon()/shape_twist()`'s shared single offset direction – necessary once the path can genuinely curve, not just wander slightly off straight.

Unlike `shape_ribbon()`'s own taper (which peaks at 0.5, an undocumented quirk of its $\sqrt{t * (1 - t)}$ formula), `shape_stroke`'s taper is normalized to peak at 1 at the path's midpoint, so width is exactly the maximum rendered width.

Resampling only redistributes points evenly along the control polyline's own straight segments – it does not smooth or curve-fit sharp corners, matching `curve_line()`'s own no-smoothing convention. A `shape_stroke()` built from only a few widely-spaced control points will still have visibly angular corners; supply a denser x/y (e.g. points already sampled from some smooth function) for a smoothly curving stroke.

Every other argument may be a plain vector, recycled against x/y via `purrr::pmap()`'s own `vctr`-based rules (any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error). A shared distortion `noise_field` is automatically recycled across every stroke; pass a `list()` of several different `noise_fields` instead to vary it per stroke. The result is a `sketch` containing one `shape_stroke()` per list element/recycled row, rather than a single drawable.

Value

A `drawable`.

For `shape_strokes()`, a `sketch`.

See Also

Other 2D shapes: `shape_bezier()`, `shape_blob()`, `shape_circle()`, `shape_combine()`, `shape_ellipse()`, `shape_polygon()`, `shape_raw()`, `shape_rectangle()`, `shape_ribbon()`, `shape_star()`, `shape_strokepath()`, `shape_twist()`, `shape_wedge()`

Examples

```
draw(shape_stroke(x = c(0, 1, 2, 3), y = c(0, 1, 0, 1), width = 0.3))
draw(shape_stroke(
  x = c(0, 1, 2, 3), y = c(0, 1, 0, 1), width = 0.3,
  distortion = noise_field(frequency = 3, seed = 7734)
))
```

```
# a few widely-spaced control points give angular corners, since
# resampling redistributes points but never smooths them
draw(shape_stroke(x = c(0, 1, 0), y = c(0, 2, 0), width = 0.2))
```

```
# denser input points (already sampled from a smooth function) give a
# smoothly curving stroke instead
t <- seq(0, 2 * pi, length.out = 200)
```

```

draw(shape_stroke(x = t, y = sin(t), width = 0.15, fill = fill_charcoal()))

# layer effect_tremor() on top for a hand-drawn ink look
draw(effect_tremor(
  shape_stroke(
    x = c(0, 1, 2, 3),
    y = c(0, 1, 0, 1),
    width = 0.2,
    fill_alpha = 0.4
  ),
  layers = 4L
))

draw(shape_strokes(
  x = list(c(0, 1, 2, 3), c(0, 1, 2, 3)),
  y = list(c(0, 1, 0, 1), c(1, 2, 1, 2)),
  width = 0.3
))

# a shared distortion recycles across every stroke; width can vary too
draw(shape_strokes(
  x = list(c(0, 1, 2), c(0, 1, 2)),
  y = list(c(0, 1, 0), c(2, 3, 2)),
  width = c(0.15, 0.35)
))

```

shape_strokepath

A ribbon following an arbitrary curve

Description

shape_strokepath() builds a tapered, noise-modulated ribbon (like [shape_stroke\(\)](#)) whose backbone follows an arbitrary curve_*() drawable's own computed points, rather than raw x/y control points.

shape_strokepaths() is a vectorized version of [shape_strokepath\(\)](#): each argument may be a vector, recycled against the others via purrr::pmap()'s own vctrs-based rules. The result is a [sketch](#) containing one shape_strokepath() per recycled row, rather than a single drawable.

Usage

```

shape_strokepath(
  path,
  width = 0.2,
  n = 100L,
  distortion = noise_field(),
  trans = trans_identity(),
  ...
)

```

```

shape_strokepaths(
  path,
  width = 0.2,
  n = 100L,
  distortion = noise_field(),
  trans = trans_identity(),
  ...
)

```

Arguments

path	A drawable with geometry == "path" (i.e. any <code>curve_*</code> () constructor's output, e.g. <code>curve_bezier()</code> , <code>curve_line()</code> , <code>curve_twist()</code> , <code>curve_spiral()</code> , <code>curve_arc()</code> , <code>curve_scribble()</code> , <code>curve_raw()</code>) – its own computed points become this ribbon's backbone.
width	Maximum width. Must be non-negative. Default 0.2.
n	Number of points used along the ribbon, resampled evenly by arc length from path's own points (independent of however many points path itself was sampled at). Must be at least 2. Default 100L.
distortion	A noise_field controlling the width ("pressure") modulation. Default <code>noise_field()</code> .
trans	A trans object applied to the ribbon's computed points, on top of any trans already applied to path itself. Default <code>trans_identity()</code> (no transform).
...	Arguments passed to <code>style()</code> .

Details

It is a thin wrapper: `path@points` is extracted and fed straight into `shape_stroke()`, so the object it returns is literally a `shape_stroke` – `shape_strokepath()` exists only to save the caller from writing `shape_stroke(x = path@points@x, y = path@points@y, ...)` by hand, and to make the "ribbon around a curve" use case discoverable under its own name.

Because the result is a `shape_stroke`, its width offset uses a true per-point unit normal (`shape_stroke()`'s own `stroke_normals()` helper) rather than a single shared offset direction – this is what lets `shape_strokepath()` work correctly for any backbone shape, including ones that loop or bend sharply (e.g. `curve_twist()`, `curve_spiral()`), not just a nearly-straight one. Passing a `curve_bezier()` here covers the same use case the package's earlier `shape_bezier_ribbon()` (since removed) provided, but the two are not numerically identical for a curved backbone: `shape_bezier_ribbon()` offset by one shared direction vector (the straight line from its start to end point) for the whole ribbon, and its taper formula peaked at 0.5; `shape_strokepath()` instead inherits `shape_stroke()`'s true per-point normals and its taper formula (which peaks at 1, so width is exactly the maximum rendered width).

Any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error. Unlike the x/y-list-column constructors (e.g. `shape_beziers()`), `path` is a single [drawable](#) object per ribbon, not a numeric vector, so a single shared path recycles automatically across every ribbon (the same way a shared distortion [noise_field](#) already does); pass a `list()` of several different `curve_*`() objects instead to vary the backbone per ribbon.

Value

A [drawable](#) (a [shape_stroke](#)).

For [shape_strokepaths\(\)](#), a [sketch](#).

See Also

Other 2D shapes: [shape_bezier\(\)](#), [shape_blob\(\)](#), [shape_circle\(\)](#), [shape_combine\(\)](#), [shape_ellipse\(\)](#), [shape_polygon\(\)](#), [shape_raw\(\)](#), [shape_rectangle\(\)](#), [shape_ribbon\(\)](#), [shape_star\(\)](#), [shape_stroke\(\)](#), [shape_twist\(\)](#), [shape_wedge\(\)](#)

Examples

```
draw(shape_strokepath(
  curve_bezier(x = c(0, 0.25, 0.75, 1), y = c(0, 1, -1, 0)),
  width = 0.2
))
draw(shape_strokepath(
  curve_twist(
    x = 0,
    y = 0,
    xend = 1,
    yend = 0,
    path_distortion = noise_bridge(seed = 7734)
  ),
  width = 0.15
))

# a ribbon around a spiral -- a backbone shape_ribbon()/shape_twist()'s
# shared single offset direction couldn't render correctly
draw(shape_strokepath(
  curve_spiral(radius_start = 0.1, radius_end = 1, turns = 3),
  width = 0.1, fill = fill_charcoal()
))

draw(shape_strokepaths(
  path = list(
    curve_bezier(x = c(0, 0.25, 0.75, 1), y = c(0, 1, -1, 0)),
    curve_line(x = c(0, 1, 2), y = c(2, 3, 2))
  ),
  width = 0.2
))
```

Description

shape_twist is like [shape_ribbon](#), but the underlying path is a Brownian bridge rather than a straight line, giving the polygon a wandering, twisted appearance.

shape_twists() is a vectorized version of [shape_twist\(\)](#): each argument may be a vector, recycled against the others via purrr::pmap()'s own vctrs-based rules. The result is a [sketch](#) containing one shape_twist() per recycled row, rather than a single drawable.

Usage

```
shape_twist(
  x = 0,
  y = 0,
  xend = 1,
  yend = 1,
  width = 0.2,
  n = 100L,
  path_distortion = noise_bridge(),
  distortion = noise_field(),
  trans = trans_identity(),
  ...
)

shape_twists(
  x = 0,
  y = 0,
  xend = 1,
  yend = 1,
  width = 0.2,
  n = 100L,
  path_distortion = noise_bridge(),
  distortion = noise_field(),
  trans = trans_identity(),
  ...
)
```

Arguments

x, y	Start point. Default 0.
xend, yend	End point. Default 1.
width	Maximum width. Must be non-negative. Default 0.2.
n	Number of points used along the path. Default 100L.
path_distortion	A noise_bridge controlling the path's Brownian bridge. Default noise_bridge().
distortion	A noise_field controlling the width modulation. Default noise_field().
trans	A trans object applied to the shape's computed points. Default trans_identity() (no transform).
...	Arguments passed to style() .

Details

Any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error. A shared path_distortion/distortion is automatically recycled across every twist; pass a list() of several different [noise_bridge/noise_field](#) objects instead to vary either per twist – as in README.Rmd's "Twists" example, which gives every twist the same path_distortion this way.

Value

A [drawable](#).

For shape_twists(), a [sketch](#).

See Also

Other 2D shapes: [shape_bezier\(\)](#), [shape_blob\(\)](#), [shape_circle\(\)](#), [shape_combine\(\)](#), [shape_ellipse\(\)](#), [shape_polygon\(\)](#), [shape_raw\(\)](#), [shape_rectangle\(\)](#), [shape_ribbon\(\)](#), [shape_star\(\)](#), [shape_stroke\(\)](#), [shape_strokepath\(\)](#), [shape_wedge\(\)](#)

Examples

```
draw(shape_twist(
  x = 0, y = 0, xend = 1, yend = 0, width = 0.2,
  path_distortion = noise_bridge(seed = 7734)
))

# more smoothing passes make the Brownian bridge wander more gently;
# fewer passes leave it jumpier
draw(shape_twist(
  x = 0, y = 0, xend = 1, yend = 0, width = 0.2,
  path_distortion = noise_bridge(smooth = 20L, seed = 7734)
))
draw(shape_twist(
  x = 0, y = 0, xend = 1, yend = 0, width = 0.2,
  path_distortion = noise_bridge(smooth = 0L, seed = 7734)
))

draw(shape_twists(x = 1:3, y = 0, xend = 2:4, yend = 1, width = 0.2))

# every twist sharing one path_distortion gives them a family
# resemblance, as in README.Rmd's "Twists" example
draw(shape_twists(
  x = 0, y = 1:5, xend = 3, yend = 1:5, width = 0.15,
  path_distortion = noise_bridge(seed = 2020)
))
```

shape_wedge

A pie-slice wedge or annulus segment

Description

shape_wedge is a [drawable](#) defined by a centroid, a radius, and a start/end angle (in radians): its outline is the centroid, followed by n points along the circular arc from start to end.

shape_wedges() is a vectorized version of [shape_wedge\(\)](#): each argument may be a vector, recycled against the others. The result is a [sketch](#) containing one shape_wedge() per recycled row, rather than a single drawable.

Usage

```
shape_wedge(
  x = 0,
  y = 0,
  radius = 1,
  inner_radius = 0,
  start = 0,
  end = pi/2,
  n = 100L,
  trans = trans_identity(),
  ...
)

shape_wedges(
  x = 0,
  y = 0,
  radius = 1,
  inner_radius = 0,
  start = 0,
  end = pi/2,
  n = 100L,
  trans = trans_identity(),
  ...
)
```

Arguments

x, y	Centroid coordinates. Default 0.
radius	Radius. Must be non-negative. Default 1.
inner_radius	Inner radius. Must be non-negative and no greater than radius. Default 0 (a pie-slice wedge, i.e. no inner arc; see Details for the ring-slice/annulus-segment shape a positive value gives instead).
start, end	Start/end angle of the arc, in radians. Default 0/ pi / 2.

n	Number of points used to approximate the arc. Must be at least 2. Default 100L.
trans	A trans object applied to the shape's computed points. Default trans_identity() (no transform).
...	Arguments passed to style() .

Details

grid's own polygon closing then draws the final straight edge back from the arc's last point to the centroid, giving the familiar pie-slice/wedge shape. [curve_arc\(\)](#) is the arc alone, with no centroid vertex or fill.

`inner_radius` (default 0) turns the pie slice into a ring slice (an annulus segment): when greater than 0, the centroid vertex is dropped entirely, and the outline instead traces the outer arc from start to end followed by a second, inner arc of radius `inner_radius` swept back from end to start – grid's own polygon closing then draws the final straight edge back to the outer arc's first point, giving a four-sided (two arcs, two straight radial edges) ring-slice outline rather than a pie slice's three-sided one (two straight edges meeting at the centroid, one arc). `inner_radius = 0` (the default) recovers the original pie-slice outline exactly, since a zero-radius "inner arc" would otherwise degenerate to the centroid repeated n times.

Recycling uses `purrr::pmap()`'s own vctrs-based rules: any length-1 element is broadcast to the common length; mismatched lengths greater than 1 raise an error.

Value

A [drawable](#).

For `shape_wedges()`, a [sketch](#).

See Also

Other 2D shapes: [shape_bezier\(\)](#), [shape_blob\(\)](#), [shape_circle\(\)](#), [shape_combine\(\)](#), [shape_ellipse\(\)](#), [shape_polygon\(\)](#), [shape_raw\(\)](#), [shape_rectangle\(\)](#), [shape_ribbon\(\)](#), [shape_star\(\)](#), [shape_stroke\(\)](#), [shape_strokepath\(\)](#), [shape_twist\(\)](#)

Examples

```
draw(shape_wedge(start = 0, end = pi / 2))
draw(shape_wedge(
  x = 1,
  y = 1,
  radius = 0.5,
  start = pi,
  end = 2 * pi,
  color = "darkred"
))

# a nearly-full sweep gives a pac-man-like shape; the arc always closes
# straight back to the centroid
draw(shape_wedge(start = 0, end = 1.9 * pi, fill = "goldenrod"))

# inner_radius > 0 gives a ring slice (annulus segment) instead of a
```

```
# pie slice -- no centroid vertex, a hole in the middle
draw(shape_wedge(
  radius = 1, inner_radius = 0.6, start = 0, end = 1.5 * pi,
  fill = "steelblue"
))

# a full sweep (start = 0, end = 2 * pi) with inner_radius > 0 gives a
# complete ring/annulus
draw(shape_wedge(radius = 1, inner_radius = 0.7, start = 0, end = 2 * pi))

draw(shape_wedges(start = 0, end = seq(pi / 2, 2 * pi, length.out = 3)))

# a pie chart: adjacent wedges sharing a centroid, one slice per value
value <- c(30, 20, 50)
cum <- c(0, cumsum(value)) / sum(value) * 2 * pi
draw(shape_wedges(
  start = cum[-length(cum)], end = cum[-1],
  fill = c("steelblue", "tomato", "goldenrod")
))

# a donut chart: the same idea, with inner_radius > 0
draw(shape_wedges(
  inner_radius = 0.5,
  start = cum[-length(cum)], end = cum[-1],
  fill = c("steelblue", "tomato", "goldenrod")
))
```

sketch

A collection of drawable objects

Description

sketch is a list of [drawable](#) objects that can be rendered together with a single call to [draw\(\)](#). Sketches can be built up incrementally using the + operator, e.g. sketch() + shape_circle() + shape_circle(x = 2).

Usage

```
sketch(shapes = list(), canvas = sketchpad::canvas())
```

Arguments

shapes	A list of drawable -classed objects. Default list().
canvas	A canvas object, giving the background/framing settings draw() applies to the sketch as a whole. Default canvas() (no background, axis limits computed from shapes).

Details

A sketch also supports list-like access to its shapes: `length(s)` counts them, `s[[i]]` returns the single [drawable](#) at position `i`, and `s[i]` returns a new sketch containing only the selected shapes (its canvas is preserved) – mirroring how `[[/[]` differ on a plain list.

See Also

Other core structure: [canvas\(\)](#), [draw\(\)](#), [drawable\(\)](#), [group\(\)](#), [style\(\)](#), [xy\(\)](#)

Examples

```
s <- sketch() + shape_circle(radius = 1) + shape_circle(x = 2, radius = 0.5)
draw(s)

s2 <- sketch(canvas = canvas(background = "grey95")) +
  shape_circle(radius = 1)
draw(s2)

length(s)
s[[1]]
s[1]

# a trans applied to a sketch composes onto every shape's own @trans
draw(s + trans_rotate(pi / 6))

# accumulate many shapes in a loop, e.g. a ring of circles
ring <- sketch()
for (angle in seq(0, 2 * pi, length.out = 9)[-9]) {
  ring <- ring + shape_circle(x = cos(angle), y = sin(angle), radius = 0.3)
}
draw(ring)
```

style

Graphical style for a drawable object

Description

`style` is a container for the graphical properties passed to [grid::gpar\(\)](#) when a [drawable](#) object is drawn.

Usage

```
style(
  color = "black",
  fill = fill_solid("black"),
  linewidth = 1,
  linetype = "solid",
  linejoin = "round",
```

```

    lineend = "round",
    linemitre = 10,
    rule = "evenodd",
    color_alpha = 1,
    fill_alpha = 1
)

```

Arguments

color	Stroke colour: a single colour string recognised by <code>grDevices::col2rgb()</code> (a name, <code>"#RRGGBB"/"#RRGGBBAA"</code> hex string, ...), or NA for a fully transparent stroke. Validated at construction time, rather than only surfacing as a <code>grid</code> error once <code>draw()</code> is called. Default "black".
fill	Fill colour or pattern. Either a plain colour string, or the output of a <code>fill_*()</code> helper – <code>fill_solid()</code> , <code>fill_none()</code> , <code>fill_hatch()</code> , <code>fill_crosshatch()</code> , <code>fill_stipple()</code> , <code>fill_noise()</code> , <code>fill_gradient()</code> , or <code>fill_vignette()</code> . A bare colour string/ <code>GridPattern</code> is coerced into a <code>fill</code> object automatically. Default <code>fill_solid("black")</code> (i.e. "black").
linewidth	Line width. Default 1.
linetype	Line dash pattern, forwarded to <code>grid::gpar()</code> 's <code>lty</code> . Either a named type ("solid", "dashed", "dotted", "dotdash", "longdash", "twodash", "blank"), an integer code 0:6, or a custom hex dash-pattern string (e.g. "13") – see <code>grid::gpar()</code> and <code>graphics::par()</code> 's <code>lty</code> for the full set of accepted forms, which aren't independently re-validated here. Default "solid".
linejoin	Line join style at each vertex, forwarded to <code>grid::gpar()</code> 's <code>linejoin</code> . One of "round", "mitre", or "bevel". Most visible on closed shapes with few, sharp vertices, or on any drawable stroked with a thick linewidth. Default "round".
lineend	Line end style at a path's free endpoints, forwarded to <code>grid::gpar()</code> 's <code>lineend</code> . One of "round", "butt", or "square". Only visible on "path"-geometry drawables (e.g. <code>curve_line()</code> , <code>curve_bezier()</code>) – a "polygon"-geometry drawable has no free endpoint, since its outline closes back on itself. Most visible at a thick linewidth. Default "round".
linemitre	Mitre limit, forwarded to <code>grid::gpar()</code> 's <code>linemitre</code> . Only takes effect when <code>linejoin = "mitre"</code> : at a vertex sharper than this limit allows, the mitred corner is truncated to a bevel instead, to avoid an arbitrarily long spike. Must be at least 1. Default 10, matching <code>grid::gpar()</code> 's own default.
rule	Fill rule used when a drawable's own points has more than one sub-path (see <code>xy</code> 's <code>id</code>), forwarded to <code>grid::pathGrob()</code> 's own <code>rule</code> argument. One of "evenodd" (default) or "winding". "evenodd" fills a region if it's enclosed by an odd number of sub-paths, regardless of each sub-path's own vertex winding direction – a sub-path nested inside another becomes a hole purely from geometric nesting, with no need to get vertex order right by hand, which is why it's the default. "winding" instead fills based on net signed winding number, which depends on each sub-path's own direction – only useful for constructions that specifically need that direction-sensitive behavior. Has no effect on a drawable with only one implicit sub-path (every current <code>shape_*()</code> / <code>curve_*()</code> constructor), since both rules agree there.

color_alpha	Stroke opacity, applied to color independently of fill_alpha. Must be a number in $[0, 1]$, where 0 is fully transparent and 1 (the default) is fully opaque. Applied by baking the value into color via <code>grDevices::adjustcolor()</code> at draw time (see <code>draw()</code> 's internal <code>apply_alpha()</code> helper), not via <code>grid::gpar()</code> 's own alpha argument – <code>gpar()</code> 's alpha applies uniformly to both stroke and fill on the same grob, which would couple color_alpha and fill_alpha together. If color already has its own alpha channel (e.g. an <code>"#RRGGBBAA"</code> hex string), color_alpha multiplies through it rather than overriding it.
fill_alpha	Fill opacity, applied to fill independently of color_alpha, via the same <code>grDevices::adjustcolor()</code> mechanism as color_alpha. Must be a number in $[0, 1]$. Default 1. Only has an effect when fill is a plain colour string (as from <code>fill_solid()</code> or <code>fill_none()</code>) – silently inert when fill is a pattern or gradient (the output of any other <code>fill_*()</code> helper), since <code>grDevices::adjustcolor()</code> has no defined effect on a <code>GridPattern</code> object. This mirrors fill itself already having no effect for "path"/"points"-geometry drawables (see <code>drawable</code> 's geometry docs), and <code>lineend/linemitre</code> already being inert for some geometries – geometry- or fill-type-conditional inertness, not an error, is this package's existing convention for style properties that don't universally apply.

See Also

Other core structure: `canvas()`, `draw()`, `drawable()`, `group()`, `sketch()`, `xy()`

Examples

```
style(color = "steelblue", fill = "lightblue", linewidth = 2)
style(fill = fill_hatch(angle = 30))

# linejoin/linemitre are most visible on a thick-stroked shape with a
# sharp vertex
star <- shape_polygon(n = 5, radius = 1, fill = "white")
draw(shape_stroke(
  x = star@points@x, y = star@points@y, width = 0.25,
  linejoin = "mitre", linemitre = 1.5
))

# color_alpha/fill_alpha control stroke/fill opacity independently
draw(shape_circle(
  radius = 1, color = "black", fill = "tomato",
  color_alpha = 1, fill_alpha = 0.3, linewidth = 3
))

# lineend only affects a path's free endpoints, not a closed polygon
draw(curve_line(
  x = c(0, 1, 2), y = c(0, 1, 0), linewidth = 15, lineend = "square"
))
```

trans	<i>A 2D affine transformation</i>
-------	-----------------------------------

Description

trans wraps a 3x3 homogeneous-coordinates affine transformation matrix. It is not usually constructed directly – use [trans_identity\(\)](#), [trans_translate\(\)](#), [trans_rotate\(\)](#), [trans_scale\(\)](#), [trans_reflect\(\)](#), [trans_shear\(\)](#), or the general-purpose [trans_affine\(\)](#) instead.

Usage

```
trans(matrix = integer(0))
```

Arguments

matrix	A 3x3 numeric matrix in homogeneous coordinates (i.e. its third row must be <code>c(0, 0, 1)</code>).
--------	--

Details

Every [drawable](#) carries a trans property (default [trans_identity\(\)](#)) that's applied to its computed points as the very last step – see [drawable](#)'s trans documentation.

Two trans objects combine with `+`: `t1 + t2` produces a new trans whose effect is "apply t1 first, then t2" – see [trans_translate\(\)](#)'s Details for the composition order convention and a worked example. Composing a trans with a [trans_warp](#) (a non-rigid, noise-based deformation) or a [trans_fn](#) (a non-rigid deformation from an arbitrary caller-supplied function) – neither of which can be represented as a matrix – instead produces a [trans_chain](#) – see there.

See Also

Other transform helpers: [trans_affine\(\)](#), [trans_chain\(\)](#), [trans_fn\(\)](#), [trans_identity\(\)](#), [trans_reflect\(\)](#), [trans_rotate\(\)](#), [trans_scale\(\)](#), [trans_shear\(\)](#), [trans_translate\(\)](#), [trans_warp\(\)](#)

Examples

```
trans_identity()
trans_translate(1, 0) + trans_rotate(pi / 4)

# overlay a shape's original outline (faded) with a transformed copy
# (solid) to see a trans's effect directly
original <- shape_rectangle(
  width = 1.5,
  height = 0.6,
  fill_alpha = 0.3,
  color_alpha = 0.3
)
draw(sketch() + original + (original + trans_rotate(pi / 6)))
```

trans_affine	<i>Build a custom affine transform</i>
--------------	--

Description

General-purpose escape hatch for an affine transform not covered by [trans_translate\(\)](#)/[trans_rotate\(\)](#)/[trans_scale\(\)](#)/[trans_shear\(\)](#): wraps a caller-supplied matrix directly.

Usage

```
trans_affine(matrix)
```

Arguments

matrix	A 3x3 numeric matrix in homogeneous coordinates (its third row must be <code>c(0, 0, 1)</code>), or a 2x3 matrix giving only the first two rows (the third row is added automatically).
--------	--

See Also

Other transform helpers: [trans\(\)](#), [trans_chain\(\)](#), [trans_fn\(\)](#), [trans_identity\(\)](#), [trans_reflect\(\)](#), [trans_rotate\(\)](#), [trans_scale\(\)](#), [trans_shear\(\)](#), [trans_translate\(\)](#), [trans_warp\(\)](#)

Examples

```
# equivalent to trans_scale(2, 3)
trans_affine(matrix(c(2, 0, 0, 0, 3, 0, 0, 0, 1), nrow = 3, byrow = TRUE))

# the 2x3 form omits the trivial homogeneous third row
draw(shape_square(
  side = 1,
  trans = trans_affine(matrix(c(2, 0, 0, 0, 3, 0), nrow = 2, byrow = TRUE))
))
```

trans_chain	<i>A sequence of composed transforms</i>
-------------	--

Description

`trans_chain` is what `+` produces when combining transforms that can't collapse into a single [trans](#) matrix – e.g. a [trans_warp](#) with another [trans_warp](#), or a [trans_warp](#) mixed with a [trans](#). It is not usually constructed directly; it holds an ordered list of steps (each a [trans](#), [trans_warp](#), or [trans_fn](#)), applied in sequence – `steps[[1]]` first, `steps[[length(steps)]]` last – exactly like chained `+` calls on a [drawable](#) would suggest.

Usage

```
trans_chain(steps = list())
```

Arguments

steps A list of [trans/trans_warp](#)/trans_chain objects.

Details

Two consecutive [trans](#) (affine) steps are *not* automatically collapsed into one matrix when they're already part of a chain (only a bare trans + trans collapses); this only costs a little efficiency, not correctness.

See Also

Other transform helpers: [trans\(\)](#), [trans_affine\(\)](#), [trans_fn\(\)](#), [trans_identity\(\)](#), [trans_reflect\(\)](#), [trans_rotate\(\)](#), [trans_scale\(\)](#), [trans_shear\(\)](#), [trans_translate\(\)](#), [trans_warp\(\)](#)

Examples

```
# combining an affine trans with a trans_warp produces a trans_chain,
# applied in the order given by +
chained <- trans_rotate(pi / 6) + trans_warp(amount = 0.08)
chained
draw(shape_rectangle(width = 1.5, height = 0.6, trans = chained))
```

trans_fn

An arbitrary-function, non-rigid deformation

Description

trans_fn is the general-purpose escape hatch for a non-rigid deformation not covered by [trans_warp\(\)](#)'s noise-driven domain warping: it wraps a caller-supplied displacement function directly, the same relationship [trans_affine\(\)](#) has to the rigid trans_*() family.

Usage

```
trans_fn(fn = function() NULL)
```

Arguments

fn A function taking two numeric vectors (x, y) and returning a list(x = ..., y = ...) of the same length.

Details

fn is called as fn(x, y), where x/y are a [drawable](#)'s own computed points (already flattened to plain numeric vectors, not an [xy](#) object), and must return a `list(x = ..., y = ...)` of the same length – checked at draw/apply time (not at construction), since fn's own behavior can't be verified without calling it. This makes `trans_fn` strictly more general than `trans_warp()`: any noise-based warp could be expressed as a `trans_fn` closing over a [noise_field](#), but also deterministic formulas (a swirl, pinch, or bulge) or a warp driven by something [noise_field](#) can't express at all, e.g. a second `drawable`'s own geometry captured in fn's enclosing environment.

Like [trans_warp](#), this can't be represented as a single matrix, so composing it with a [trans](#) or another non-rigid deformation with `+` produces a [trans_chain](#) rather than collapsing.

See Also

Other transform helpers: [trans\(\)](#), [trans_affine\(\)](#), [trans_chain\(\)](#), [trans_identity\(\)](#), [trans_reflect\(\)](#), [trans_rotate\(\)](#), [trans_scale\(\)](#), [trans_shear\(\)](#), [trans_translate\(\)](#), [trans_warp\(\)](#)

Examples

```
# a deterministic swirl: rotate each point by an angle that grows with
# its own distance from the origin. A shape centred at the origin
# (e.g. a plain shape_circle()) is rotationally symmetric about it, so
# every point shares the same distance and the swirl just rotates the
# whole shape rigidly -- offsetting the shape away from the origin
# gives points at varying distances instead, showing the effect
swirl <- function(x, y) {
  r <- sqrt(x^2 + y^2)
  theta <- atan2(y, x) + r * 1.5
  list(x = r * cos(theta), y = r * sin(theta))
}
draw(shape_circle(x = 1, radius = 0.4, n = 200, trans = trans_fn(swirl)))

# a bulge: points near the origin are pushed outward more than points
# far from it. Offsetting the shape away from the origin (for the same
# reason as the swirl example above) means one side sits closer to the
# origin than the other, so the bulge dents that side outward more
bulge <- function(x, y) {
  r <- sqrt(x^2 + y^2)
  scale_factor <- 1 + 0.6 * exp(-4 * r^2)
  list(x = x * scale_factor, y = y * scale_factor)
}
draw(shape_circle(x = 0.8, radius = 0.5, n = 200, trans = trans_fn(bulge)))

# combining a trans_fn with a trans (or a trans_warp) produces a
# trans_chain, applied in the order given by +
draw(shape_circle(
  x = 0.8, radius = 0.5, n = 200,
  trans = trans_fn(bulge) + trans_rotate(pi / 6)
))
```

trans_identity	<i>The identity transform</i>
----------------	-------------------------------

Description

Returns a [trans](#) that leaves points unchanged – the default trans for every [drawable](#).

Usage

```
trans_identity()
```

See Also

Other transform helpers: [trans\(\)](#), [trans_affine\(\)](#), [trans_chain\(\)](#), [trans_fn\(\)](#), [trans_reflect\(\)](#), [trans_rotate\(\)](#), [trans_scale\(\)](#), [trans_shear\(\)](#), [trans_translate\(\)](#), [trans_warp\(\)](#)

Examples

```
trans_identity()

# the default trans for every drawable -- points are left unchanged
draw(shape_rectangle(width = 1.5, height = 0.6, trans = trans_identity()))
```

trans_reflect	<i>Reflect points</i>
---------------	-----------------------

Description

Flips points across a vertical and/or horizontal line through the pivot (about_x, about_y): x = TRUE mirrors left-right (flips the x-coordinate), y = TRUE mirrors top-bottom (flips the y-coordinate). Setting both reflects through the pivot point itself (equivalent to a half turn, i.e. trans_rotate(pi, about_x, about_y)).

Usage

```
trans_reflect(x = FALSE, y = FALSE, about_x = 0, about_y = 0)
```

Arguments

x, y	Whether to flip the x/y coordinate. Default FALSE for both (i.e. the identity transform).
about_x, about_y	Pivot point coordinates. Default 0.

Details

Reflection across an arbitrary-angle line isn't exposed as a separate argument – it composes from existing pieces instead: rotate the desired line onto an axis, reflect, then rotate back, e.g. `trans_rotate(-theta) + trans_reflect(y = TRUE) + trans_rotate(theta)` reflects across a line through the origin at angle `theta`.

See Also

Other transform helpers: [trans\(\)](#), [trans_affine\(\)](#), [trans_chain\(\)](#), [trans_fn\(\)](#), [trans_identity\(\)](#), [trans_rotate\(\)](#), [trans_scale\(\)](#), [trans_shear\(\)](#), [trans_translate\(\)](#), [trans_warp\(\)](#)

Examples

```
draw(shape_bezier(
  x = c(0, 0.5, 1),
  y = c(0, 1, 0.2),
  trans = trans_reflect(x = TRUE)
))

# overlay the original (faded) with each reflected copy (solid)
original <- shape_rectangle(
  x = 1,
  width = 1,
  height = 0.4,
  fill_alpha = 0.3,
  color_alpha = 0.3
)
draw(sketch() + original + (original + trans_reflect(x = TRUE)))
draw(sketch() + original + (original + trans_reflect(y = TRUE)))

# reflecting through both axes is equivalent to a half turn
draw(shape_bezier(
  x = c(0, 0.5, 1), y = c(0, 1, 0.2),
  trans = trans_reflect(x = TRUE, y = TRUE)
))
```

trans_rotate

Rotate points

Description

Rotates points by angle radians (counterclockwise for positive angles) about the pivot (about_x, about_y).

Usage

```
trans_rotate(angle, about_x = 0, about_y = 0)
```

Arguments

angle Rotation angle, in radians.
 about_x, about_y Pivot point coordinates. Default 0.

See Also

Other transform helpers: [trans\(\)](#), [trans_affine\(\)](#), [trans_chain\(\)](#), [trans_fn\(\)](#), [trans_identity\(\)](#), [trans_reflect\(\)](#), [trans_scale\(\)](#), [trans_shear\(\)](#), [trans_translate\(\)](#), [trans_warp\(\)](#)

Examples

```
draw(shape_square(side = 1, trans = trans_rotate(pi / 4)))

# overlay the original (faded) with the rotated copy (solid)
original <- shape_rectangle(
  width = 1.5,
  height = 0.6,
  fill_alpha = 0.3,
  color_alpha = 0.3
)
draw(sketch() + original + (original + trans_rotate(pi / 6)))

# rotating about a pivot away from the shape's own centroid sweeps it
# around that point instead -- shown here against a fixed frame, with
# a small marker at the pivot
square <- shape_square(
  x = 2,
  side = 0.5,
  fill_alpha = 0.3,
  color_alpha = 0.3
)
pivot <- shape_circle(radius = 0.05, fill = "tomato", color = NA_character_)
draw(
  sketch() + square +
    (square + trans_rotate(pi / 2, about_x = 0, about_y = 0)) +
    pivot,
  xlim = c(-2.5, 2.5), ylim = c(-2.5, 2.5)
)
```

trans_scale

Scale points

Description

Scales points by s_x/s_y along the x/y axes, about the pivot (`about_x`, `about_y`).

Usage

```
trans_scale(sx = 1, sy = sx, about_x = 0, about_y = 0)
```

Arguments

sx Scale factor along the x axis. Default 1.

sy Scale factor along the y axis. Default sx (a uniform scale).

about_x, about_y Pivot point coordinates. Default 0.

See Also

Other transform helpers: [trans\(\)](#), [trans_affine\(\)](#), [trans_chain\(\)](#), [trans_fn\(\)](#), [trans_identity\(\)](#), [trans_reflect\(\)](#), [trans_rotate\(\)](#), [trans_shear\(\)](#), [trans_translate\(\)](#), [trans_warp\(\)](#)

Examples

```
draw(shape_square(side = 1, trans = trans_scale(2)))
draw(shape_square(side = 1, trans = trans_scale(2, 0.5)))

# overlay the original (faded) with the scaled copy (solid)
original <- shape_rectangle(
  width = 1.5,
  height = 0.6,
  fill_alpha = 0.3,
  color_alpha = 0.3
)
draw(sketch() + original + (original + trans_scale(1.5, 0.7)))

# scaling about a pivot other than the shape's own centroid also moves
# it, since distance from the pivot is what gets scaled -- shown here
# against a fixed frame, with a small marker at the pivot
square <- shape_square(
  x = 1,
  side = 0.5,
  fill_alpha = 0.3,
  color_alpha = 0.3
)
pivot <- shape_circle(radius = 0.05, fill = "tomato", color = NA_character_)
draw(
  sketch() + square +
    (square + trans_scale(2, about_x = 0, about_y = 0)) +
    pivot,
  xlim = c(-3, 3), ylim = c(-3, 3)
)
```

trans_shear	<i>Shear points</i>
-------------	---------------------

Description

Applies a shear transform about the pivot (about_x, about_y): shx displaces x-coordinates in proportion to y (distance from about_y), shy displaces y-coordinates in proportion to x (distance from about_x).

Usage

```
trans_shear(shx = 0, shy = 0, about_x = 0, about_y = 0)
```

Arguments

shx, shy Shear factors along the x/y axes. Default 0 for both.
about_x, about_y Pivot point coordinates. Default 0.

See Also

Other transform helpers: [trans\(\)](#), [trans_affine\(\)](#), [trans_chain\(\)](#), [trans_fn\(\)](#), [trans_identity\(\)](#), [trans_reflect\(\)](#), [trans_rotate\(\)](#), [trans_scale\(\)](#), [trans_translate\(\)](#), [trans_warp\(\)](#)

Examples

```
draw(shape_square(side = 1, trans = trans_shear(shx = 0.5)))

# overlay the original (faded) with the sheared copy (solid)
original <- shape_rectangle(
  width = 1,
  height = 0.6,
  fill_alpha = 0.3,
  color_alpha = 0.3
)
draw(sketch() + original + (original + trans_shear(shx = 0.6)))

# shy shears vertically instead of horizontally
draw(shape_square(side = 1, trans = trans_shear(shy = 0.5)))
```

trans_translate	<i>Translate points</i>
-----------------	-------------------------

Description

Shifts points by (dx, dy).

Usage

```
trans_translate(dx = 0, dy = 0)
```

Arguments

dx, dy Distance to shift along the x/y axes. Default 0.

Details

trans objects compose with +, applied left to right: `t1 + t2` means "apply `t1`'s effect first, then `t2`'s". For example, `trans_translate(1, 0) + trans_rotate(pi / 2)` translates a point one unit right, then rotates the *result* a quarter turn about the origin – not the same as `trans_rotate(pi / 2) + trans_translate(1, 0)`, which rotates first and translates second. Internally, `(t1 + t2)@matrix` is `t2@matrix %*% t1@matrix`, since points are homogeneous column vectors transformed as `matrix %*% point`.

See Also

Other transform helpers: [trans\(\)](#), [trans_affine\(\)](#), [trans_chain\(\)](#), [trans_fn\(\)](#), [trans_identity\(\)](#), [trans_reflect\(\)](#), [trans_rotate\(\)](#), [trans_scale\(\)](#), [trans_shear\(\)](#), [trans_warp\(\)](#)

Examples

```
draw(shape_square(side = 1, trans = trans_translate(2, 0)))

# overlay the original (faded) with the translated copy (solid)
original <- shape_rectangle(
  width = 1.5,
  height = 0.6,
  fill_alpha = 0.3,
  color_alpha = 0.3
)
draw(sketch() + original + (original + trans_translate(1, 0.5)))

# composition order matters: translate-then-rotate differs from
# rotate-then-translate -- a fixed xlim/ylim makes the difference in
# final position visible (a lone shape otherwise always fills its own
# auto-scaled frame)
square <- shape_square(x = 1, side = 0.4)
draw(
  square + trans_translate(1, 0) + trans_rotate(pi / 2),
```

```

    xlim = c(-1, 3),
    ylim = c(-1, 3)
)
draw(
  square + trans_rotate(pi / 2) + trans_translate(1, 0),
  xlim = c(-1, 3),
  ylim = c(-1, 3)
)

```

trans_warp

A non-rigid, noise-based deformation

Description

trans_warp displaces each point by simplex/fractal noise sampled at the point's own (x, y) position (domain warping), giving a wobbly, non-rigid distortion rather than an affine map.

Usage

```

trans_warp(
  amount = 0.1,
  distortion_x = noise_field(),
  distortion_y = noise_field(seed = distortion_x@seed + 1L)
)

```

Arguments

amount	Displacement amplitude. Must be non-negative. Default 0.1.
distortion_x	A noise_field controlling the x displacement. Default noise_field().
distortion_y	A noise_field controlling the y displacement. Default distortion_x, with its seed offset by 1.

Details

Unlike [trans](#) (translate/rotate/scale/reflect/shear), this can't be represented as a single matrix, since the displacement varies smoothly but irregularly from point to point.

The x and y displacements are sampled from two independent [noise_fields](#) (distortion_x/distortion_y), each rescaled to [-1, 1] and multiplied by amount. By default distortion_y reuses distortion_x's own settings with its seed offset by 1, so the two axes wander independently without the caller needing to specify two full [noise_field](#) objects – the same convention twisted_path_points() uses internally for [shape_twist\(\)](#)'s path.

Like [trans](#), a trans_warp is attached to a [drawable](#) via its trans property/argument, and composes with + – combining it with another trans_warp, a [trans_fn](#), or a [trans](#) produces a [trans_chain](#), since a non-rigid warp can't collapse into a single matrix. See [trans_fn\(\)](#) for a general-purpose escape hatch taking an arbitrary caller-supplied displacement function, rather than only a [noise_field](#)-driven one.

See Also

Other transform helpers: [trans\(\)](#), [trans_affine\(\)](#), [trans_chain\(\)](#), [trans_fn\(\)](#), [trans_identity\(\)](#), [trans_reflect\(\)](#), [trans_rotate\(\)](#), [trans_scale\(\)](#), [trans_shear\(\)](#), [trans_translate\(\)](#)

Examples

```
draw(shape_circle(radius = 1, n = 200, trans = trans_warp(amount = 0.15)))

# a smaller amount gives a subtler wobble; distortion_x's own frequency
# controls how quickly the warp varies across space
draw(shape_circle(radius = 1, n = 200, trans = trans_warp(amount = 0.03)))
draw(shape_circle(
  radius = 1, n = 200,
  trans = trans_warp(amount = 0.1, distortion_x = noise_field(frequency = 4))
))
```

xy

*A set of locations in 2D space***Description**

xy represents a collection of locations in two-dimensional space as parallel x/y coordinate vectors, plus a parallel id vector grouping those locations into sub-paths.

Usage

```
xy(x, y, id = NULL)
```

Arguments

x	Numeric vector of x coordinates.
y	Numeric vector of y coordinates.
id	Integer (or integerish numeric) vector the same length as x/y, grouping locations into sub-paths. Default NULL, filled in as <code>rep(1L, length(x))</code> (a single sub-path).

Details

Most [drawable](#) subclasses expose their geometry as a computed `points` property of class xy; [shape_raw](#) is the exception, where the user supplies x/y directly. Named xy rather than points so this exported constructor doesn't mask `graphics::points()`.

id marks which sub-path/contour each x/y location belongs to – locations sharing the same id are connected into one contour; a different id starts a new one. Every existing single-contour drawable has exactly one implicit sub-path, so id defaults to `rep(1L, length(x))` whenever left NULL (the default), meaning no drawable's own `points` getter needs to pass id explicitly to keep its current, single-contour behavior. Multiple sub-paths let a single [drawable](#) render as several disjoint shapes

sharing one `style` (e.g. two separate blobs), or – combined with `style()`'s rule – as a shape with a hole (a sub-path nested inside another). No constructor currently exposes a way to *set* a non-trivial `id` when building geometry by hand; this is the underlying data-shape change alone, not yet paired with an author-facing API (see `.agents/PLAN.md`).

See Also

Other core structure: `canvas()`, `draw()`, `drawable()`, `group()`, `sketch()`, `style()`

Examples

```
xy(x = c(0, 1, 1, 0), y = c(0, 0, 1, 1))
```

```
# two sub-paths: a pair of disjoint 4-point contours sharing one xy object
```

```
xy(x = c(0, 1, 1, 0, 2, 3, 3, 2), y = c(0, 0, 1, 1, 0, 0, 1, 1), id = rep(1:2, each = 4))
```

Index

- * **0D points**
 - points_raw, 76
- * **1D curves**
 - curve_arc, 11
 - curve_bezier, 13
 - curve_line, 15
 - curve_raw, 16
 - curve_scribble, 18
 - curve_spiral, 20
 - curve_twist, 22
- * **2D shapes**
 - shape_bezier, 79
 - shape_blob, 80
 - shape_circle, 82
 - shape_combine, 84
 - shape_ellipse, 85
 - shape_polygon, 87
 - shape_raw, 88
 - shape_rectangle, 90
 - shape_ribbon, 92
 - shape_star, 94
 - shape_stroke, 96
 - shape_strokepath, 98
 - shape_twist, 100
 - shape_wedge, 103
- * **color helpers**
 - color_alpha, 5
 - color_greyscale, 6
 - color_invert, 7
 - color_jitter, 7
 - color_lighten, 8
 - color_mix, 9
 - color_rotate_hue, 10
 - color_saturate, 11
- * **core structure**
 - canvas, 4
 - draw, 24
 - drawable, 25
 - group, 65
 - sketch, 105
 - style, 106
 - xy, 120
- * **effects**
 - effect_bristle, 27
 - effect_grain, 29
 - effect_swatch, 31
 - effect_tremor, 32
- * **export helpers**
 - save_png, 77
- * **fill helpers**
 - fill, 34
 - fill_charcoal, 35
 - fill_checker, 36
 - fill_crosshatch, 38
 - fill_flow, 40
 - fill_gradient, 42
 - fill_halftone, 44
 - fill_hatch, 45
 - fill_image, 47
 - fill_marble, 49
 - fill_noise, 51
 - fill_none, 53
 - fill_scatter, 54
 - fill_scribble, 56
 - fill_solid, 59
 - fill_stipple, 60
 - fill_stripe, 62
 - fill_vignette, 64
- * **noise helpers**
 - noise_bridge, 67
 - noise_field, 68
 - noise_sample, 69
- * **palette helpers**
 - palette_cosine, 70
 - palette_gradient, 71
 - palette_harmony, 72
 - palette_manual, 73
 - palette_tonal, 74

- palette_walk, 75
- * **transform helpers**
 - trans, 109
 - trans_affine, 110
 - trans_chain, 110
 - trans_fn, 111
 - trans_identity, 113
 - trans_reflect, 113
 - trans_rotate, 114
 - trans_scale, 115
 - trans_shear, 117
 - trans_translate, 118
 - trans_warp, 119
- abs(), 71
- ambient::billow(), 68
- ambient::fbm(), 68
- ambient::fracture(), 68, 69
- ambient::gen_perlin(), 68
- ambient::gen_simplex(), 68
- ambient::gen_worley(), 68
- canvas, 4, 105
- canvas(), 25, 26, 34, 66, 78, 106, 108, 121
- color_alpha, 5
- color_alpha(), 6–11
- color_darken (color_lighten), 8
- color_darken(), 11
- color_desaturate (color_saturate), 11
- color_desaturate(), 6
- color_greyscale, 6
- color_greyscale(), 5, 7–11
- color_invert, 7
- color_invert(), 5, 6, 8–11
- color_jitter, 7
- color_jitter(), 5–7, 9–11
- color_lighten, 8
- color_lighten(), 5–11
- color_mix, 9
- color_mix(), 5–11
- color_rotate_hue, 10
- color_rotate_hue(), 5–9, 11
- color_saturate, 11
- color_saturate(), 5–10
- convert(), 76, 84, 88
- curve_arc, 11
- curve_arc(), 11, 14, 16, 17, 20, 22, 24, 99, 104
- curve_arcs (curve_arc), 11
- curve_bezier, 13
- curve_bezier(), 13, 15–17, 20, 22, 24, 26, 79, 99, 107
- curve_beziers (curve_bezier), 13
- curve_line, 15
- curve_line(), 13–17, 20, 22, 24, 26, 32, 33, 97, 99, 107
- curve_lines (curve_line), 15
- curve_raw, 16, 25
- curve_raw(), 13, 14, 16, 20, 22, 24, 26, 76, 99
- curve_raws (curve_raw), 16
- curve_scribble, 18
- curve_scribble(), 13, 14, 16–18, 22, 24, 99
- curve_scribbles (curve_scribble), 18
- curve_spiral, 20
- curve_spiral(), 13, 14, 16, 17, 20, 24, 99
- curve_spirals (curve_spiral), 20
- curve_twist, 22
- curve_twist(), 13, 14, 16, 17, 20, 22, 26, 99
- curve_twists (curve_twist), 22
- draw, 24
- draw(), 4, 26, 34, 35, 37, 38, 40, 42, 44, 46, 48, 50, 52, 55, 57, 61, 62, 64, 66, 77–79, 105–108, 121
- drawable, 4, 12–15, 17–21, 23, 24, 25, 25, 27, 29–33, 53–55, 65, 66, 76–97, 99, 100, 102–106, 108–110, 112, 113, 119, 120
- drawable(), 4, 25, 66, 106, 108, 121
- effect_bristle, 27
- effect_bristle(), 26, 30–33
- effect_grain, 29
- effect_grain(), 28, 32, 33
- effect_swatch, 31
- effect_swatch(), 28, 30, 33
- effect_tremor, 32
- effect_tremor(), 26–28, 30–32
- fill, 4, 34, 35, 37, 38, 40, 44, 46, 48, 50, 52, 54, 57, 60–62, 107
- fill(), 36, 37, 39, 41, 43, 45, 47, 48, 50, 53, 54, 56, 58, 60, 61, 63, 65
- fill_charcoal, 35
- fill_charcoal(), 30, 34, 37, 39, 41, 43, 45, 47, 48, 50, 53, 54, 56, 58, 60, 61, 63, 65
- fill_checker, 36

- `fill_checker()`, 34, 36, 39, 41, 43, 45, 47, 48, 50, 53–56, 58, 60, 61, 63, 65
- `fill_crosshatch`, 38
- `fill_crosshatch()`, 34, 36, 37, 41, 43, 45, 47, 48, 50, 53, 54, 56, 58, 60, 61, 63, 65, 107
- `fill_flow`, 40
- `fill_flow()`, 34, 36, 37, 39, 43, 45, 47, 48, 50, 53, 54, 56, 58, 60, 61, 63, 65
- `fill_gradient`, 42
- `fill_gradient()`, 34, 36, 37, 39, 41, 45, 47, 48, 50, 53–56, 58, 60, 61, 63–65, 107
- `fill_halftone`, 44
- `fill_halftone()`, 28, 34, 36, 37, 39, 41, 43, 47, 48, 50, 53, 54, 56, 58, 60, 61, 63, 65
- `fill_hatch`, 45
- `fill_hatch()`, 34, 36, 37, 39, 41–43, 45, 48, 50, 52–58, 60, 61, 63–65, 107
- `fill_image`, 47
- `fill_image()`, 34, 36, 37, 39, 41, 43, 45, 47, 50, 53, 54, 56, 58, 60, 61, 63, 65
- `fill_marble`, 49
- `fill_marble()`, 34, 36, 37, 39, 41, 43, 45, 47, 48, 53, 54, 56, 58, 60, 61, 63, 65
- `fill_noise`, 51
- `fill_noise()`, 29, 30, 34–37, 39–41, 43, 45, 47–50, 54, 56, 58, 60, 61, 63, 65, 107
- `fill_none`, 53
- `fill_none()`, 4, 34, 36, 37, 39, 41, 43, 45, 47, 48, 50, 53, 56, 58, 60, 61, 63, 65, 107, 108
- `fill_scatter`, 54
- `fill_scatter()`, 28, 34, 36, 37, 39, 41, 43–45, 47, 48, 50, 53, 54, 58, 60, 61, 63, 65
- `fill_scribble`, 56
- `fill_scribble()`, 18, 34, 36, 37, 39, 41, 43, 45, 47, 48, 50, 53, 54, 56, 60, 61, 63, 65
- `fill_solid`, 59
- `fill_solid()`, 34, 36, 37, 39, 41, 43, 45, 47, 48, 50, 53, 54, 56, 58, 61, 63, 65, 107, 108
- `fill_stipple`, 60
- `fill_stipple()`, 28, 34, 36, 37, 39, 41, 43–45, 47, 48, 50, 53–56, 58, 60, 63–65, 107
- `fill_stripe`, 62
- `fill_stripe()`, 34, 36, 37, 39, 41, 43, 45, 47, 48, 50, 53, 54, 56, 58, 60, 61, 65
- `fill_vignette`, 64
- `fill_vignette()`, 30, 34, 36, 37, 39, 41, 43, 45, 47, 48, 50, 53, 54, 56, 58, 60, 61, 63, 107
- `grDevices`, 77
- `grDevices::adjustcolor()`, 5, 108
- `grDevices::as.raster()`, 48
- `grDevices::col2rgb()`, 107
- `grDevices::colorRampPalette()`, 71, 73
- `grDevices::convertColor()`, 9
- `grDevices::pdf()`, 78
- `grDevices::png()`, 78
- `grDevices::svg()`, 78
- `grid`, 26, 107
- `grid::as.mask()`, 29, 64, 65
- `grid::circleGrob()`, 44, 55, 61
- `grid::gpar()`, 36, 37, 39, 41, 43, 45, 46, 48, 50, 52, 53, 56, 58, 59, 61, 63, 65, 106–108
- `grid::linearGradient()`, 42, 62, 63
- `grid::pathGrob()`, 55, 107
- `grid::pattern()`, 30, 34–39, 41–52, 55–65
- `grid::pointsGrob()`, 26
- `grid::polygonGrob()`, 26, 53, 55
- `grid::polylineGrob()`, 26
- `grid::radialGradient()`, 42, 64
- `grid::rasterGrob()`, 48
- `group`, 27, 28, 31–33, 65
- `group()`, 4, 25, 26, 106, 108, 121
- `noise_bridge`, 4, 22, 23, 67, 69, 101, 102
- `noise_bridge()`, 68, 70
- `noise_field`, 4, 23, 29, 30, 67, 68, 69, 81, 93, 96, 97, 99, 101, 102, 112, 119
- `noise_field()`, 67, 69, 70
- `noise_sample`, 69
- `noise_sample()`, 67, 68
- `palette_cosine`, 70
- `palette_cosine()`, 72–75
- `palette_gradient`, 71
- `palette_gradient()`, 9, 71, 73–75
- `palette_harmony`, 72
- `palette_harmony()`, 10, 31, 71–75
- `palette_manual`, 73

- palette_manual(), [31](#), [71–75](#)
- palette_tonal, [74](#)
- palette_tonal(), [71–73](#), [75](#)
- palette_walk, [75](#)
- palette_walk(), [7](#), [71–74](#)
- points_raw, [25](#), [76](#)
- points_raw(), [26](#), [76](#)
- points_raws (points_raw), [76](#)

- S7::set_props(), [28](#), [31–33](#)
- save_pdf (save_png), [77](#)
- save_png, [77](#)
- save_svg (save_png), [77](#)
- shape_bezier, [79](#)
- shape_bezier(), [13](#), [15](#), [26](#), [32](#), [79](#), [81](#), [83](#), [84](#), [86](#), [88](#), [89](#), [91](#), [93](#), [95](#), [97](#), [100](#), [102](#), [104](#)
- shape_beziers (shape_bezier), [79](#)
- shape_beziers(), [99](#)
- shape_blob, [25](#), [80](#), [88](#)
- shape_blob(), [4](#), [26](#), [29](#), [35](#), [41](#), [50–52](#), [60](#), [68](#), [80](#), [83](#), [84](#), [86](#), [88](#), [89](#), [91](#), [93](#), [95](#), [97](#), [100](#), [102](#), [104](#)
- shape_blobs (shape_blob), [80](#)
- shape_circle, [25](#), [80](#), [82](#)
- shape_circle(), [26](#), [80–82](#), [84–86](#), [88](#), [89](#), [91](#), [93](#), [95](#), [97](#), [100](#), [102](#), [104](#)
- shape_circles (shape_circle), [82](#)
- shape_combine, [84](#)
- shape_combine(), [80](#), [81](#), [83](#), [86](#), [88](#), [89](#), [91](#), [93](#), [95](#), [97](#), [100](#), [102](#), [104](#)
- shape_ellipse, [85](#)
- shape_ellipse(), [80](#), [81](#), [83–85](#), [88](#), [89](#), [91](#), [93](#), [95](#), [97](#), [100](#), [102](#), [104](#)
- shape_ellipses (shape_ellipse), [85](#)
- shape_polygon, [87](#)
- shape_polygon(), [80](#), [81](#), [83](#), [84](#), [86](#), [87](#), [89](#), [91](#), [93](#), [95](#), [97](#), [100](#), [102](#), [104](#)
- shape_polygons (shape_polygon), [87](#)
- shape_raw, [16](#), [25](#), [76](#), [84](#), [88](#), [120](#)
- shape_raw(), [26](#), [76](#), [80](#), [81](#), [83](#), [84](#), [86](#), [88](#), [91](#), [93](#), [95](#), [97](#), [100](#), [102](#), [104](#)
- shape_raws (shape_raw), [88](#)
- shape_rectangle, [90](#)
- shape_rectangle(), [31](#), [80](#), [81](#), [83](#), [84](#), [86](#), [88–90](#), [93](#), [95](#), [97](#), [100](#), [102](#), [104](#)
- shape_rectangles (shape_rectangle), [90](#)
- shape_ribbon, [25](#), [92](#), [101](#)
- shape_ribbon(), [26](#), [29](#), [60](#), [68](#), [80](#), [81](#), [83](#), [84](#), [86](#), [88](#), [89](#), [91](#), [92](#), [95](#), [97](#), [100](#), [102](#), [104](#)
- shape_ribbons (shape_ribbon), [92](#)
- shape_square (shape_rectangle), [90](#)
- shape_square(), [90](#)
- shape_squares (shape_rectangle), [90](#)
- shape_star, [94](#)
- shape_star(), [80](#), [81](#), [83](#), [84](#), [86](#), [88](#), [89](#), [91](#), [93](#), [94](#), [97](#), [100](#), [102](#), [104](#)
- shape_stars (shape_star), [94](#)
- shape_stroke, [96](#), [100](#)
- shape_stroke(), [26–29](#), [32](#), [33](#), [36](#), [80](#), [81](#), [83](#), [84](#), [86](#), [88](#), [89](#), [91](#), [93](#), [95](#), [96](#), [98–100](#), [102](#), [104](#)
- shape_strokepath, [98](#)
- shape_strokepath(), [80](#), [81](#), [83](#), [84](#), [86](#), [88](#), [89](#), [91](#), [93](#), [95](#), [97](#), [98](#), [102](#), [104](#)
- shape_strokepaths (shape_strokepath), [98](#)
- shape_strokes (shape_stroke), [96](#)
- shape_twist, [25](#), [88](#), [100](#)
- shape_twist(), [4](#), [22](#), [23](#), [26](#), [60](#), [67](#), [68](#), [80](#), [81](#), [83](#), [84](#), [86](#), [88](#), [89](#), [91](#), [93](#), [95](#), [97](#), [100](#), [101](#), [104](#), [119](#)
- shape_twists (shape_twist), [100](#)
- shape_wedge, [103](#)
- shape_wedge(), [11](#), [80](#), [81](#), [83](#), [84](#), [86](#), [88](#), [89](#), [91](#), [93](#), [95](#), [97](#), [100](#), [102](#), [103](#)
- shape_wedges (shape_wedge), [103](#)
- sketch, [4](#), [11](#), [13–15](#), [17](#), [19–25](#), [65](#), [66](#), [76–95](#), [97](#), [98](#), [100–104](#), [105](#)
- sketch(), [4](#), [25](#), [26](#), [66](#), [108](#), [121](#)
- style, [17](#), [25](#), [66](#), [84](#), [89](#), [106](#), [121](#)
- style(), [4](#), [12](#), [14](#), [15](#), [17](#), [19](#), [21](#), [23](#), [25](#), [26](#), [29](#), [34](#), [66](#), [76](#), [79](#), [81](#), [83](#), [84](#), [86](#), [87](#), [89](#), [91](#), [93](#), [95](#), [97](#), [99](#), [101](#), [104](#), [106](#), [121](#)

- trans, [12](#), [14](#), [15](#), [17](#), [19](#), [21](#), [23](#), [25](#), [26](#), [66](#), [76](#), [79](#), [81](#), [83](#), [86](#), [87](#), [89](#), [91](#), [93](#), [95](#), [96](#), [99](#), [101](#), [104](#), [109](#), [110–113](#), [119](#)
- trans(), [110–118](#), [120](#)
- trans_affine, [110](#)
- trans_affine(), [26](#), [109](#), [111–118](#), [120](#)
- trans_chain, [25](#), [26](#), [66](#), [109](#), [110](#), [112](#), [119](#)
- trans_chain(), [109](#), [110](#), [112–118](#), [120](#)
- trans_fn, [25](#), [26](#), [66](#), [109](#), [110](#), [111](#), [119](#)
- trans_fn(), [109–111](#), [113–120](#)
- trans_identity, [113](#)

`trans_identity()`, [12](#), [14](#), [15](#), [17](#), [19](#), [21](#), [23](#),
[26](#), [66](#), [76](#), [79](#), [81](#), [83](#), [86](#), [87](#), [89](#), [91](#),
[93](#), [95](#), [96](#), [99](#), [101](#), [104](#), [109–112](#),
[114–118](#), [120](#)
`trans_reflect`, [113](#)
`trans_reflect()`, [26](#), [109–113](#), [115–118](#),
[120](#)
`trans_rotate`, [114](#)
`trans_rotate()`, [26](#), [109–114](#), [116–118](#), [120](#)
`trans_scale`, [115](#)
`trans_scale()`, [26](#), [109–115](#), [117](#), [118](#), [120](#)
`trans_shear`, [117](#)
`trans_shear()`, [26](#), [109–116](#), [118](#), [120](#)
`trans_translate`, [118](#)
`trans_translate()`, [26](#), [31](#), [109–117](#), [120](#)
`trans_warp`, [25](#), [26](#), [66](#), [109–112](#), [119](#)
`trans_warp()`, [109–118](#)

`withr::with_seed()`, [28](#), [54](#), [60](#)

`xy`, [17](#), [25](#), [76](#), [84](#), [89](#), [107](#), [112](#), [120](#)
`xy()`, [4](#), [25](#), [26](#), [66](#), [106](#), [108](#)